OULUN AMMATTIKORKEAKOULU

Alex Rapatti

Rootless Docker Containers in Continuous Integration

Rootless Docker Containers in Continuous Integration

Alex Rapatti
Bachelor's Thesis
Spring 2021
Information Technology
Oulu University of Applied Sciences

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Software Development

Author: Alex Rapatti
Title of the thesis: Rootless Docker Containers in Continuous Integration
Thesis examiner: Jukka Jauhiainen
Term and year of thesis completion: Spring 2021 Pages: 74

This thesis is a commission made by Nokia Corporation. The aim of the thesis is to collect information on a new container practice, Rootless containers, and integrate or replace Docker containers. These containers are used for software development environments, such as continuous integration pipelines. The change to rootless containers would improve the accessibility and security aspect of the containerized work environments of the developer teams, as they need to maintain the same work environments for efficiency and consistency.

The goal of the thesis is to inspect if the scripts that configure and run Docker containers with Linux distributions can be modified to have non-privileged users inside them, to ensure no user is required to have the administrative privileges to run and access them. This would be a major improvement in security, as having root privileges meant that a user could access anything inside the system with administrative rights. The research of the thesis is conducted from practical viewpoint and online sources and classes were used to gather knowledge on containers, continuous integration, Linux distributions, rootless containers and Docker.

While the implementations of modifying the Docker containers to run a non-root user inside the containers was deemed to be possible through trial and error, challenges were presented by the original scripts having and requiring root access in order to complete them. Further tests need to be conducted in order to integrate the changes into the official development work environments of the company. The developer teams of Nokia Corporation, however, can still utilize the information and practical knowledge gathered in this thesis and from the code modifications. It may also be useful for developers that inspect the Docker containers further and want to implement rootless containers for their own development work environments.

Keywords: containers, virtualization, Docker, rootless containers, continuous integration, Linux, container security

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikka, Ohjelmistokehityksen Suuntautumisvaihtoehto

Tekijä: Alex Rapatti
Opinnäytetyön nimi: Rootless Docker containers in Continuous Integration
Työn ohjaaja: Jukka Jauhiainen
Työn valmistumislukukausi- ja vuosi: Kevät 2021 Sivumäärä: 74

Tämä opinnäytetyö tilattiin Nokia Oyj:n toimesta. Yksi opinnäytetyön päätoiveista on kerätä tietoa uudesta konttimenetelmästä, juurettomista konteista, ja mahdollisesti integroida tai korvata niillä käytössä olevat Docker kontit. Docker kontteja hyödynnetään yrityksen ohjelmistokehitykseen liittyvissä työympäristöissä, kuten jatkuvan integraation automatisoiduissa testaus putkissa. Juurettomien konttien käyttöönotto edistäisi turvallisuutta ja mahdollistaisi konttien käyttöönoton kaikille yrityksen ohjelmistokehittäjille. Kontit ylläpitävät johdonmukaista ja tehokasta työntekoa takaamalla ohjelmistokehittäjille saman työympäristön.

Opinnäytetyön päämääränä on analysoida ja muokata Docker kontteja käynnistäviä ja konfiguroivia skriptejä. Ideana on mahdollistaa, että Linux-jakelupaketteihin perustuva käyttäjä voisi ilman juurioikeuksia kirjautua kontin sisälle. Juurioikeudet vastaavat Linux ympäristössä kaikkia hallinnollisia oikeuksia, minkä vuoksi niiden pois ottaminen käytöstä on suotuisaa kontin turvallisuuden kannalta. Opinnäytetyön tutkimusvaiheet perustuvat käytännön kokeiluihin ja opiskeluihin konttiteknologiasta, jatkuvasta kehityksestä, Linux-jakelupaketeista, juurettomista konteista ja Dockerista.

Vaikka skriptien muutokset ja tavallisen käyttäjän implementointi Docker kontteihin huomattiin olevan mahdollista, haasteita ilmeni siksi koska alkuperäiset skriptit toimivat ja ajoivat Docker kontteja juurioikeuksilla. Lisätutkimuksia ja testejä vaaditaan, jotta muutoksia voidaan harkita yrityksen viralliseen käyttöön. Ohjelmistokehittäjät Nokia Oyj:n puolella pystyvät kuitenkin hyödyntämään opinnäytetyöstä saatua teoreettista ja käytännöllistä tietoa skripteihin liittyvien muutosten perusteella. Teos saattaa myös olla hyödyllinen ohjelmistokehittäjille, jotka haluavat tehdä lisätutkimuksia Docker kontteihin liittyen ja haluavat mahdollisesti implementoida juurettomia kontteja omiin työympäristöihin.

CONTENTS

ABBREVATIONS

CI	Continuous integration. Merging code changes from developers and executing tests in an automated manner to maintain software development processes.
CD	Continuous delivery. An approach to produce software in short cycles, allowing the release of software at any given moment.
DevOps	Development and operations. A set of practices that work to automate and integrate the processes between software development and information and communication operations. The aim is to shorten the systems development life cycle, performing building, testing and releasing of software faster.
CPU	Central processing unit. The portion of a computer that retrieves and executes instructions.
OS	Operating system. A system software that manages the hardware and resources of a computer and provides common services for computer programs.
CLI	Command line. A text-based user interface to the computer.
API	Application programming interface. A software intermediary that allows two applications to talk to each other.
REST API	Representational state transfer, an application programming interface that allows interactions with web services to request to access and use data.
UNIX	Uniplexed information computing system. A portable, multitasking, multiuser, time-sharing operating system.
IT	Information technology. Refers to anything related to computing technology, such as networking and software.

JSON	JavaScript Object Notation. An open standard file format that uses human-readable text to store and transmit data objects.
YAML	YAML Ain't Markup Language. A human-readable data serialization language.
LXC	Linux Containers. An operating-system-level virtualization method for running multiple isolated Linux system containers.
URL	Uniform resource locator. The address of a web page.
VCS	Version Control System. A system that records changes to a file or set of files over time to enable reverting back to specific versions later.
IP	Internet Protocol. The set of rules governing the format of data sent via the internet or local network. IP addresses are the identifier that allows information to be sent between devices on a network.
UPD	User Datagram Protocol. A protocol used for data transferring. Known as a stateless protocol, which means there will not be any acknowledgement of the status of the sent packages.
TCP	Transmission Control Protocol. The most used protocol on the Internet for data transferring.
PID	Process identifier, also known as process ID. Represents a number used by most operating system kernels, such as Unix and Windows to uniquely identify an active process.
UID	Unique identifier, also known as user identification. An identifier for a specific user of a computer system.
GID	Group identifier. An identifier for a group that holds a set of users. Group permissions apply to its group members.

VETH	Virtual Ethernet Devices. Act as tunnels between network namespaces to create a bridge to a physical network device in another namespace, can also be used as a standalone network device.
VFS	Virtual File System. An abstract layer on top of a more concrete file system.

1 INTRODUCTION

As modern software development practices, environments and manufacturing have advanced and become more complex to build high-quality, reliable and scalable applications, simplifying and optimizing the software development stages can be crucial regarding the maintaining and managing developer work environments and production delivery [1]. Automated and concise testing and deployment directed by CI (Continuous integration) and CD (Continuous delivery) allows developers to independently add features and maintain well-defined pieces of the software, and new members can engage with the code since understanding a smaller section of the it is all they have to start with [2].

Containers, particularly Docker containers that have gained and maintained popularity, have made their way into the software industry by introducing a way to simplify and isolate development environments, such as CI processes and applications, by packaging them with all their dependencies, tools, libraries and operating systems. The containers remain lightweight, offering easy distribution inside extensive development teams. Container technology itself is not a new practice, being introduced in 1979 with Unix version 7 and the *chroot* system that restricted the software access to resemble an isolated environment. Isolated processes, however, did not progress until they were picked up again in the 2000s. This era introduced new container systems to isolate the usage of resources, such as the CPU (Central processing unit) and memory. The most complete and stable version of container technology was LXC (Linux Containers) in 2008, which worked as the base for the current market-leading technology, Docker [3]. The convenient qualities of Docker, such as fast scaling, flexibility and software delivery capabilities with multiple operating systems, are behind its success [4].

Despite the promising features of Docker, this platform has its flaws in security. This has raised concerns for container safety, as containers have various use cases from automating CI tasks to holding development work environments. A default Docker container run requires root access, which leads to it running on top of a Linux environment, as the administrative user. This practice, for instance, can open a door for malicious acts with all the root capabilities of the host machine [5]. The security questions regarding Docker containers have led to the existence of rootless containers, which are examined in this thesis.

This thesis was requested by Nokia Corporation, a multinational telecommunications and information technology company. Their request was to inspect if their Docker containers used in CI and software development environment control can be replaced with rootless ones, and if so, proceed with the implementations. These modifications would allow users to access the containers without root privileges, which alone would improve accessibility and security of the container practices of the company. The project background will be covered in the next section.

The rest of the thesis consists of theoretical chapters that cover topics needed to understand the company requested project collectively. These sections include topics that cover containers, Docker, CI, the use cases of Docker containers in CI and distribution of work environments between large-scale developer teams. Section 6 will also explain what rootless means in container context, showcase ways of assembling a rootless container and ponder why the usage of rootless containers is something to consider in containerized environments. Section 6.4 showcases the result and section 7 holds the conclusions.

2 PROJECT BACKGROUND

2.1 Project goals

Nokia Corporation made a request to have their containerized environments inspected in order to see if they can be switched to use rootless containers, a new way of utilizing containers. If the task were proven to be realistic within the time frame given, it would be implemented and put in use. The Docker containers are used for development work environment control and for tasks created by CI, such as pipelines and their tests. Section 5 covers the topic of Docker containers in CI. As Docker containers run with root privileges by default, rootless containers would improve accessibility for multiple software developer teams from CI to DevOps (Development & Operations) because they will be able to run and maintain the same work environments without having root access for their accounts. Root is discussed in detail in section 6.1. This change would also improve security, which will be covered in section 6.2.

2.2 Project environment and requirements

The project began by studying the terms used in the requested thesis, such as Docker, containers, Linux and rootless containers. The project plan was used to specify the work environment for testing the modifications made to the scripts used by the developer teams. The scripts that needed changes varied from Python 3 to bash scripts, and the version control for the source code was Git (covered in section 5.2.2). The main request of the project was to enable the changes to be compatible with the current working container system of the company, which was running with root access. As the rootless options were introduced, a minimal requirement was set: to test their current Docker containers with a user switch, a *--user* parameter that would change the root user to a regular user inside the container. Another container engine, Podman, was also introduced for research purposes, as it works with rootless containers by default and is compatible with Docker.

The first test environments were held in virtual machines with different Linux distributions, such as Fedora, CentOS and Ubuntu, over a host computer with Windows operating system. Fedora was deemed to be the target distribution, as it resembled the distribution used for the current work environments of the company. Nokia also provided their servers for final tests, as they would

resemble their work environments the best. The tools, development and CI environments and details of development teams of Nokia will not be discussed further in this thesis to protect the privacy of the company.

3 CONTAINERS

Software containers add a new layer of virtualization (section 3.1) that abstracts the interface between software and operating system; instead of installing the applications on the operating system, they are built into containers that can be run on their supported platforms. This abstraction of applications offers a logical packaging mechanism; containers package the application and its dependencies together, allowing version control and extensive replication across developers and machines. Containerization also lets developers to focus on their application logic and dependencies, without having to concern themselves with specific software versions or configurations specific to the application regarding deployment and management. Work environments stay consistent, clean and separate from other applications, regardless of where they are deployed. This ease in deployment is possible because containers virtualize CPU, memory, storage and network resources at the operating system level, and all the operating systems from Linux, Windows and Mac are operatable. The features containers introduce bring greater agility and productivity while easing development and testing, since developer teams spend less time debugging and diagnosing issues in their applications and environments, and the time gained is shifted to developing and delivering new features. Developers can also form assumptions of the predictable environments, be it for testing or development; because of the consistency that containers provide [6].

3.1 Containers and virtual machines

Virtualization is technology that can generate multiple simulated environments or dedicated resources from one physical hardware system. A software called a hypervisor connects directly to that hardware and allows splitting a system into separate, distinct, and secure environments known as virtual machines. The physical hardware, equipped with a hypervisor, is called the host and the virtual machines use its resources as guests. The guests can receive the resources they need when they need them, because they treat computing resources, such as CPU, memory and storage, as a pool of resources that are relocatable and the host machine is able to distribute these resources appropriately [7].

Virtualized environments, such as virtual machines, are often compared with containers (FIGURE 1). They have similar features, such as packaging computing environments that combine various components and isolate them from the rest of the system; but their differences come out in scaling and portability. A virtual machine usually contains their own, complete operating system, allowing them to perform multiple resource-intensive functions at once [8]. Containers, however, attempt to package only the application and all the resources necessary for it to run, taking up much less space than virtual machines by sharing the OS kernel with other containers, each running an isolated process in user space [9]. While there are more resources available for virtual machines to abstract, split, duplicate and emulate entire servers and operating systems, the lightweight nature of containers and their shared operating system constructs them to be much easier to move across different environments [8].

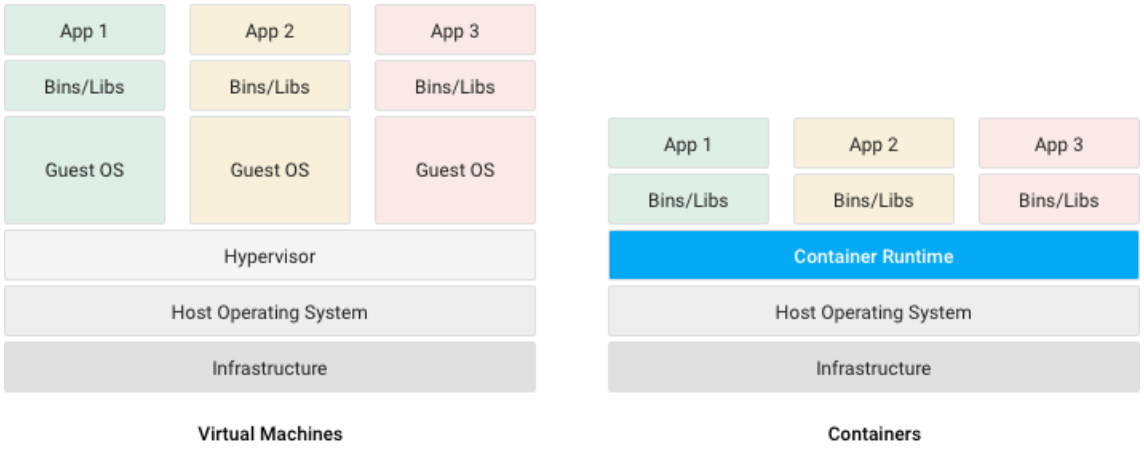

FIGURE 1. The structure difference between virtual machines and containers (6)

The use case of containers and virtual machines depend on the user's needs. A lightweight container fits a small use instance that needs to be portable, versus a permanent allocation of a virtual machine; containers can be moved across bare metal systems as well as "public", "private", "hybrid" and "multi-cloud" environments. Cloud-native apps built by containers speed up application development and deployment process and optimization of existing ones. Besides scalable projects that are easier to transfer and distribute with containers' assistance, IT practices such as CI/CD and DevOps practices, also benefit from containers; the smallest possible units that hold the workload can be packaged in containers, its isolation aspect allowing developers to work on individual parts of an application or service without threatening code in other containers or environments. Regarding instances that require holding risky development cycles and providing

infrastructural resources and heavy workloads, virtual machines are considered more suitable for the task because they have expanded functionality and resources [8].

3.2 Microservices

Microservices are a structural design for building a distributed application. They disassemble an application into independent, loosely coupled, individually deployable services. Microservices acquired their name because each important function of an application operates as an independent proxy. This allows each service to scale or update without disrupting other services in the application, enabling automated CI/CD pipelines and deployment of applications. Together, containers and microservices create a massively scalable and distributable system and has led to container engines, such as Docker, to become a standard way of deploying microservices by packaging them into container images and deploying each service instance as a container [10].

Containers with microservices can be divided into two categories: stateless and stateful microservices. A stateless microservice does not save or store data but handles requests and returns responses. Anything stored is lost when the container is restarted. A stateful microservice requires storage to run and allows microservices to read and write data saved in a database. A restarted container does not lose its data [10].

Microservices architecture was designed to resolve issues with the traditional way of building applications, which was guided by monolithic architecture (FIGURE 2). Monolithic application is a single, integrated software instance that resides on a single server. It has an extensive life cycle with infrequent updates, and any changes made can affect the entire application. For developers to add new features, reconfiguring and updating the entire stack was required. This made application development more costly and time-consuming. In microservices architecture, application is broken into modular components and it can be distributed across clouds and data centers. In practice, this architectural way creates all services individually and deploys them separately, allowing autoscaling. A developer must only update the individual microservice to add a new feature. When dozens of monolithic applications are modernized into microservice based applications, provisioning an extensive number of containers is not maintainable nor scalable by discrete appliances from the past. This is where container orchestration offers a solution [10].

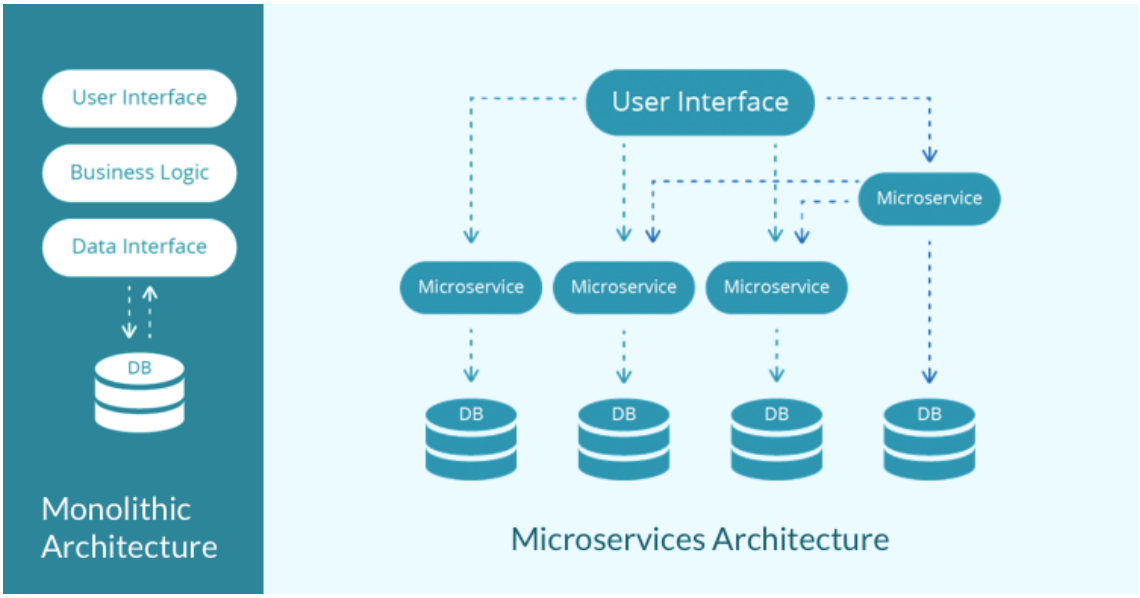

FIGURE 2. A monolithic architecture compared to microservices architecture (11)

3.3 Container orchestration

Organizing and managing containers can become a task on its own when container usage becomes prominent with a substantial number of containers in use. This is where container orchestration comes along, to automate the deployment, management, scaling and networking of containers. It helps developers to deploy the same, configured application across different environments without needing to design it again. There are many orchestration tools available for managing containers and their lifecycles, with popular ones being Kubernetes and Docker Swarm [12].

Kubernetes

Kubernetes, also known as K8s, is an open-source container orchestration tool, designed by Google and donated to Cloud Native Computing Foundation in 2015. This tool can build application services that span multiple containers, schedule them across a cluster and scale them. This automated process eliminates many of the manual practices involved in deploying and scaling containerized applications. In production environment, Kubernetes helps users to implement and rely on a container-based infrastructure [12]. Kubernetes' features involve automated rollouts, which allows Kubernetes to roll out changes to the application or its configuration, while monitoring application health to ensure it does not kill all the instances at the same time. If an error occurs,

Kubernetes rolls back the changes. Kubernetes also mounts the storage system of choice automatically, and the configuration management ensures the secrets the user wants to keep are not exposed in stack configurations when deploying and updating application configurations. Container health is at the core of Kubernetes usage, restarting and killing containers that do not pass user defined health checks [13].

Kubernetes, resembling other orchestration tools, works with configuration management. A developer can specify the configuration of an application using either a YAML or JSON file, also known as a compose file. This file will direct the configuration management tool to the container image it needs to use, while also configuring network establishment and the location to use for log storage. When a new container is deployed, the container management tool schedules the deployment to a cluster and considers any requirements or restrictions needed. Based on the configurations in the compose file, the orchestration tool can manage the lifecycle of the container. The repeatable patterns of Kubernetes allow developers to manage, configure and scale their containers and services in any environment [12].

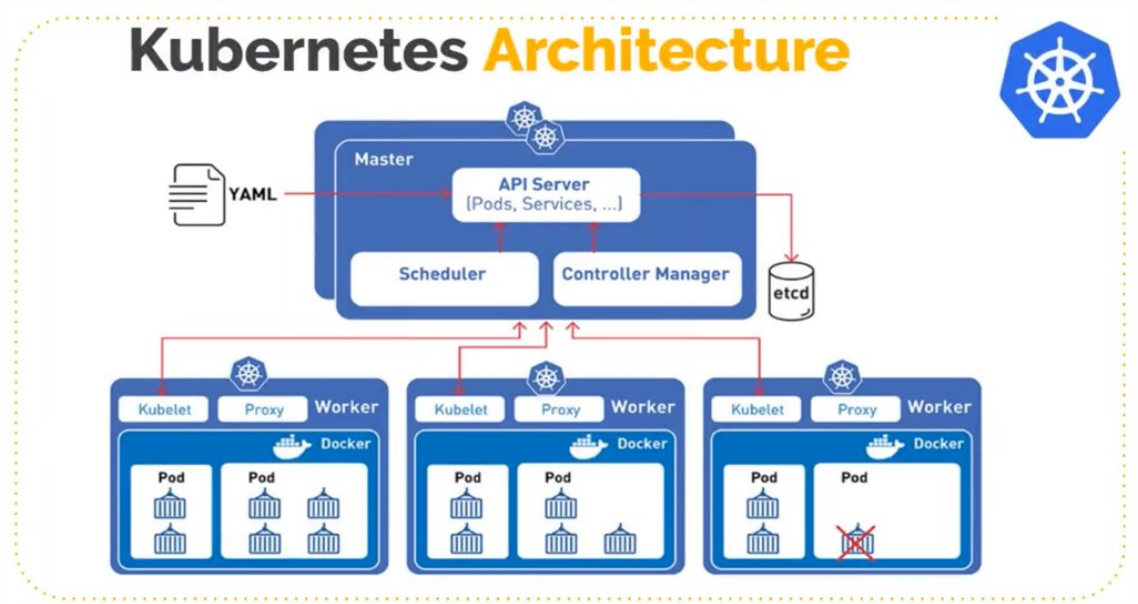

FIGURE 3. Kubernetes architecture with Docker containers (14)

As shown in FIGURE 3, Kubernetes patterns are constructed with Kubernetes components. “The worker nodes” hold the pods that are the components of the application workload. The Control plane, also known as master, manages the worker nodes and the pods in the cluster. “Cluster”

comprises of a set of worker machines, with each cluster owning at least one worker node. The control plane tends to run across multiple computers, while cluster runs multiple nodes, which leads to fault-tolerance and high availability. Control plane component, scheduler, watches for newly created pod and assigns a node for them to run on. Controller Manager runs controller processes, such as node controller, job controller, endpoint controller and service account and token controller; these controllers are separate processes with their own tasks to reduce complexity, but they are all still complied into a single binary and run in a single process [15]. The Kubernetes API is the core of Kubernetes' control plane, allowing communication between end users, cluster parts and components [16]. *Kubelet* and *kube-proxy* are components of the worker nodes; *Kubelet* is an agent that runs on each node in the cluster, to ensure the containers are running in a pod. *Kubelet* does not manage containers; only ensures that the containers are running and healthy based on the specifications set. *Kube-proxy* is a network proxy for each node in the cluster, to maintain network rules. *Etcd* works as a backup storage for Kubernetes [15].

4 DOCKER

Docker, launched in 2013, is an open platform for developing, testing, shipping and running applications using containers and their images, achieving a significant decrease in delay between writing code and delivering it. This effect comes from the ability to share isolated, standardized, lightweight and packaged environments between developers for any of their work environments, from physical to virtual machines. With Docker, container becomes the unit for distributing, testing and deploying applications into production environment [17].

Docker stands out from other container engines by setting industry standards, and their containers can be found utilized anywhere. Their engine leverages existing computing concepts around containers and specifically in Linux world, primitives known as cgroups (control groups) and namespaces [9]. Namespaces is a technology to provide the isolated workspace for containers; when a container is running, Docker creates a set of namespaces for that container. This feature is usable for Docker because of its programming language Go, which takes advantage of several features of the Linux kernel [17]. Docker, however, is unique for its focus on the requirements of developers and systems operators to separate application dependencies from infrastructure. The success with Linux allowed Docker company to create a partnership with Microsoft, which in return introduced Docker containers into Windows Servers. Additionally, technology available from Docker and its open-source project, Moby, has been leveraged by all major data center vendors and cloud providers. The leading open source serverless frameworks also utilize Docker container technology. Besides democratizing software containers and developing a Linux container technology, Docker has donated the container image specification and runtime code to the Open Container Initiative (OCI) to help establish consistency as the container ecosystem grows and evolves [9].

4.1 Docker architecture

FIGURE 4 showcases the client-server architecture of Docker. The Docker client is the primary way to interact with Docker. Docker commands, such as *docker run*, are sent to Docker daemon, which handles building, running and distributing containers with Docker objects. The Docker client and daemon can run on the same system or Docker client can be connected to a remote daemon.

The communication occurs through REST API, over UNIX sockets or a network interface. Docker Compose is another Docker client if a user works with applications consisting of a set of containers. Docker objects and registries will be discussed in the next chapters.

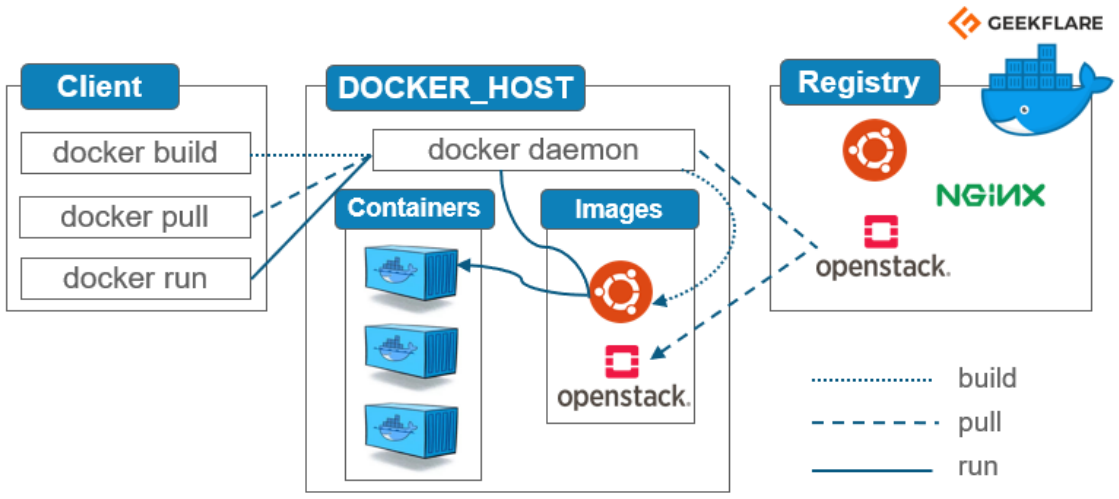

FIGURE 4. A diagram of the architecture of Docker (17)

4.2 Docker objects

4.2.1 Images

An image is a read-only layout with instructions for creating a Docker container. Images (FIGURE 5) tend to be based on other images, with additional configurations and applications the user can specify. Images can also be obtained from registries, such as Docker hub, to be used as they are [17].

```
$ docker images

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              77af4d6b9913        19 hours ago        1.089 GB
committ             latest              b6fa739cedf5        19 hours ago        1.089 GB
<none>              <none>              78a85c484f71        19 hours ago        1.089 GB
docker              latest              30557a29d5ab        20 hours ago        1.089 GB
<none>              <none>              5ed6274db6ce        24 hours ago        1.089 GB
postgres            9                   746b819f315e        4 days ago          213.4 MB
postgres            9.3                 746b819f315e        4 days ago          213.4 MB
postgres            9.3.5               746b819f315e        4 days ago          213.4 MB
postgres            latest              746b819f315e        4 days ago          213.4 MB
```

FIGURE 5. A Docker command to list images a user has stored locally (18)

In order to create an image, a Dockerfile must define the steps needed for the image. Each instruction in a Dockerfile generates a layer [16], showcased in FIGURE 6. The image has a base layer which is read-only, and the top layer that can be written [19]. When the Dockerfile is changed and the image has been rebuilt, those layers will be changed. This method of rebuilding is the reason why images remain lightweight and fast compared to other virtualization technologies [17]. As a container is started, Docker only creates the thin writable container layer [20].

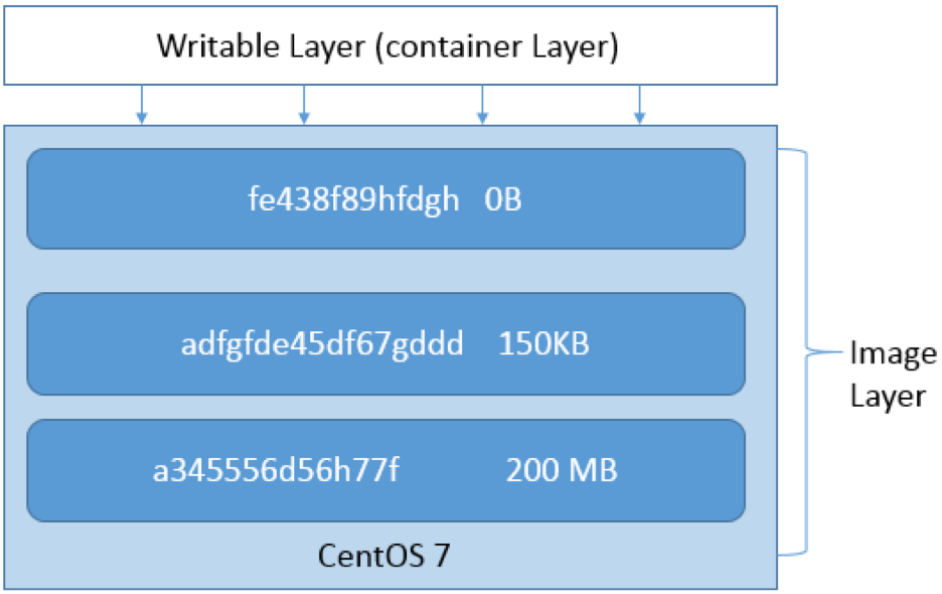

FIGURE 6. A diagram of container image layers with CentOS 7 as base image (20)

4.2.2 Containers

A container is a runnable instance of an image, defined by images instructions and the configuration options given when starting or running a container. Containers use the Docker API or CLI (Command line) to execute Docker commands, such as *create*, *start* and *stop*. Networks and storages can be attached to a container, and a new container state can be used to create a new image. Isolating the container from other containers and host machines can be controlled by modifying how isolated they are with the network, storage, or other underlying subsystems of a container [17]. When a container is removed, any changes that are not stored in persistent storage disappear because the writable layer is also deleted. Only the underlying image remains unchanged [20].

```
$ docker run -i -t ubuntu /bin/bash
```

FIGURE 7. docker run command with an ubuntu image (17)

As an example, FIGURE 7 showcases how to start an *ubuntu* container and execute it with */bin/bash*. If a user does not have the ubuntu image locally, Docker pulls it from registry to replicate *docker pull ubuntu* command. Once the container creation process is running, Docker allocates a read-write filesystem to the container, as its final layer, to allow the container to create or modify files and directories in its local filesystem. If networking options are not defined, Docker creates a network interface to connect the container to the default network; this includes assigning an IP (Internet Protocol) address to the container. The command flags *-i* and *-t* run the container in an interactive mode, allowing input from keyboard and output from terminal. When the user types exit to terminate the */bin/bash* command, the container stops but is not removed [17]. A command *docker ps* (FIGURE 8) gives the user a list of running containers, while *docker ps* with *-a* flag lists all the containers that have not been removed [21].

```
$ docker ps

CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS              PORTS
4c01db0b339c        ubuntu:12.04                   bash                     17 seconds ago      Up 16 seconds       3300-3310/tcp
d7886598dbe2        crosbymichael/redis:latest     /redis-server --dir      33 minutes ago      Up 33 minutes       6379/tcp
```

FIGURE 8. A list of running containers (21)

4.2.3 Volumes

In order to persist data generated by and used by Docker containers, volumes are preferred over bind mounts because they can be managed using Docker CLI commands or the Docker API and they do not increase the size of the containers using it. Additionally, volumes work on both Windows and Linux containers. The contents of the volumes exist outside the lifecycle of a given container, and volumes offer higher performance and safety over mounting. *Tmpfs* mounting is an option if a container generates non-persistent state data, to avoid storing it anywhere permanently (FIGURE 9). With both volumes and *tmpfs* mounting, the goal is to avoid writing into the writable layer of the container to maintain the size of it. Starting a container with a volume requires a *-v* or a *--mount* flag added to the run command, which is showcased in FIGURE 10. The instance shown mounts the volume “myvol2” into “/app/” in the container [22].

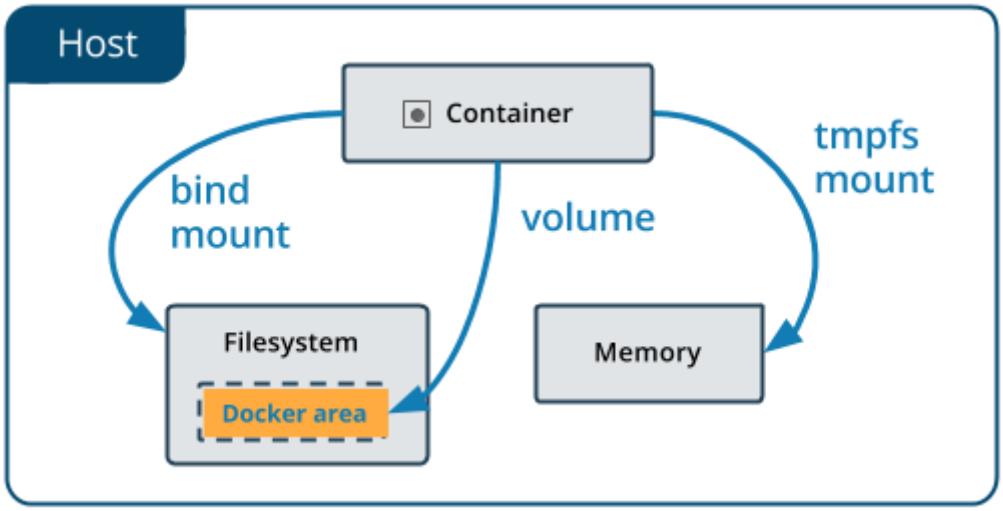

FIGURE 9. Types of mounts for a Docker container (22)

```
$ docker run -d \
  --name devtest \
  --mount source=myvol2,target=/app \
  nginx:latest
```

```
$ docker run -d \
  --name devtest \
  -v myvol2:/app \
  nginx:latest
```

FIGURE 10. Docker run command with mount and volume flags (22)

4.2.4 Networks

The ability of Docker containers to connect to each other and to non-Docker workloads, allows communication between containers and services. Whether Docker hosts are running Linux, Windows, or a mix of the two, Docker is available for use to manage them anywhere [23].

There are mainly five network drivers in Docker: *bridge*, *host*, *overlay*, *none* and *macvlan*. Bridge is the default network drive for a container. This network is utilized when an application is running on standalone containers, to enable communication between containers and their shared host. Host network driver removes the network isolation between Docker containers and Docker host. Overlay network enables swarm services to communicate with each other. It is used when the containers are running on different Docker hosts. “None” driver option disables all the networking, and *macvlan* driver assigns mac addresses to containers to enable them to resemble physical devices; this is useful in cases of migrating a virtual machine setup [19].

4.3 Docker registries

Docker registries are stateless, server-side applications that store Docker images and allow users to distribute them [24]. Docker registries can be public or private, with Docker hub being the default location for public Docker images. *Docker pull* and *docker run* commands pull Docker images from the configured registry, whereas *docker push* command stores the image on that registry (FIGURE 11) [19].

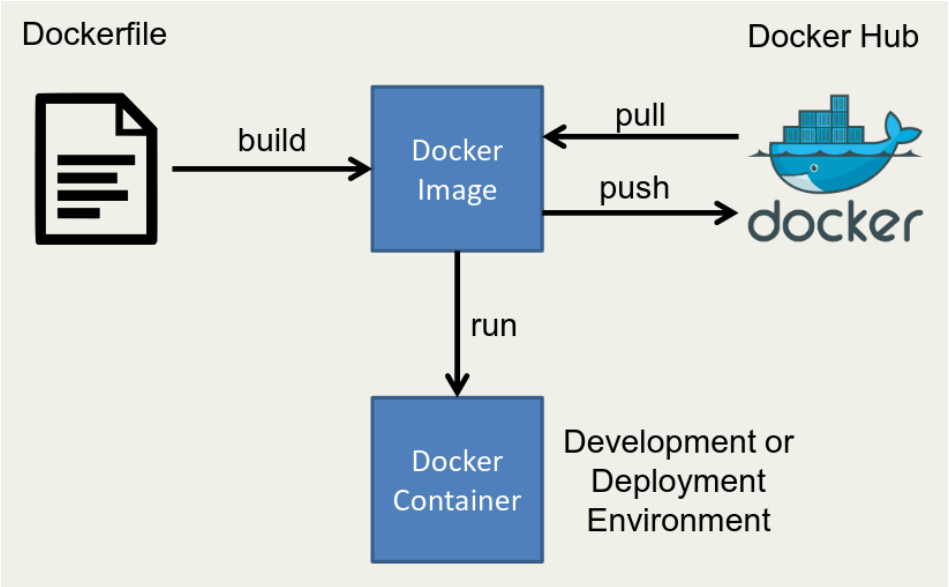

FIGURE 11. A diagram to show the commands between Dockerfile and Docker registries (25)

4.4 Dockerfile

A Dockerfile is a text file that contains all commands that are required for building an image. The order is important, as the commands are executed after each other. These commands are called instructions, and each instruction creates a read-only layers for the image. When images are running and a container is generated, a new writable layer is added on top of the underlying layers that were previously created by instructions. All the modifications from the running container, such as altering files, are written to this container layer [26].

In order to understand the structure of a Dockerfile, its instruction commands must be comprehended. As in FIGURE 12, Dockerfile starts with *FROM* instruction, which initializes new build stage and sets the base image for subsequent instructions. This instruction is needed for each

Dockerfile, and it must consist a valid image; pulling an image from a public repository stored in Docker hub is a recommended way of starting [27]. The command that comes after, *RUN*, executes any commands in a new layer on top of the current image and commits the results. These results gained are used for the step that comes next in the Dockerfile. Layering *RUN* instructions and producing commits is the core concept of Docker, where commits are cheap, and containers can be made from any point in the history of an image. [27]. The most common use-case for *RUN* is an application of *apt-get* because it installs packages [26].

Regarding instances of copying local files into the container, *COPY* and *ADD* commands are in use. The given example in FIGURE 12 relies on *COPY*, as its transparency enables it to be more preferrable over *ADD* despite sharing similar functionalities. Using *ADD* is also discouraged for package fetching from remote URLs to avoid increasing the image size extensively; commands such as "curl" and "wget" will do the same while allowing file deletion after they have been extracted. *ADD* is, however, a recommended command for local tar file auto-extraction. If Dockerfile has multiple steps that require different files, it is encouraged to use *COPY* on each file individually rather than all at once to ensure that each build cache of a step is only invalidated if the specifically required files change. This results in fewer cache invalidations for the *RUN* step [26].

The *WORKDIR* instruction is stated to "set the working directory for any *RUN*, *CMD*, *ENTRYPOINT*, *COPY*, and *ADD* instructions that come after it in the Dockerfile" according to Docker documents. A common working directory format is *WORKDIR /path/to/workdir*. If *WORKDIR* does not exist, it will be created despite its lack of usage in the Dockerfile steps. If *WORKDIR* is used multiple times in a row in Dockerfile, it will create a relative path to the previous *WORKDIR* instructions. For example, if working directories of *WORKDIR /a*, *WORKDIR b* and *WORKDIR c* were placed after one another, this would result into a final path of */a/b/c* [26].

The *CMD* instruction is used to run the software contained in the image, along with any arguments. In most cases, *CMD* should be given an interactive shell, such as *Bash*, *Python* and *Perl*. For example, a container running in interactive mode (*docker run -it python)*, enables the user to be placed into a usable shell named in the command. [26].

One of the last instructions of Dockerfile (FIGURE 12) is *ENTRYPOINT*. This instruction is commonly used to set the main command of the image, allowing that image to be run as thought it was the command. The *ENTRYPOINT* can also be used in combination with a helper script,

allowing users to setup more of the required and complex steps that are not possible with *CMD* instructions alone, and to keep the Dockerfile simple. The helper script is copied into the container and run via *ENTRYPOINT* on container start. This ensures that the default application in the image is targeted when the container is created [26].

```
# syntax=docker/dockerfile:1
FROM golang:1.16-alpine AS build

# Install tools required for project
# Run `docker build --no-cache .` to update dependencies
RUN apk add --no-cache git
RUN go get github.com/golang/dep/cmd/dep

# List project dependencies with Gopkg.toml and Gopkg.lock
# These layers are only re-built when Gopkg files are updated
COPY Gopkg.lock Gopkg.toml /go/src/project/
WORKDIR /go/src/project/
# Install library dependencies
RUN dep ensure -vendor-only

# Copy the entire project and build it
# This layer is rebuilt when a file changes in the project directory
COPY . /go/src/project/
RUN go build -o /bin/project

# This results in a single layer image
FROM scratch
COPY --from=build /bin/project /bin/project
ENTRYPOINT ["/bin/project"]
CMD ["--help"]
```

FIGURE 12. A Dockerfile example for a Go application (26)

When the Dockerfile has been structured, the *docker build* command is utilized to build a new Docker image, showcased in FIGURE 13. The command can have additional arguments, such as the flag *-t*, to give the image a tag; a human-readable name for the final image. This enables the reference to the image when its running as a container. A directory path or a *URL* must be given to specify where the files for the build can be found, such as Dockerfile and the files given through *ADD* or *COPY* commands inside the Dockerfile. Adding "." at the end of the *docker build* command indicates that Docker should look for the Dockerfile in the current directory [28]. The *docker run* command with the image name executes the fresh image as a container.

```
docker build -t getting-started .
```

FIGURE 13. A command to build a new image from Dockerfile (28)

5 CONTINUOUS INTEGRATION

“Build your team’s agility with faster feedback. Because you only move as fast as your tests” is stated by Atlassian, a company known for improving software development and project management with popular products, such as Jira and Confluence, on their dedicated web site that overviews CI/CD practices, their importance, challenges and benefits. CI practice automates the integration of code changes from multiple developers into a single software project. It is a primary DevOps practice, allowing developers to frequently merge their changes into a repository where builds and tests run. Before integration, the automated tools ensure the changed code is formed correctly and passes the required tests [29]. Since source code version control is the key to CI process, it will be overviewed in section 5.2.

In order to understand why CI is important, the absence of CI and its issues must be comprehended. Without CI, developers would have to contribute their code manually and communicate the changes with everyone inside the organization. Responsibilities must be assigned between teams, and feature launches and fixes must be communicated clearly. This communication can become complex and entangled because of how much synchronization it requires. This will lead to slower code releases with high failure rates because integrations require a significant amount of communication, thoughtfulness and organizing between developers; and this becomes even more of an issue as the size of the teams and codebase increases. Delivery time estimations are harder to form, as developers have no certain expectations of how fast new changes can be integrated [29].

CI is often used with an agile software development workflow. An organization will compile list of tasks that comprise a product roadmap, and these tasks are then distributed amongst software developer teams for delivery. When CI practices are in use, software developers can work independently on features in parallel, increasing delivery output from engineering teams. When changes are ready to be merged, the features can be added fast and without spending time on extensive communication. Therefore, CI is a valuable practice in modern, high performance software engineering organizations. CI also benefits the organization as whole; it enables better transparency into the process of software development and delivery. These benefits enable the rest of the organization to better plan and execute *go to market* strategies [29].

5.1 Test automation

Test automation is an important aspect of any CI/CD pipeline. Without automated testing, CI/CD benefits cannot be applied to the software development environment. Automated tests include unit testing, integration testing and system testing. Teams are also required to add testing for functionality, usability, performance, load, stress and security [30].

Regarding the main three automated tests, a unit test covers units of code such as methods, classes and API services. An effective unit testing increases test coverage for the whole system. Integration testing ensures modules and parts of the system work together smoothly, and system tests run on the entire system to simulate active users. This test environment should be as close to the production environment as possible [30].

FIGURE 14 provides an example of how the automated tests are placed in pipelines. The main idea of a CI/CD pipeline is to deliver a unit of change through three phases of an automated software release pipeline: continuous integration, deployment and delivery. The integration phase is the first step in the process and covers the process of multiple developers merging their code changes with the master code repository of a project [29]. Then the change is integrated from repository into a build and automated tests will be run against the build [30]. Continuous delivery is the next extension of CI; its responsible for packaging an application together to be delivered to end-users. This phase runs automated building tools to generate this application. Continuous deployment is the last aspect of the pipeline. It launches and distributes the application to end-users. At this point the application has successfully passed the integration and delivery phases. Scripts and tools can be used to automatically deploy and distribute the product [29].

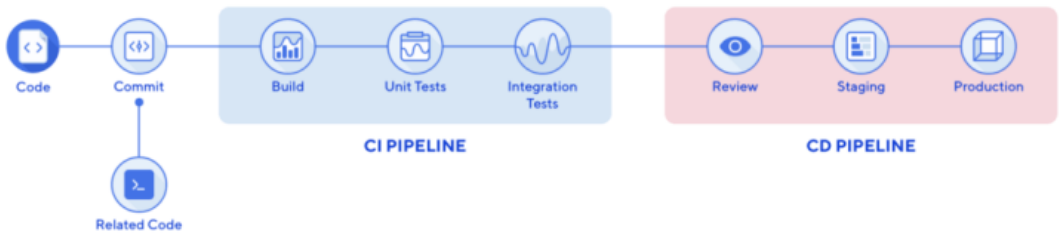

FIGURE 14. An example of a CI/CD workflow pipeline (30)

5.2 Version Control

As stated in chapter 5, source code version control is the key to CI process because a VCS (Version Control System) begins the CI in CI/CD pipelines. The VCS creates the source code of record and organizes it for the build [31]. This enables the reverting of selected files or entire projects back to a previous state. VCS also offers utilities to compare changes over time and see information of the developers responsible of the modifications made. Recovering files is also possible with this system [32].

5.2.1 Version Control Systems

In order to understand where Version Control Systems came from and why they are in use, it is important to understand their history. VCSs gained their roots from simple version control practices, such as a user copying their files into another directory. Its simplicity, however, did not compensate for its weakness to errors. A user could, for example, write to the wrong file or copy over files that were not intended to be changed. In order to combat this issue, local VCSs were invented with a simple database that kept all the changes to files under revision control (FIGURE 15). One of the most popular VCS tools was a system called RCS (Revision Control System), which is still in use today. The system works by keeping patch sets, which held the differences between files, in a special format on disk so it could re-create any file at any point in time by adding up the patches [32].

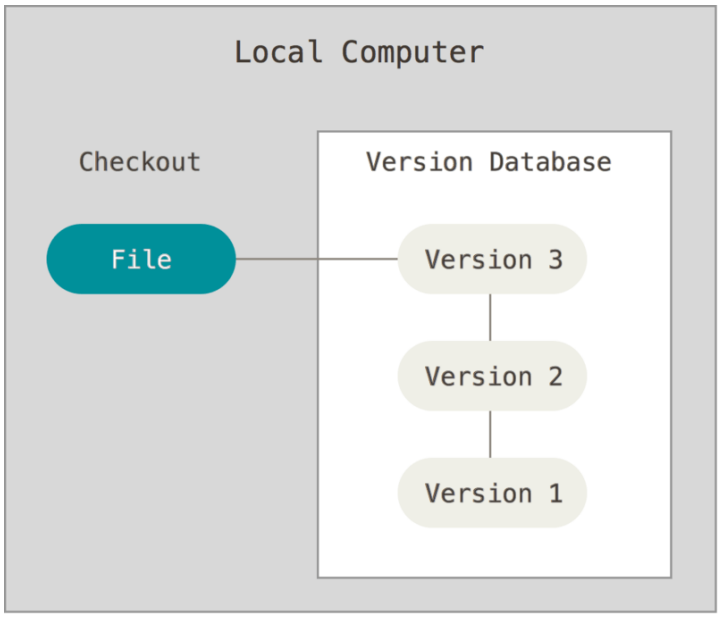

FIGURE 15. An example of a local version control system (32)

After local VCSs, Centralized Version Control Systems (CVCS) were developed to allow collaboration with developers on other systems. These systems, such as CVS, Subversion and Perforce, have a single server that contains all the versioned files, and several clients that obtain files from a central place (FIGURE 16). This has been the standard for version control. The main advantages come from knowing what everyone else on the project is doing, and administrators have control over who can do what. It is also easier to administrate a CVCS than it is to deal with local databases on every client. The main downside, however, is the importance of the server on the project; if that server goes down for an hour, then during that hour no one can collaborate or save versioned changes to anything they are working on. And if the hard disk of the central database becomes corrupted and there are no proper backups, the project loses everything [32].

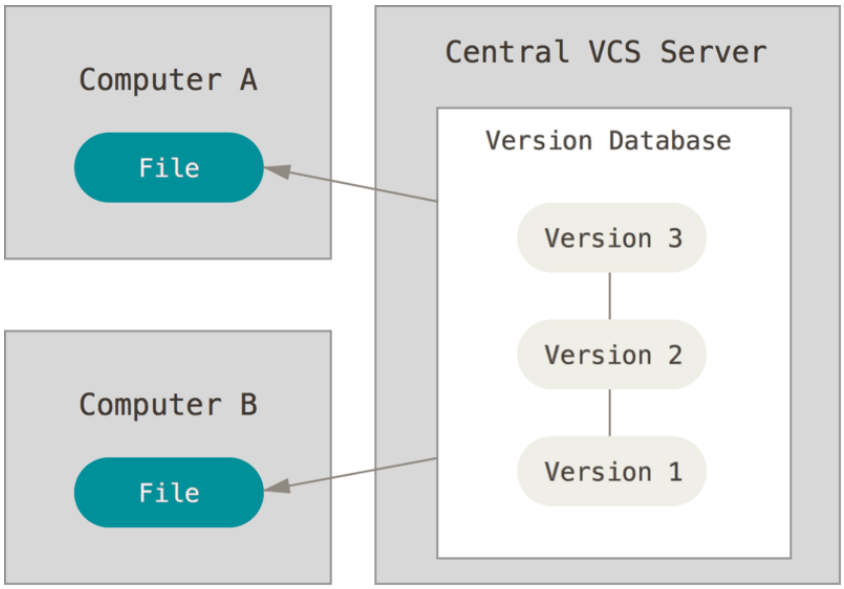

FIGURE 16. Centralized Version Control System (32)

With the high risks of losing everything in a CVCS project, Distributed Version Control Systems (DVCSs) have been introduced to give a solution. Currently popular VCSs, such as Git and Mercurial, fall under this VCS. Instead of relying on the latest snapshot of the files, DVCSs fully mirror the repository, including its full history. Therefore, if a server dies, the client repositories can be copied back up to the server to reinstate it. Each copy is a full backup of all the data. Furthermore, these systems deal with having several remote repositories they can work with, enabling simultaneous work and collaborations with different groups (FIGURE 17) [32].

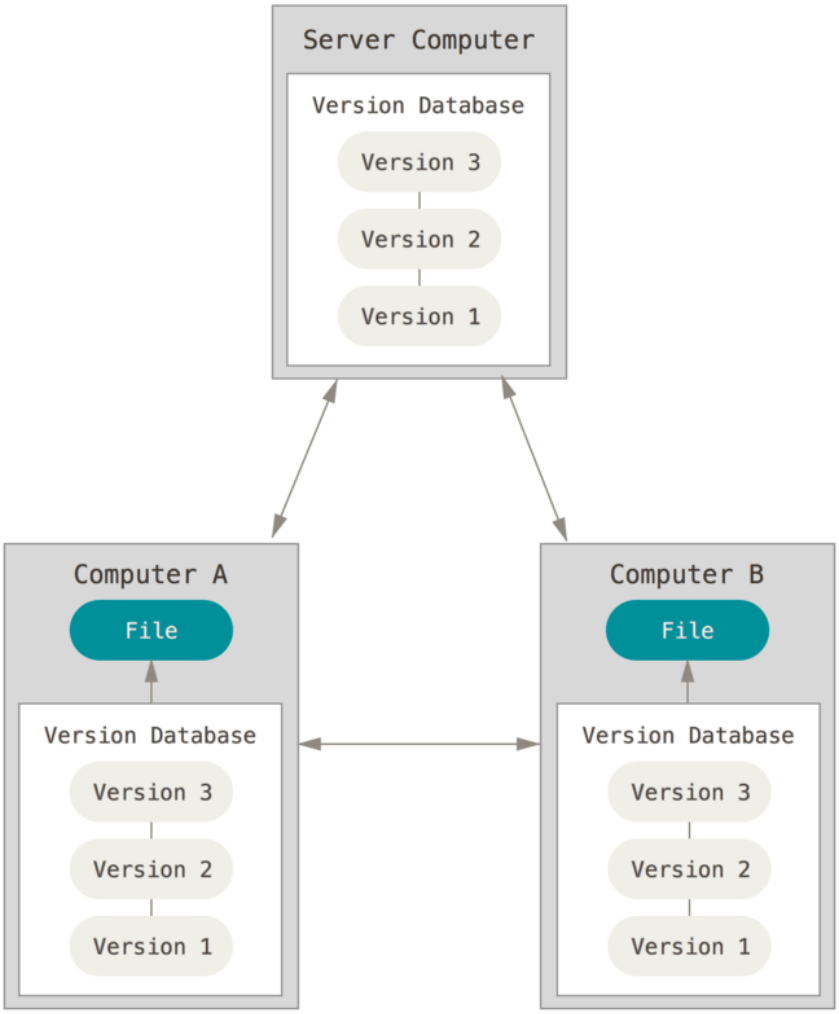

FIGURE 17. Distributed Version Control System, a reflection of currently popular VCSs (32)

5.2.2 Git

Git is “a free and open-source distributed version control system designed to handle everything from small to very large projects with speed and efficiency”, stated by their website. Git is considered lightweight and easy to learn, and it outclasses other VCSs, such as Subversion and Perforce, with local branching, convenient staging areas and multiple workflows [33]. Git is also the version control used in this thesis work.

While the user interface of Git corresponds to other VCSs, Git stores and examines information in a different way. Most other systems store information as a list of file-based changes; a set of files and the changes made to each file over time. Git, however, considers its data as a series of

snapshots of a miniature filesystem (FIGURE 18). Every time a user commits, or saves the state of their project, Git takes a picture of what all the files resemble to at that moment and stores a reference to that snapshot. If the files remain the same, Git links them to the previous identical files it has already stored. A data, to Git, is a stream of snapshots. Git re-examines almost every view and detail of version control that most other systems copied from the previous VCS methods, concluding the ways of working important to distinct. Git also does not require network for most of its operations since they can be run with just local files and resources. This allows developers to have the entire history of their project on their local disk [34].

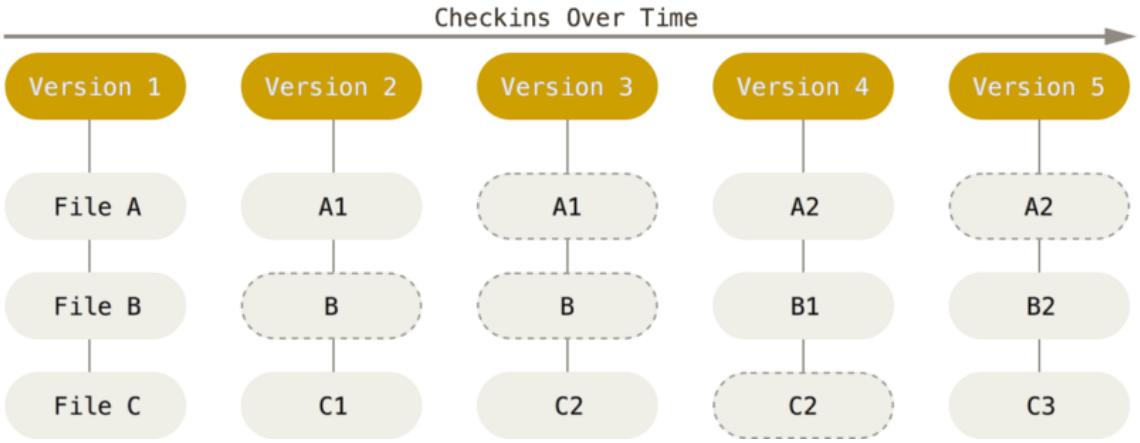

FIGURE 18. Data as snapshots of the project over time (34)

Branching, a way of diverging from the main line of development and continuing work without threatening that main line, is a common process for most VCSs. But for most, this process is expensive, often requiring recreation of a copy that holds the source code directory. The branching of Git is distinct from the rest of VCSs by being lightweight and instant with operations through branching. Switching between branches is also fast. Git also encourages workflows that branch and merge regularly. As a commit is made with *git commit*, a commit object is stored, and it holds a pointer to the snapshot of the staged content. This object includes the author's name and email address, the message typed and pointers to the commit that directly came before this commit. When changes and commits are made again, the next commit stores a pointer to the commit that came before it (FIGURE 19) [35].

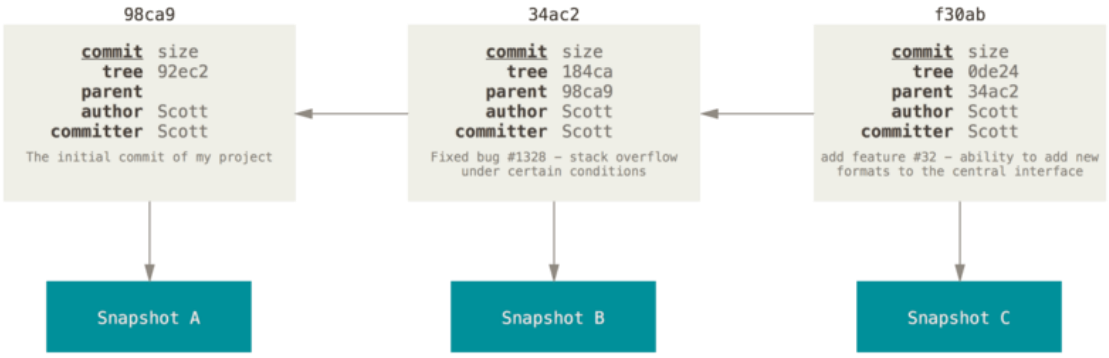

FIGURE 19. Commits with author details, pointers and parents (35)

A branch in Git is a lightweight movable pointer to one of the commits made. The default branch name in Git is *master*. When a commit is made, a master branch is given to point to the last commit that was made. Every time a commit is made, the master branch pointer moves forward automatically. In the FIGURE 20, a special pointer *HEAD* indicates the branch the user is on [35].

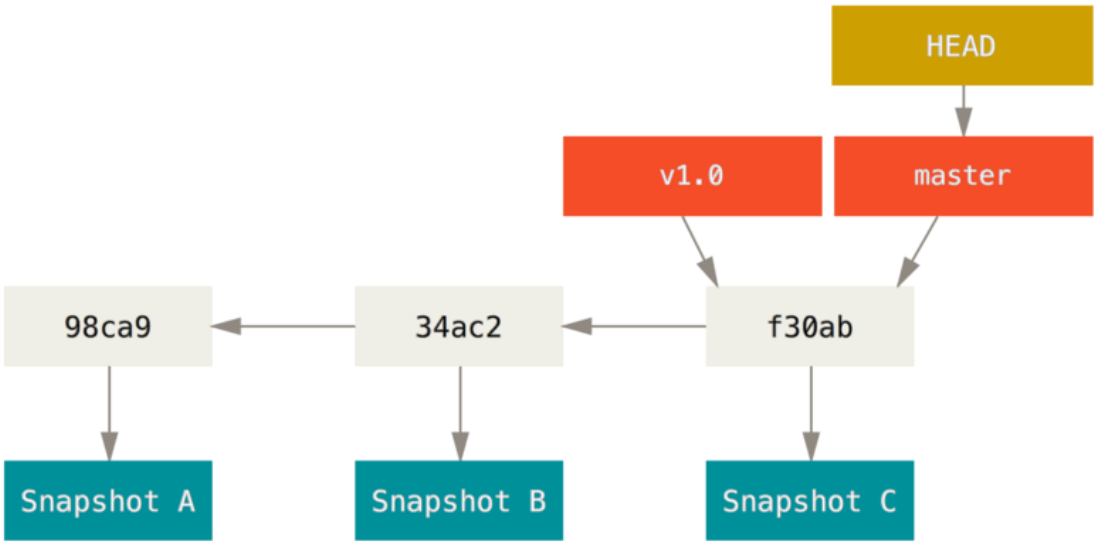

FIGURE 20. A branch and its commit history (35)

FIGURE 21 showcase an example of how branching benefits and works in CI environment. There is a repository named “master”, and each commit is signified with circles. The circle that holds the term master refers to the head of the branch. A branch “develop” is created from one commits owned by master, which allows developers to work at the same time without interfering each other. When changes are ready to be united, a merge request is made; a request to unite two different states of the repository. In this scenario, develop branch is merged into master branch to create a

new state. As part of CI practices, the commit will be reviewed and tested before integration to a working repository such as master. The development continues afterwards, shown with a new "feature" branch.

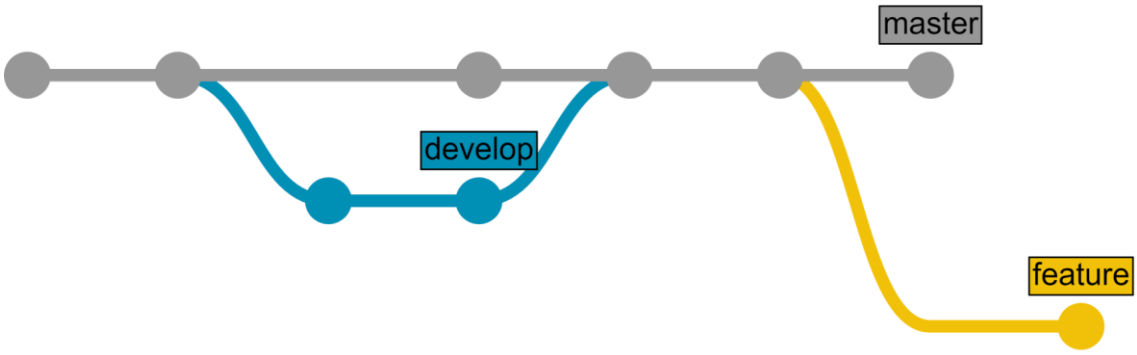

FIGURE 21. Examples of branches in a development environment (36)

5.3 Docker containers

When the discussion of improving a CI/CD based application development process is mentioned, the most modern approach is to use containerization (FIGURE 22) to bring even more flexibility and benefits, such as faster deployment of software releases and cost reduction [37]. Containers also ensure consistency across multiple development and release cycles, standardizing the work environment with the configurations of containers to maintain all configurations and dependencies internally. This leads to the ability of re-using the container from development to production with no discrepancies or manual interventions. Docker containers also ensure that developers do not need an identical production environment set up; they can use their own system to run Docker containers. [38] This also removes the proneness to errors; developers do not have to solve issues such as driver compatibility or library conflicts [37].

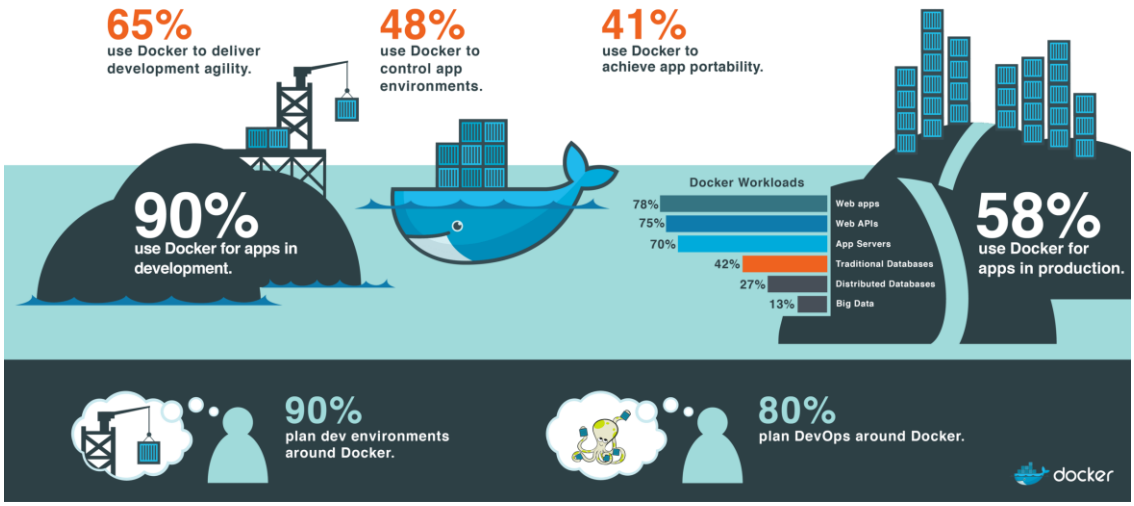

FIGURE 22. A survey made for more than 500 technology professionals, showcasing Docker usage (39)

Containers also overcome enterprise challenges for CI/CD based development processes. New features and fixes do not have to wait for the release cycle, which would commonly take anywhere between 6 months to a year. New major releases can be deployed to production environment without downtime. The gap between development and operations has also been closed with the way containers solve dependency issues. Docker containers also take all the manual processes away from development and deployment. Tools do not have to be restricted due to incompatibility with the existing system because containers create their own, isolated environment. All the benefits lead to faster production and minimized resource wastage [37].

Regarding version control systems, the starting points of CI processes, Docker containers work in a similar way to Git, a version control system discussed in the previous chapter. They allow users to commit changes to Docker images and version control them. This permits rollbacking to a previous version of a Docker image, much as going back to the previous commit in Git. Resembling standard deployment and integration processes, Docker allows developers to build, test and release images that can be deployed across multiple servers. Additionally, launching Docker images is as fast as running a machine process [38].

Jenkins Docker Pipeline

In order to understand how Docker containers can be used inside CI/CD pipelines, Jenkins, “a leading open-source automation server for building, automating and deploying projects” [40], will

be used as an example CI/CD platform tool. Jenkins Pipelines enables defining the whole application lifecycle and to help support CD, contrast to the “freestyle” jobs of Jenkins that support simple CI by defining sequential tasks in an application lifecycle. Pipelines are Jenkins jobs used through the “Pipeline plugin” and built with simple text scripts that use a Pipeline DSL (domain-specific language) formed with the Groovy programming language. They leverage the power of multiple steps to execute both simple and complex tasks according to parameters user has established. Therefore, pipelines can build code and orchestrate the work required to drive applications from commit to delivery (FIGURE 23) [41].

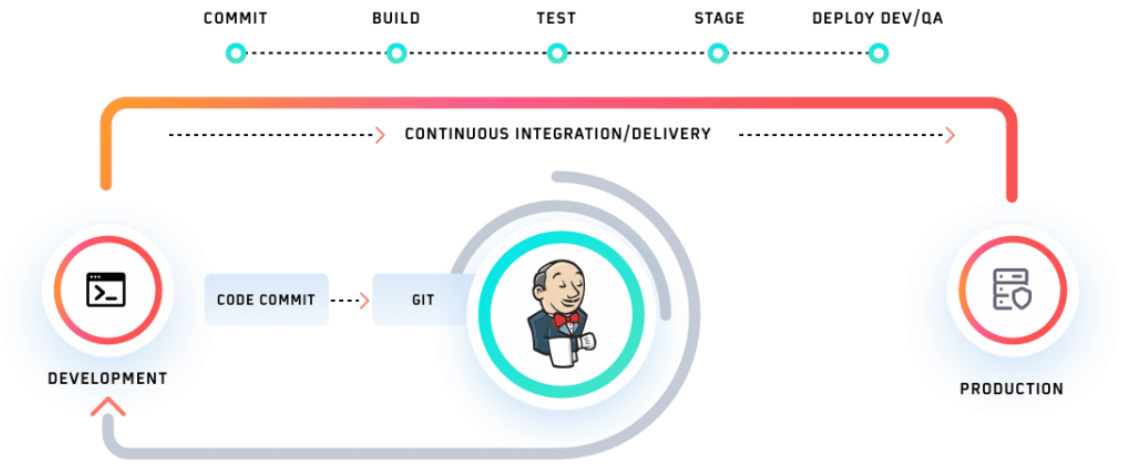

FIGURE 23. A Jenkins pipeline illustration with the logo of the company (42)

Jenkins pipelines have terms used specifically for Jenkins. A *step* indicates a build step, a single task that is part of sequence. Steps tell Jenkins what to do. A *node* is a step that does two tasks with available executors on agents: it schedules the steps contained within it to run by adding them to the Jenkins build queue and creates a workspace, a file directory specific to a particular job, to allocate resource-intensive processes to occur without impacting the pipeline performance. They last for the duration of the tasks assigned to them. A *stage* refers to a step that calls supported APIs. A stage can have one or more build steps inside of it. A *controller* is the basic installation of Jenkins on a computer, and it handles tasks for the build system. It ensures that pipeline scripts are parsed, and steps are wrapped in nodes performed on free executors. An *executor* is “a computational resource for compiling code”. It can run on controller or agent machines. An *agent* is a computer set up to offload projects from the controller. The configurations made by the user determines the number of scope of operations that an agent can perform [41]. The FIGURE 24 and FIGURE 25 showcase how the Jenkins Pipeline terms are used in a Jenkinsfile, a text file that

contains the definition of a Jenkins Pipeline and is pushed into source control. The build stages are visualized by Jenkins user interface.

Jenkinsfile

```
pipeline {
    agent none
    stages {
        stage('Run Tests') {
            parallel {
                stage('Test On Windows') {
                    agent {
                        label "windows"
                    }
                    steps {
                        bat "run-tests.bat"
                    }
                    post {
                        always {
                            junit "**/TEST-*.xml"
                        }
                    }
                }
                stage('Test On Linux') {
                    agent {
                        label "linux"
                    }
                    steps {
                        sh "run-tests.sh"
                    }
                    post {
                        always {
                            junit "**/TEST-*.xml"
                        }
                    }
                }
            }
        }
    }
}
```

FIGURE 24. An example of Jenkins pipeline inside Jenkinsfile (43)

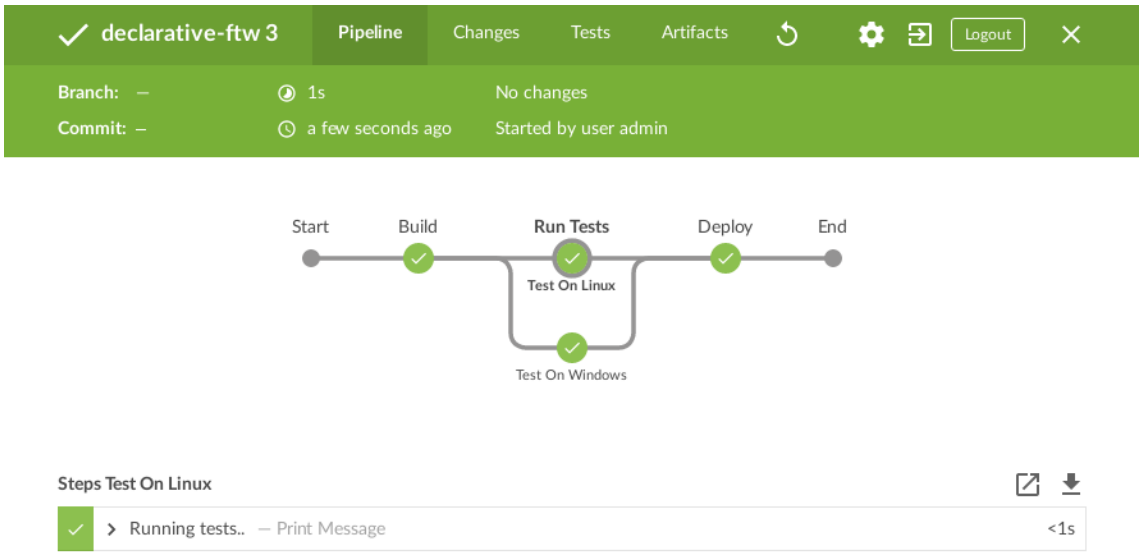

FIGURE 25. Visualized Declarative pipeline with parallel stages (43)

Many companies use Docker to integrate their build and test environments over machines, and to provide a technique for deploying applications. The Docker instruction site of Jenkins states that "Starting with versions 2.5 and higher, Jenkins Pipeline has built-in support for interacting with Docker from within a Jenkinsfile". Pipeline is designed to utilize Docker images (FIGURE 26) as the work environment for a stage or the entire Pipeline, indicating that a user can set the tools required for their Pipeline without manual configurations of agents. If the specified tools can be packaged in a Docker container, they can be used by editing a Jenkinsfile. When the Pipeline executes, Jenkins will automatically start the specified container and execute the defined steps within it [44].

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent {
        docker { image 'node:14-alpine' }
    }
    stages {
        stage('Test') {
            steps {
                sh 'node --version'
            }
        }
    }
}
```

FIGURE 26. A Jenkinsfile with a Docker section for Docker image (44)

Users can also pass custom arguments to Docker through Pipeline, allowing users to define custom Docker volumes to mount to caching data on the agent between Pipeline runs. This will avoid the need to re-download dependencies for subsequent runs of the Pipeline. For projects that need a more personalized execution environment, Pipeline also supports building and running a container from a Dockerfile if it is in the source repository. Using the *agent { dockerfile true }* syntax will build a new image from a Dockerfile (FIGURE 27) rather than pulling one from Docker hub. With this Dockerfile being committed to the source repository, the Jenkinsfile can be changed to build a container based on this Dockerfile and then run the defined steps using that container. It is also possible to define multiple Docker images for Jenkinsfile to run multiple different containers [44].

Jenkinsfile (Declarative Pipeline)

```
pipeline {
    agent { dockerfile true }
    stages {
        stage('Test') {
            steps {
                sh 'node --version'
                sh 'svn --version'
            }
        }
    }
}
```

FIGURE 27. Jenkinsfile syntax with a custom Dockerfile (44)

6 ROOTLESS CONTAINERS

Rootless containers are containers that allow an “unprivileged user” to create, run and manage containers without excluding container tooling. An “unprivileged user” refers to a user who does not have any administrative rights; they do not have the ability to ask for more privileges to be granted to them, which includes software package installations. This restriction of user privileges can mitigate potential container-breakout vulnerabilities and enables them to be more friendly for shared machines. The main downside of rootless containers is, however, their complexity [45]. For this reason, it is important to understand the terms involved with rootless containers, their definitions and restrictions, especially when applied to an existing containerized system that relies on root access. By default, Docker daemon has always needed to be started by the root user to perform its functionalities, such as Linux namespaces that enable isolation of containers. These functionalities are integrated to the features of the Linux kernel and require privileged capabilities [46]. Linux kernel is “the main component of a Linux operating system and is the core interface between the hardware of a computer and its processes” [47].

6.1 Linux root privileges

In order to understand rootless, as it expresses the lack of root, it is important to understand what root means in Linux. Root, also known as “Superuser”, is an exceptional directory represented by a forward slash (/). Root has access to all files and commands in any operating system that resembles Unix. Root (/) is the starting point for the entire directory, whose hierarchy is laid out to resemble a tree, that all other directories branch out from. Those that can access root have special privileges that enable them to modify the system at will. These privileges also allow other users to be granted with permissions. If root privileges and their usage are not monitored, a complete system failure or at the very least, a corrupt system becomes a possibility. [48]. FIGURE 28 showcases that root is the system itself, which is why root should only be used to enable other superuser privileges on certain accounts.

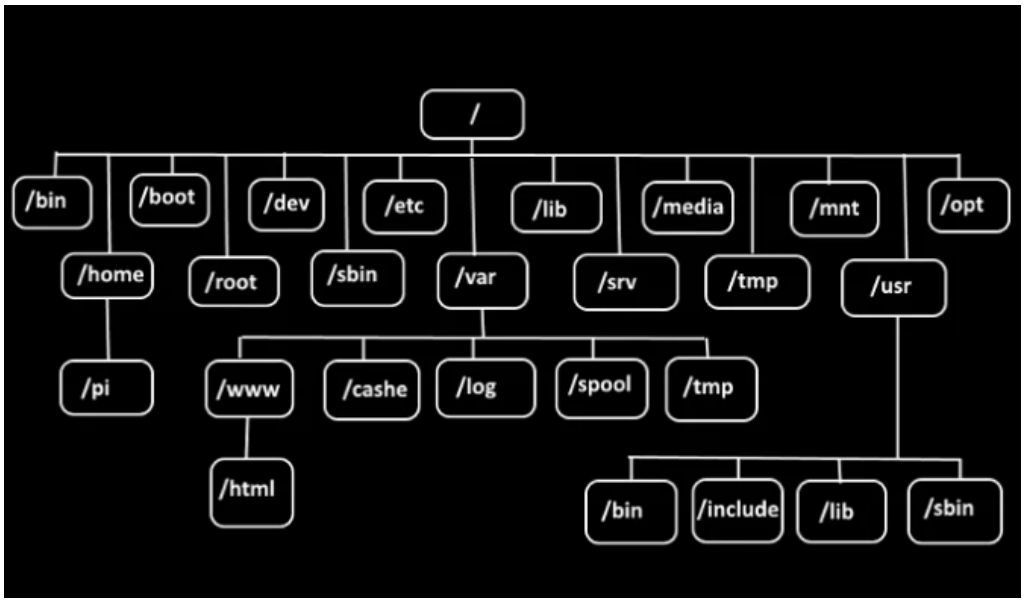

FIGURE 28. Root directory shown at the top of the file hierarchy of a Unix-system (48)

In order to avoid using root to modify anything at an account level, since root should be reserved for system changes, other accounts can be given minimal privileges by using the *visudo* command to edit */etc/sudoers* file (FIGURE 29). *Sudoers* is the file that controls which accounts have access to the *sudo,* a privileged command. Visudo command allows users to edit this file and save the changes, and then locking it to prevent anyone else from changing its content. These accounts with given privileges would not have to access root to perform user management tasks, such as adding new users or modifying their accounts. There will be tasks that require superuser permissions that only the root user can provide; if the additional permissions are needed, editing sudoers file suffices. Otherwise, no other account should be allowed access to root besides root user. Permissions should only be added based on needs of the users and groups to access what is necessary and available with sudo command [48].

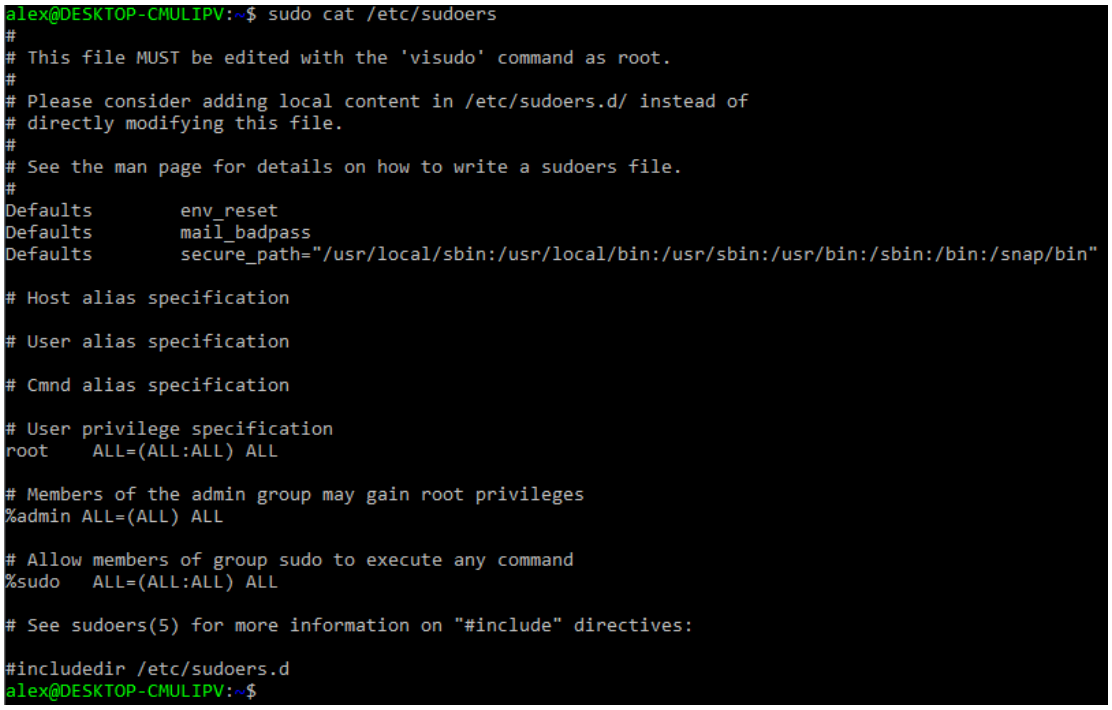

FIGURE 29. A Linux terminal showing the content of /etc/sudoers with sudo cat command

Sudo, an acronym for “Super User DO”, is a command that allows users to perform privileged tasks without the need to be logged in as root or using the *Su* (switch user) command. Adding this prefix with any Linux command runs that specified command with elevated privileges, which are required to perform administrative tasks. In Linux, working around without permissions can enable its use to be much more difficult; saving files, compiling code, running an installed program or even rebooting the system gives an unprivileged user an “Access Denied” prompt that hinders working in a Linux environment. All of this can be avoided by requesting permission with sudo as a prefix to the commands [48].

The main reason why sudo is preferred over root is safety. Sudo grants safety over the system because a user has a greater chance of executing fatal changes to the system when in root, putting the whole system under a risk of collapsing. Sudo is also a better alternative to the switch user (Su) command because *Su* will request the root password and provide a superuser prompt in the form of “#” (FIGURE 30). If this is seen in the directory, it indicates that the user in the system is currently inside a root directory. This allows the execution of all elevated commands and if the user forgets the existence of root directory and continues inserting commands, altering the system without knowledge becomes a possibility. Even just a typo can be deemed to have fatal results [48].

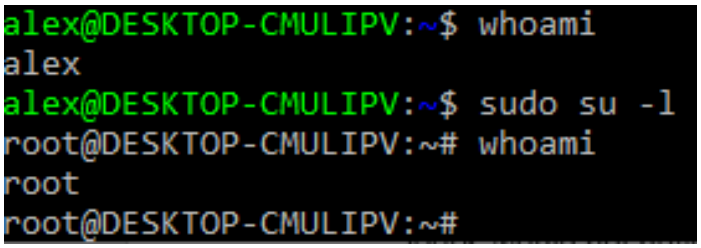

FIGURE 30. A Linux terminal showing how sudo privileged user can switch to root with sudo su command

6.2 Docker container security

Docker supports adding and removing capabilities of containers, allowing users to create a non-default profile. Removing capabilities can enable Docker to be more secure, while addition of capabilities does the reverse. The best practice for users would be to remove all capabilities except those that are required for their processes. By default, Docker starts containers with a restricted set of capabilities; while Docker daemon runs as root, in most cases, containers do not need "real" root privileges, enabling containers to run with a reduced capability set; this is proven by the way Docker organizes tasks that usually require root from servers to be handled by the infrastructure around the container. This indicates that the root inside the container has much less privileges than the root of the host. Restriction and removal of capabilities can involve denying all mounting operations, denying access to raw sockets and file system operations, such as changing the owner of files, and deny module loading. This indicates that even if an attacker manages to gain access to root within a container, it is difficult to do serious damage to the host. The primary risk of Docker containers, however, is that the default set of capabilities and mounts given to a container may provide incomplete isolation, either independently or when used with kernel vulnerabilities [49].

There are four major areas of concern regarding the security of Docker containers: the intrinsic security of kernel and its support for namespaces and control groups (cgroups), the attack surface of the Docker daemon, loopholes in the container configuration profile and the security features of the kernel and how they interact with containers [49]. Developers should also be aware of "poisoned images" that will be discussed in section 6.2.3.

6.2.1 Linux kernel

Docker containers are very similar to LXC containers (Linux Containers) and they have similar security features because of Linux kernel, the main component of Linux operating system. As a container is started with *docker run*, Docker produces a set of namespaces and control groups for that container [49]. Namespaced resources are mapped to a separate value on the host that is running containers; for example, PID (Process identification number) 1 inside a container is not PID 1 on the host or in any other container. Resources that are not namespaced are the same on the host and in containers [50]. Namespaces are in use because they provide the isolation feature of containers, which ensures that processes running in a container cannot be seen or affected by processes running in another container or the host systems. Kernel namespaces were introduced in Linux 2005 and since July 2008, namespace code has been exercised and scrutinized on many production systems. Since the code has been around for an extensive timeframe, the design and implementation of namespaces are mature and give a more secure base for namespaces that are in use [49].

Container breakouts, however, can occur because the users are not namespaced by default in Docker containers. The devices, the kernel itself, kernel keyring and kernel modules are also not namespaced, bringing their own vulnerabilities to the container run. For example, if a container loads a kernel module, a privileged act, the module will be available across all containers and the host because the kernel is shared among them. A compromised kernel can take down the whole host, while compared to virtual machines, an attacker would have to route an attack through both the virtual machine kernel and the hypervisor first before coming across the host kernel. When the users are not namespaced, any process that escapes the container will have the same privileges on the host as it did in the container. For instance, a user with root privileges inside the container will have the root privileges in the host as well. Besides the concern of container breakouts, there is potential for privilege escalation attacks; a user gaining elevated privileges, such as the root privileges. While container breakouts are unlikely, container security should be organized around the assumption that it can occur [50]. Root access inside containers through Docker daemon will be discussed in section 6.2.2.

As network is also one of resources of Linux kernel, [50], network security will also be discussed. By default, each container receives its own network stack. This indicates that a container does not gain privileged access to the sockets or interfaces of another container. If the host system is setup

properly, containers are able to interact with each other through their network interfaces. If the containers have been specified to use public ports, network operations and IP traffic, such as establishing TCP (Transmission Control Protocol) connections or sending and receiving UPD (User Datagram Protocol) packets, can be restricted if necessary. From a network architecture point of view, all containers resemble physical machines connected through a common Ethernet switch [49], which only connects devices on a computer network.

Control groups (cgroups) are another crucial part of a Linux container. The code for control groups was introduced in 2006, and initially merged in kernel version 2.6.24. They implement resource accounting and limiting and provide useful metrics, while ensuring that each container receives its fair share of memory, CPU and disk input and output. With cgroups, a single container cannot bring the system down by exhausting the resources listed. While they do not prevent containers from accessing or affecting the data and processes of other containers, control groups are essential to fend off some denial-of-service attacks [49]. As all containers share kernel resources, if one container was able to monopolize access to certain resources, including memory and more esoteric resources such as user IDs (known as UIDs, unique identifiers), it can starve out other containers on the host; this is called a denial-of-service attack, leading to users unable to access part or all the system they need [50].

6.2.2 Docker daemon

In order to run Docker containers, the Docker daemon must be running as well. The daemon requires root privileges unless rootless mode is utilized for every container run. A common rule is to grand control of Docker daemon for trusted users only because of the capabilities Docker features have with root access. For example, a default Docker container run allows users to share a directory between the Docker host and a guest container without limiting the access rights of the container. This indicates the possibility of starting a container where the */host* directory is the root directory (*/*) on the host. The container will also be able to alter the host filesystem without any restrictions. This resembles virtual machines that can have filesystem resource sharing, which means that root filesystem can be shared with a virtual machine too [49].

To demonstrate the risk of running a container as root, an example from Marc Campbell's blog post [51] will be used as the base. In FIGURE 31, *docker run -i -t ubuntu /bin/bash* command runs a container based on an *ubuntu* image that is shown to be pulled for the first time, in interactive shell mode that allows input and output of terminal, with a bash session. The container run is launched by a regular Linux user with no extensive privileges of their own. As there are no configurations performed to the user that will be inside the container, the container is running as root because of the root privileges of Docker daemon, showcased by the word "root@". A file named *secrets.txt* is created in the */root* directory, preventing anyone other than root from viewing it.

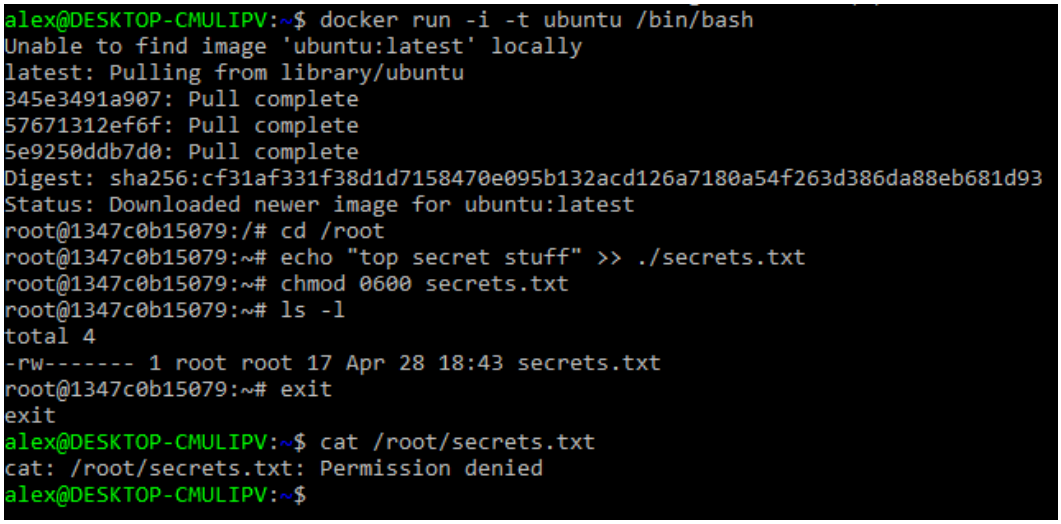

FIGURE 31. Creating a file for root, inside a container

As the container has been exited, a file named */root/secrets.txt* now exists in the root directory. The non-root user attempts to access the file that was created inside the container only to receive a "permission denied" prompt because the file has been specified to only be accessible by root. This example indicates that if root can be modified from the container, the container can also have the whole root filesystem of the host machine mounted inside of it. Vlatka Pavišić's blog post [52] will be used to showcase this instance. Pavišić's example of root and host mounting is also showcased in "Docker Security" report by Adrian Mouat [50].

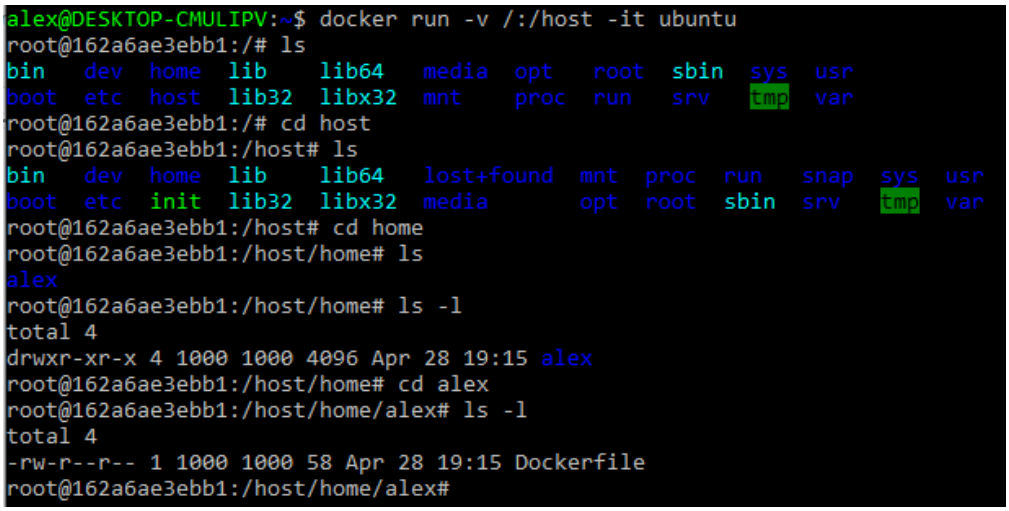

FIGURE 32. A container accessing what the host machine of the user has inside their home directory

FIGURE 32 showcases how root (*/*) filesystem can be mounted to */host* directory inside the container with *-v* mounting flag. As *docker run -v /:/host -it ubuntu* is performed with the same ubuntu image, *cd* command moves between directories, and in this case, inside the regular user's home directory. *ls -l* command shows what the user has inside their home folder. The whole host file system is now in the host directory made for the container.

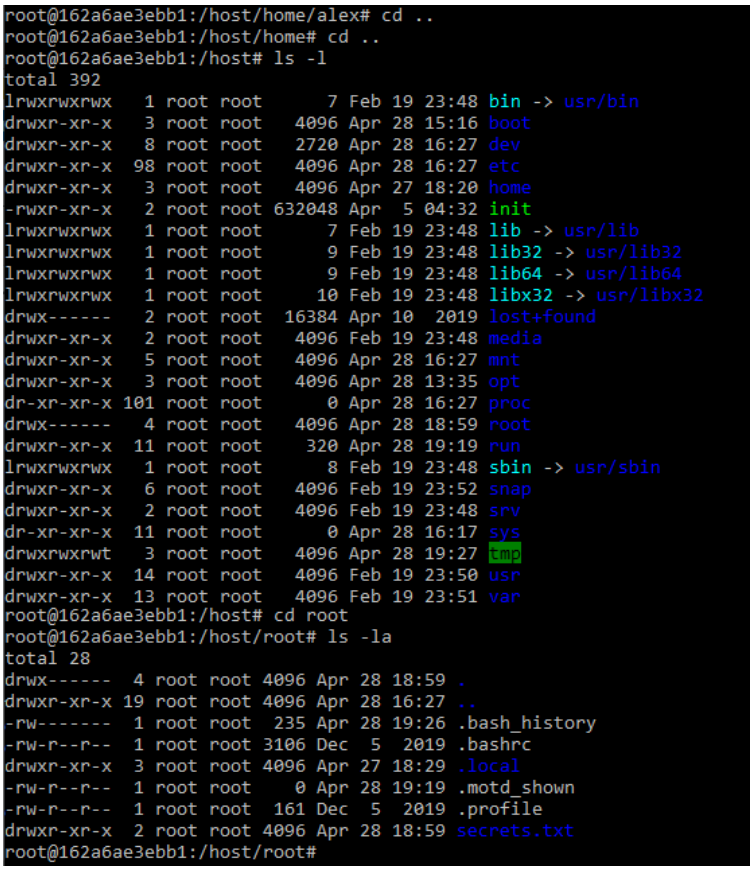

FIGURE 33. *A container inside root directory, with /root/secret file created from a previous container run*

In FIGURE 33, */host* is shown to have root as one of its directories. As the root inside the container is the same root as the root of the host, *ls -la* command can list all the files and directories inside of it. The *secrects.txt* file that was created in the previous steps to */root* directory is shown to exist as well. If the container wants to hold the most power, running a command *echo "(username) ALL=(ALL) NOPASSWD: ALL" > /host/etc/sudoers.d/(username)* grants the regular user root privileges without being root. Editing *sudoers* file with the command mentioned, allows the user to run *sudo* without password prompts for administrative commands that would normally need it, after exiting the container [52].

A common method of enabling non-root users to run Docker commands with the root privileges required by Docker daemon, is to create a Unix group called *docker* and add users to it. When the Docker daemon starts, a Unix socket is made available for the members of the docker group. As the Unix socket is owned by the root user, sudo privileges a requirement for other users to gain access to it. [53]. Because root is, even for non-root users, used to run containers, those containers have access to everything root has access to in the server. This would lead to every container

pulled from Docker Hub to have full access to everything [51]. Additionally, if a container accesses a database or service, it will likely require a secret, such as an API key or username and password. An attacker that has access to that secret, has access to the service [50].

A way to combat the security concern of root access inside a container, is to create a regular user for the container. This can be implemented in a numerous of ways, from Dockerfile to a helper script in ENTRYPOINT after a *--user* parameter has been specified in the docker command. This would ensure that the user inside the container would not have root privileges on the host [51]. The user switch implementation will be covered in section 6.4.

6.2.3 Docker images

If an attacker can trick users to run tampered images pulled from a public registry, such as Docker hub, both the host and the data are at risk. Therefore, it is important to examine that the images are safe, untampered and come from a known source, and to ensure that the images running are up-to-date and do not contain versions of software with known vulnerabilities. Images that need updating, however, are not as easy to identify. Both the base images and dependent images must be updated and fetched in order to create an updated version of them. The new versions must be pushed to the registry that holds them. Containers must also be restarted, and the new images pulled with *docker pull* to ensure they are up to date. Old images should be removed from the host and the registry. Patching a vulnerability found in a third-party image, including the official images, is a task dependent on that party providing a timely update. If the Dockerfile that was used to build the image is accessible with its sources, creating a new image is a simple and effective temporary solution if updates are not put in place by the third-party. As the main safety rule, knowing where the images came from and who created them is important, to ensure that tampered, corrupted, or malicious images are not used in development work environments. If such an instance occurred, a malicious image can have full access to the host [50]. Docker can also be configured to only run signed images, which have been given the trust signature verification [49].

6.3 Rootless container structure

As new solutions and ideas of the ways containers are implemented are being introduced, rootless containers have gained fraction with the popularity of containerized development environments.

Rootless containers are defined as a concept of containers that do not require root privileges in order to operate. They were introduced to overcome the technological challenges of creating a container with an unprivileged user. However, they are still in their early stages of development and adoption while being supported by major names in the field [54].

Rootless containers are known for adding a new security layer to containers. If a container engine, runtime or orchestrator is compromised, the attacker will not gain root privileges on the host. Rootless containers also allow multiple unprivileged users to run containers on the same machine, and there is an added isolation inside nested containers. A new development in the Linux kernel allows unprivileged users to create new user namespaces, enabling isolation processes for rootless containers [54]. User namespaces isolate security-related identifiers and attributes, especially, UIDs and GIDs. With this isolation, the UID and GID of a process can be different inside and outside a user namespace. This enables the utilization of an unprivileged UID outside a user namespace while at the same time possessing a UID of 0 inside the namespace, 0 representing the UID of root. The process has full privileges for operations inside the user namespace but is unprivileged for operations outside the namespace [55]. Namespace root is not as privileged as real root in areas that affect the entire system. As an example, a namespace root cannot load or delete kernel modules [54].

As Docker needs privileged capabilities to utilize Linux features, rootless mode must work around them. The key solution comes from user namespaces. User namespaces map a range of UIDs so that the root user in the inner namespace maps to an unprivileged range in the original namespace. A new process in user namespace also takes up a full set of process capabilities. The user namespaces feature has been integrated into Docker with the *–userns-remap* flag that maps the users inside containers to a different range in the host, providing an enhanced security for cases where the container has access to the same resources outside of it. The rootless mode, however, functions by creating a user namespace first, allowing Docker daemon to start with the remapped namespace instead of the root of the host. The daemon and the containers will both use the same user namespace that is different from the host one (FIGURE 34) [46].

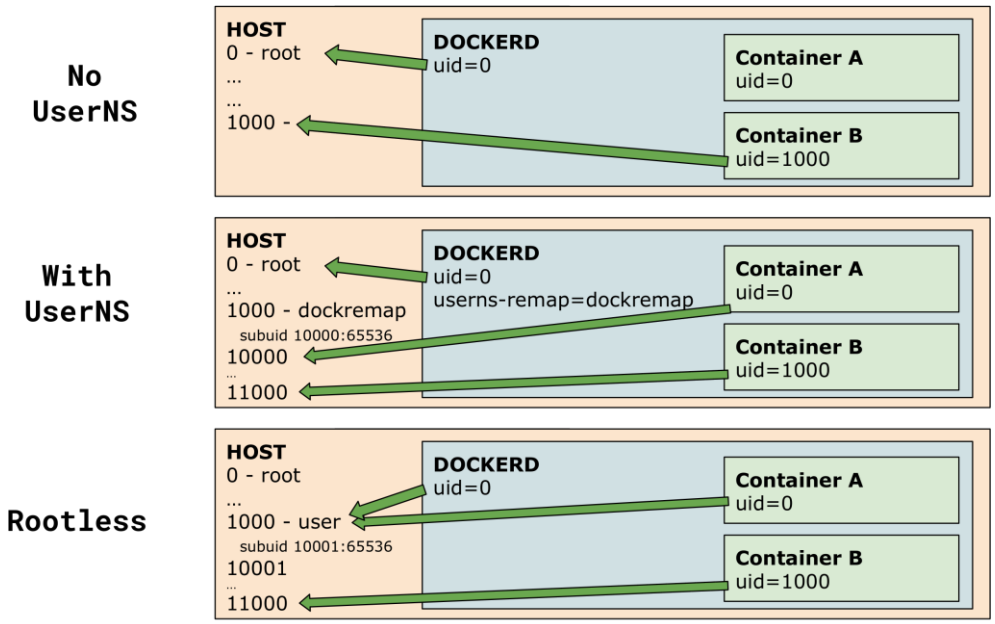

FIGURE 34. User namespace remapping (46)

It must be noted that while Linux allows creating user namespaces without root privileges, these namespaces only map a single user and therefore do not work with many current existing containers. As a solution, rootless mode has a dependency on the *uidmap* package that can do the remapping of users without the need of manual work [46]. The binaries in *uidmap* package use *setuid (set user identity)* bit to set the effective user ID of the calling process [56], or file capabilities, and therefor always run as root internally [46]. Networking inside a container, however, has shown to be a challenge for rootless containers [54].

6.3.1 Networking

Usually, a Virtual Ethernet device (VETH) is created and in charge of the networking to allow proper networking inside the containers. As only real root has the privileges to create such devices, a rootless container will not be able to do so. In order to overcome this issue, rootless containers turn to other ways of setting up a network. *Slirp* (slirp4netns), *LXC-User-Nic* and *RootlessKit* projects are mentioned regarding the solutions of rootless container networking. Rootless Docker containers rely on networking based on the *moby/vpnkit* project of Docker, the same project that is used for Docker Desktop products. The rootless script of Docker and Podman use RootlessKit for their rootless setups, such as namespace remapping and networking, while Slirp and LXC-User-Nic are

more commonly known for creating networking for rootless containers [54, 46, 57]. RootlessKit will be discussed further in section 6.6.

Slirp was originally designed to be an internet dial-up for unprivileged users and in time it became a network stack for virtual machines and emulators. Later, it was adjusted to enable networking in rootless containers. It works by forking into the user of the container and networking namespaces and creating a tap device that becomes the default route. It then passes the file descriptor of the device to the parent who runs in the default network namespace, which can now communicate with the container and the internet. LXC-User-Nic, however, sets up networking by running a setuid binary that creates a VETH device. While it enables networking inside the container, it misunderstands the goal of rootless containers because it needs the container binary to run with root privileges [54].

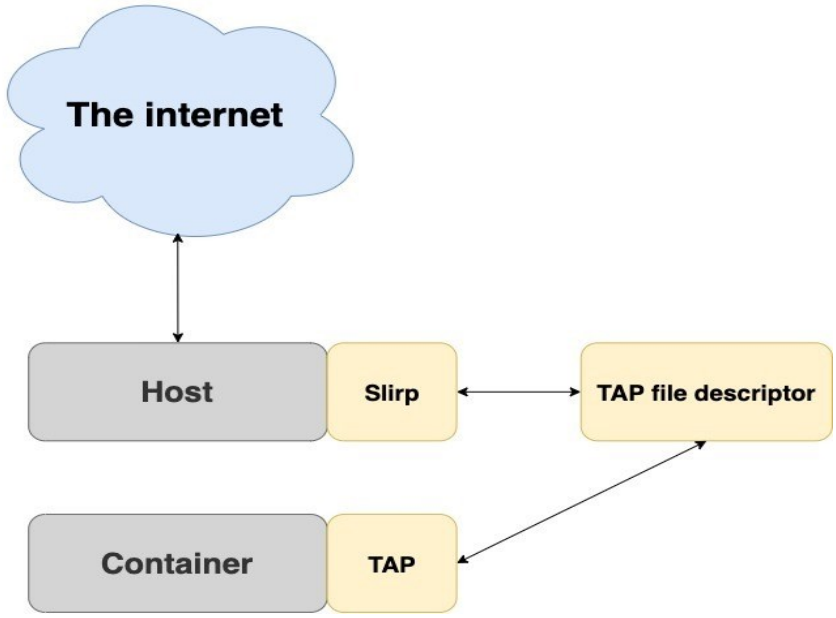

FIGURE 35. Slirp networking (54)

6.3.2 Control groups and storage

Another key element of containers, the Linux control groups (cgroups), is also dependant on root privileges by default. Since the cgroups interface is provided through a pseudo-filesystem that usually resides in *"/sys"*, a root owned directory, a non-root user cannot access and utilize it. As this feature to control and limit resources is considered a necessary part of containers, two approaches have been introduced for rootless containers: cgroups V2 and PAM (Pluggable

Authentication Module). Cgroups V2 is a new kernel implementation of cgroups that supports delegating permissions to unprivileged users. The downside is that V2 does not support all the controllers that were implemented for cgroups V1. PAM module, a solution provided by LXC, works after installing it through *pam_cgfs.so* to allows unprivileged users to authenticate and manage cgroups [54].

For storage management, container engines by default use a special driver called *Overlay2* or *Overlay* to create a layered filesystem that is efficient in both space and performance. This, however, cannot be executed with rootless containers because most Linux distributions do not allow mounting overlay filesystems in user namespaces. Therefore, rootless containers have had to rely on other drivers and filesystems. One of the solutions have been to use another driver, such as the VFS (Virtual File System) storage driver. Despite it being compatible with rootless containers, it is considered less efficient than the default drivers. Another solution is to create a new storage driver to suit the needs of rootless containers, such as a driver called *FUSE-OverlayFS*. This driver is a user-space implementation of Overlay; more efficient than VFS and capable of running inside user namespaces [54].

6.3.3 Non-rootless methods

Regarding rootless containers, the definitions can vary often. Usually, the term refers to a user running the entire container runtime as well as the containers without the root privileges. With this definition, even if containers are running as non-root users but the runtime is running as the root, this indicates that the container is not actually rootless. Rootless containers, however, can fall under *setuid* and/or *setcap* binaries to configure *newuidmap*. If the entire runtime is running with setuid, also known as setting a user ID on execution, they are not typically called rootless containers. *Docker rootless-mode*, *Podman rootless-mode*, *BuildKit rootless-mode*, *LXC unprivileged-mode* and *Singularity fakeroot-mode* are considered common rootless containers [45].

Methods, such as allowing a non-root user to access Docker socket by adding the user to docker group, does not count as a method of implementing rootless containers because the runtime is started as root, and the container opens as root. Command flags that change the user inside the container, such as *--user* switch and *--userns-remap* for Docker, are not considered to fall under the scope of rootless containers [45]. They can, however, be considered “half-rootless” options

because of the additional security layer they bring for the Docker container run; the user inside the docker will not be root, despite the container run starting as root. This statement is based on the content of section 6.2.2.

6.4 Docker user switch implementation

After the test environments had been setup according to the goals and requirements of section 2, the implementation part of the thesis began by fetching the work environment of the company from their Git repository to gain access to the scripts that were involved with Docker containers. Enabling Docker container to be running as a non-root user was deemed to be a more accessible option to start the project with, as it did not require installations of foreign platforms and programs for a rootless container, and it lacked in complexity. Therefore, the installable, experimental rootless-mode script [58] provided by Docker, was not used either.

The command that starts a container run, *docker run*, had been integrated into a Python 3 file, working as the starter script of the Docker container with its configurations. The first step was to edit this file to have a *--user* parameter for *docker run* command in order to override the *USER* instruction of the Dockerfile, as it sets the given user's name or UID to use when running the image of the container and for any *RUN*, *CMD* and *ENTRYPOINT* instructions that follow it in the Dockerfile [59]. This modification is made because root (UID = 0) is the default user within a container even when Dockerfile does not include USER instruction in its file. Additionally, when passing a numeric ID (identifier) to the user switch, the user does not have to exist in the container. The *--user* parameter can accept user names, user group names, UIDs and GIDs combined as values [60]. In this implementation, the *--user* parameter (FIGURE 36) takes the UID and the GID values of the current user through *os.getuid()* and *os.getgid()* functions [61], with *whoami* variable holding the UID value (FIGURE 37). The *getpass* module of Python uses *getuser()* method to fetch the login name of the user that is running in the current process [62]. This name is used to create a sanity check, to ensure the container is not started or running as root. This check is also executed for UID. If either value indicates the user is root, the container run will not be started.

```
'--user', f'{whoami}:{os.getgid()}',
```

FIGURE 36. The –user switch added to docker run command inside a Python script with UID and GID values

```
    # Check that user is not root
    whoami = os.getuid()
    assert whoami != 0, 'do not start as root'
    myname = getpass.getuser()
    assert myname != 'root', 'do not start as root'
```

FIGURE 37. A part of the script setting up the UID and the login name of the user and sanity checks ensuring root no access for a container run

The next step was to add a user inside the container, with the values given by the *--user* parameter. As ENTRYPOINT instruction of Dockerfile holds the helper script *entrypoint.sh* (FIGURE 38) that runs inside the container after it has been set to run, it also configures the user's work environment inside the container, such as work directory paths. The helper script used in this ENTRYPOINT is a bash script.

```
COPY VERSION /
ENTRYPOINT ["/scripts/entrypoint.sh"]
```

FIGURE 38. A helper script provided for ENTRYPOINT instruction to run more complex environment setup processes inside the container

In FIGURE 39, environment variables USERID and GROUPID are assigned with the UID and GID of the values provided by the user switch. Both FIGURE 39 and FIGURE 40 also showcase additional sanity checks, with FIGURE 40 being called later in the ENTRYPOINT script because *whoami* variable is only known after the user has been created. The steps of creating a user inside the container are showcased in FIGURE 41.

```
USERID=$(id -u)
GROUPID=$(id -g)

# Check that current user is not root
if (( $USERID == 0 )); then
    echo "Failing as root user was used to launch start.py"
    exit 1
fi
```

FIGURE 39. Setting up the USERID and GROUPID variables and ensuring the UID does not belong to root inside the ENTRYPOINT script

```
# Check that current user is not root
if [[ "$(whoami)" == "root" ]]; then
    echo "Prevent root user from launching the container"
    exit 1
fi
```

FIGURE 40. Preventing root from running inside the container

In FIGURE 41, the environmental value USERNAME is set to hold the same value as USER that was given by the starter script of the container, both indicating the name of the login user. The GROUPNAME is also used to hold the USERNAME value, as the name of the group can be assigned to be anything preferred. HOME variable receives a path to the user's home directory as it otherwise remains empty without proper setup. The command *groupadd -g* has GROUPID and GROUPNAME as values, creating a group name based on the USERNAME and the GID set in FIGURE 39. The command *useradd -ms /bin/bash* creates a user with the given parameters: *-d* flag sets the path of the home directory, *-u* flag sets the user ID, *-g* flag sets the group ID, and the USER variable finalizes the command with the given user name it holds. The flag *-ms* after *useradd* specifies that the command is running with specific options: *-s* with */bin/bash* allows the specification of the new user's login shell and *-m* flag creates the home directory of the user [63].

```
USERNAME=$USER
GROUPNAME=$USERNAME

# Setting home environment variable
HOME=/home/$USERNAME

groupadd -g $GROUPID $GROUPNAME
useradd -ms /bin/bash -d $HOME -u $USERID -g $GROUPID $USER
```

FIGURE 41. Creating a user inside the container

As creating a user requires privileged commands, the execution must be given rights in the configuration scripts that are used in Dockerfile (FIGURE 42). In this implementation, *default.sh* holds the configurations that are set before the container is opened. Thus, root can give the permissions that have been set in this file before the container is opened as a non-root user.

```
# default system configuration
COPY default.sh entrypoint.sh /scripts/
RUN /scripts/default.sh
```

FIGURE 42. Showcasing the scripts that are copied to Dockerfile and used by RUN instruction

To ensure that a regular user can create another user inside a container, FIGURE 43 introduces *chmod* commands inside the *default.sh* file. A standard subdirectory of the root directory in Linux [64], */sbin/*, holds the commands *useradd* and *groupadd* that are used in FIGURE 41. The command *chmod* changes the additional permissions or special modes of a file or directory. The symbolic mode 's', also known as a *setuid bit,* used in this implementation indicates that this change is only applied to the appropriate classes. As *chmod a+s* uses 'a+' to indicate all, all classes from users to groups can execute the command specified by the executable without requiring root or sudo [65], which is why the helper script of ENTRYPOINT can create a user inside the container.

```
chmod a+s /sbin/useradd
chmod a+s /sbin/groupadd
```

FIGURE 43. Commands allowing all users to create another user inside a container

The first successful run was executed inside Fedora, a Linux distribution running inside a virtual machine. A new image was built based on the changes made and the starter script was configured to start the container with the version of the new image instead of the old one. To ensure that the user inside the container was not root anymore and did not hold any sudo privileges that were not given to them, *sudo -v* command was used in the terminal of the opened container. An empty response from the terminal would have indicated that the user has sudo privileges and is able to run administrative commands without having their password asked. As seen in FIGURE 44, the user, however, receives a negative response, confirming the privileged user has been successfully switched to a non-root inside the container with the modifications specified in this chapter.

```
+ echo 'Running bash in container'
Running bash in container
+ /bin/bash
[alex@l1sw-env-2021.03.16.0931 l1sw]$ git-lfs --version
git-lfs/2.13.2 (GitHub; linux amd64; go 1.15.5)
[alex@l1sw-env-2021.03.16.0931 l1sw]$ sudo -v
Sorry, user alex may not run sudo on l1sw-env.
[alex@l1sw-env-2021.03.16.0931 l1sw]$
```

FIGURE 44. A screenshot of a Fedora terminal, showcasing that the user inside the container does not have sudo privileges

Despite the success of having a non-root user operating inside Docker container, the original environment setup through scripts was executed by root. This indicated that especially the helper script in *ENTRYPOINT* had multiple snippets of code that required sudo or root access to be completed. As the script in ENTRYPOINT was needed to setup the proper work environment inside the container, the user required at least sudo privileges to complete the steps that involved creating working directories. These privileges would have to be granted through the configuration script, *default.sh,* for the non-root user. The CI team was also unsure if the pipeline stages had jobs that required root access with their tasks, as they opened containers to run their tests. These privilege challenges of implementing and merging the user switch to be utilized by the company will be discussed further in the section 7 as part of conclusions.

6.5 Podman

As Podman was introduced in the project plan as one of the potential container engines to provide rootless containers for Docker containers, it will be overviewed briefly in this chapter. Podman resembles container engine Docker, creating and managing pods, containers and container images [66]. Podman and Docker images are compatible and Podman uses the same commands as Docker, which enables Podman to be easy to integrate into an environment that uses Docker [67].

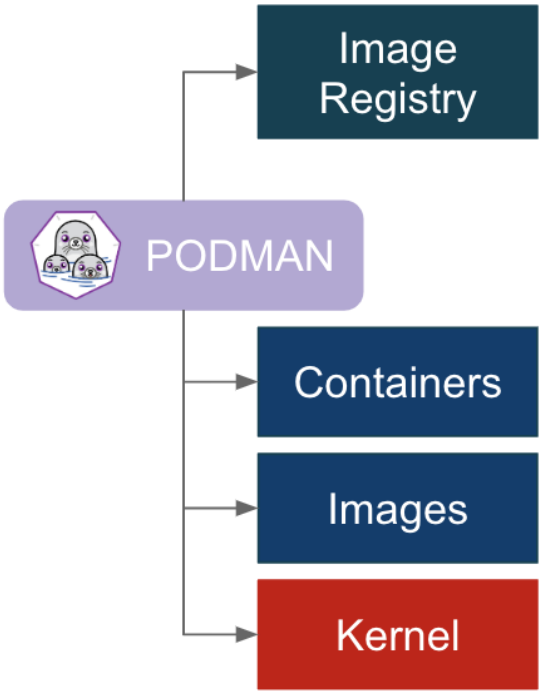

FIGURE 45. A diagram of the architecture of Podman (67)

Compared to the daemon and client approach of Docker, where Docker daemon does all the work related to containers and its operations are conducted by root users, Podman interacts directly with the image registries, containers, image storage and the Linux kernel (FIGURE 45) through the *runC* container runtime process, which is not a daemon [67]. This daemonless approach enables the containers of Podman to be rootless by default. FIGURE 46 showcases that the container run of Podman does not allow root directory to be accessed nor mounted, whereas Docker container, run by a regular user, has the access to root directory by root privileges of Docker daemon, and is able to create files inside of the root directory by becoming root inside the container [68]. This also indicates that the rootless feature of Podman allows almost any container to be run as a regular user, with no elevated privileges. This is considered a major security benefit compared to Docker containers [69].

	User Outside - Root Inside

Docker	$ whoami
	fatherlinux
	
	$ docker run --privileged -itv /:/host ubi8 touch /host/etc/hacker
	
	$ ls -alh /etc/hacker
	-rw-r--r--. 1 root root 0 Oct 23 17:02 /etc/hacker

Podman	$ whoami
	fatherlinux
	
	$ podman run --privileged -itv /:/host ubi8 touch /host/etc/hacker
	touch: cannot touch '/host/etc/hacker': Permission denied

FIGURE 46. The root access difference of Podman and Docker container runs (68)

Podman, however, does have its challenges with pulling images because of the restrictions of the rootless containers. An image can end up not fitting into the default UID mapping of rootless Podman. Rootless containers run inside of a user namespace, which maps the users of the host and groups into the container [69]. This means that Podman allows developers to run it as their own user account. In a container, developers can use a virtual set of users which are mapped to a set of UIDs controlled by the developer's own account for the containerized processes [68]. Regardless of what a user appears to be in a rootless container, the container is still running as the own user of the host and the access to files is limited to what the host can access. In a scenario where an attacker breaks out of a container, they are still constricted to a non-root user. This asserts user mapping to be important to rootless containers but as its indicated, any user required by the container must be mapped in. This can bring errors when pulling images, since a certain image can be tied to a certain UID/GID that was not defined in its user namespace [69]. This is one of the reasons why Podman was, despite being introduced, not used in this thesis project; the images used by the containers of the company did not fit into the namespacing and UID/GID restrictions of Podman, requiring extra configurations and studying.

6.6 RootlessKit

To overview other rootless container options, RootlessKit will be overviewed in this chapter. Singularity Fakeroot will be discussed in the next chapter. The overviews exist in order to understand other rootless options available besides popular names, such as Docker and Podman, and the rootless methods available for them since they were not tested nor used.

RootlessKit is a Github project for rootless containers, offering a Linux implementation of “fake root” by utilizing user namespaces. The main goal of the project is to run Docker and Kubernetes as an unprivileged user, to protect the root of the host from potential container-breakout attacks. Rootless container projects, such as Podman and BuildKit, also utilize RootlessKit. RootlessKit creates, besides user namespaces, *mount namespaces,* and executes UID and GID mapping with *newuidmap*, *newgidmap*, *subuid* and *subgid*. There is also support for isolating *network namespaces [57],* a way to isolate the system resources associated to networking [70]*,* with userspace NAT (Network address translation) using *slirp*. All the namespaces are features of the Linux kernel [57].

To understand how RootlessKit creates rootless containers, it is important to understand the Linux kernel features in use. User namespaces are explained briefly in section 6.3. Mount namespaces produce a separate, isolated view of the filesystem mount points for the processes. When mounting is performed on a filesystem, the change will be noticed by all processes because they all share the same default namespace. As a new mount namespace is created, the new process gets a copy of the calling process mount tree and all mounts in the default namespace will be visible in the new namespace but the changes in the per-process mount namespaces will not be noticed outside of it. Mount namespaces are convenient because they provide a way for a contrasting file system layout to exist inside the container [71].

UID and GID mapping, one of the main concepts of rootless containers, is executed by allowing *newuidmap* and *newgidmap* to set UIDs and GIDs based on the arguments given to them. UIDs are set in */etc/subuid/*, a file list of user’s subordinate UIDs, and GIDs to */etc/subgid*, a file list of user’s subordinate GIDs. It is important to note that the root user is not immune from the requirement for a valid */etc/subuid* or */etc/subgid* entry. After the PID argument, *newuidmap* conjectures 3 integers: *uid*, *loweruid* and *count*. “uid” is the beginning of the range of UIDs inside the user namespace, while “loweruid” is the range of the UIDs outside the user namespace. “Count” is the length of the ranges, both inside and outside the user namespace. *Newuidmap* examines and ensures that the caller is the owner of the process indicated by PID and that for each of the above integers, each of the UIDs in the range is dedicated to the caller according to */etc/subuid/* before mapping UIDs between user namespaces. The same occurs with *newgidmap*, by replacing the UID processes of *newuidmap* with GIDs [72, 73].

In rootlessKit, the user UID is mapped to 0 (root UID) inside the *rootlesskit bash*. It is, however, not the real root of host because the shell is spawned with a new user namespace and a mount namespace. FIGURE 47 showcases how */etc/shadow/*, a file normally owned and accessed by root, has an owner called "nobody" and the group name of "nogroup". Despite the user information indicating that the user is root (UID = 0), the user cannot access the root owned directories [57].

```
(host)$ rootlesskit bash
(rootlesskit)# id
uid=0(root) gid=0(root) groups=0(root),65534(nogroup)
(rootlesskit)# ls -l /etc/shadow
-rw-r----- 1 nobody nogroup 1050 Aug 21 19:02 /etc/shadow
(rootlesskit)# cat /etc/shadow
cat: /etc/shadow: Permission denied
```

FIGURE 47. An example of how the root user inside shell is not actually root with RootlessKit (57)

6.7 Singularity Fakeroot

Singularity, a container platform, provides two modes for running containers as a non-root user: SETUID mode and Fakeroot mode. SETUID mode does not fall into the scope of Rootless containers, as the actual runtime binary gains the root privileges via the *setuid bit*. Fakeroot mode counts as a rootless mode but it does not support network namespaces with internet connectivity. It is not possible to protect the abstract sockets on the host from being connected from containerized processes unless internet connectivity is disabled [74].

The Fakeroot feature, a rootless mode, allows an unprivileged user to run a container as "fake root" user by leveraging user namespace UID/GID mapping, in a similar manner to other rootless containers. A "fake root" user has almost the same administrative rights as root but only inside the container and the requested namespaces. This indicates that the user can set different user/group ownerships for files or directories they own and change user/group identities with su/sudo commands. They also have full privileges inside the requested namespaces, such as network [75].

Restrictions and security come from the "fake root" user not being able to access or modify files and directories for which they do not already have access or rights on the host filesystem, which means the user will not be able to access root-only host files such as the host */root* directory.

Additionally, all files or directories created by the “fake root” user are owned by *root:root* inside container but as *user:group* outside of the container. FIGURE (48) showcases an example where the user is authorized to use the fakeroot feature and can use 65536 UIDs starting at 131072. The same applies to GIDs. If the “fake root” user, for example, creates a file under a bin user in the container, this file will be owned by 131073:131073 outside of the container. Administrator must ensure that there will not be overlap cases with the current user’s UID/GID on the system [75].

UID inside container	UID outside container
0 (root)	1000 (user)
1 (daemon)	131072 (non-existent)
2 (bin)	131073 (non-existent)
...	...
65536	196607

FIGURE 48. UID differences using the fakeroot mode of Singularity (75)

7 CONCLUSION

While the research on rootless containers and the current containers of the company was interesting and eye-opening for everyone involved, implementing rootless Docker containers proved to be a tentative task with its challenges. As mentioned in section 6.4, the first version of the non-root user was not able to complete the helper script in ENTRYPOINT that involved setting up the work environment for the developers starting and using the container. One of the solutions given and integrated was to give the user sudo privileges inside the container, through the configuration script *default.sh* in Dockerfile. With sudo privileges, the non-root user from the *--user* parameter was able to complete all the code from the helper script, setting up an environment that reflected the original, root accessed container run. These privileges, however, meant that the user could also become root with *sudo su -l* command, an unwanted effect of the given privileges. The user was also given *docker group* inside the container to run Docker commands; and as Docker daemon runs with root, this had the same result as giving the user of the group root privileges.

As the testing of the implementation proceeded further, most of the common test runs were able to pass inside the container. The CI pipeline jobs also included tests that had to be run for the code review each time new changes were made to the scripts. The last version of the implementation, however, was not able to pass tests from one of the jobs, returning a negative review for the code. This indicated that there might be some unknown root processes or commands that cannot be accessed with a sudo privileged user inside the container, let alone a non-root user. Updating the docker group for the user inside the container also required the user to update the shell session, which is commonly ensured through relogging inside the container or creating a new session. Both methods, however, caused unwanted issues with configuring the container environment to resemble the work environments it provided in the original version of the container run.

In conclusion, while the changes made for the container scripts were able to create a non-root user for the containers according to the main goals of the project, the original Docker scripts of the company with root access made it a challenge to integrate. These changes will not be merged into a main repository till all the issues have been solved with successful test runs, but the scripts provide a base for thoughts related to future work and safety around the containers used in the current software development environments of the company.

REFERENCES

1. AgileThought 2020. Complexity in Modern Software Development. Cited April 16, 2021. https://agilethought.com/blogs/complexity-in-modern-software-development/.

2. Madushan, Dhanushka 2018. Simplify Software Development by Using Automation. Medium. Cited April 16, 2021. https://medium.com/technology-in-essence/applying-ci-cd-to-large-software-development-process-e9d72ed76b95.

3. Basyildiz, Bora 2019. A Brief History of Container Technology. Section. Cited April 16, 2021. https://www.section.io/engineering-education/history-of-container-technology/.

4. Prajapati, Binal 2020. Top Reasons Why Docker is Popular. Medium. Cited April 16, 2021. https://javascript.plainenglish.io/top-reasons-why-docker-is-popular-31cc6056e82a.

5. Fiser, David & Oliveira, Alfredo 2019. Why A Privileged Container in Docker Is a Bad Idea. Trend Micro. Cited April 16, 2021. https://www.trendmicro.com/en_us/research/19/l/why-running-a-privileged-container-in-docker-is-a-bad-idea.html.

6. Google Cloud n.d. Containers at Google. Cited April 20, 2021. https://cloud.google.com/containers.

7. Red Hat 2018. Understanding virtualization. Cited April 20, 2021. https://www.redhat.com/en/topics/virtualization.

8. Red Hat 2020. Containers vs VMs. Cited April 20, 2021. https://www.redhat.com/en/topics/containers/containers-vs-vms.

9. Docker 2021. What is a Container? Cited April 21, 2021. https://www.docker.com/resources/what-container.

10. Avi Networks 2021. What are Microservices and Containers? Cited April 21, 2021. https://avinetworks.com/what-are-microservices-and-containers/.

11. Oberoi, Archna 2019. Monolithic vs. Microservices: Which Is the Better Architecture App Development? Microservices Zone. DZone. Cited April 21, 2021. https://dzone.com/articles/monolithic-vs-microservices-which-is-the-better-ar.

12. Red Hat 2019. What is container orchestration? Cited April 21, 2021. https://www.redhat.com/en/topics/containers/what-is-container-orchestration.

13. Kubernetes 2021. Production-Grade Container Orchestration. Cited April 21, 2021. https://kubernetes.io/.

14. Tyagi, Amit 2021. Getting Started with Kubernetes – Part Two. Cited April 21, 2021. https://www.c-sharpcorner.com/article/getting-started-with-kubernetes-part2/.

15. Kubernetes 2021. Kubernetes Components. Cited April 22, 2021. https://kubernetes.io/docs/concepts/overview/components/.

16. Kubernetes 2020. The Kubernetes API. Cited April 22, 2021. https://kubernetes.io/docs/concepts/overview/kubernetes-api/.

17. Docker Docs 2021. Docker Overview. Cited April 22, 2021. https://docs.docker.com/get-started/overview/.

18. Docker Docs 2021. Docker images. Cited April 22, 2021. https://docs.docker.com/engine/reference/commandline/images/.

19. Avi, 2019. Docker Architecture and its Components for Beginner. Geekflare. Cited April 22, 2021. https://geekflare.com/docker-architecture/.

20. Singh, Pradeep 2017. Basics of Containers, Docker and Container Orchestration. IoT Bytes. Cited April 22, 2021. https://iotbytes.wordpress.com/basics-of-containers-docker-and-container-orchestration/.

21. Docker Docs 2021. Docker ps. Cited April 22, 2021. https://docs.docker.com/engine/reference/commandline/ps/.

22. Docker Docs 2021. Use volumes. Cited April 22, 2021. https://docs.docker.com/storage/volumes/.

23. Docker Docs 2021. Networking overview. Cited April 22, 2021. https://docs.docker.com/network/.

24. Docker Docs 2021. Docker Registry. Cited April 23, 2021. https://docs.docker.com/registry/.

25. Buckler, Mark 2017. Developing in Docker. Cited April 23, 2021. http://www.markbuckler.com/post/docker-use/.

26. Docker Docs 2021. Best practices for writing Dockerfiles. Cited April 23, 2021. https://docs.docker.com/develop/develop-images/dockerfile_best-practices/.

27. Docker Docs 2021. Dockerfile reference. Cited April 23, 2021. https://docs.docker.com/engine/reference/builder/.

28. Docker Docs 2021. Sample application. Cited April 23, 2021. https://docs.docker.com/get-started/02_our_app/.

29. Rehkopf, Max, n.d. What is Continuous Integration? Atlassian. Cited April 25, 2021. https://www.atlassian.com/continuous-delivery/continuous-integration.

30. Katalon 2020. What is CI/CD? Continuous Integration & Continuous Delivery. Cited April 25, 2021. https://www.katalon.com/resources-center/blog/ci-cd-introduction/.

31. Petrocelli, Tom 2019. Version Control Systems: The Link Between Development and Deployment. CMSWire. Cited April 26, 2021. https://www.cmswire.com/information-management/version-control-systems-the-link-between-development-and-deployment/.

32. Chacon, Scott & Straub, Ben 2014. Getting Started – About Version Control. Pro Git. Second edition. Apress. Cited April 26, 2021. https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control.

33. Git SCM 2021. Cited April 26, 2021. https://git-scm.com/.

34. Chacon, Scott & Straub, Ben 2014. Getting Started – What is Git? Pro Git. Second edition. Apress. Cited April 26, 2021. https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F.

35. Chacon, Scott & Straub, Ben 2014. Git Branching – Branches in a Nutshell. Pro Git. Second edition. Apress. Cited April 26, 2021. https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell#ch03-git-branching.

36. Lebedyuk, Eduard 2018. Continuous Delivery of your InterSystems solutions using GitLab – Part I: Git. InterSystems. Cited April 26, 2021. https://community.intersystems.com/post/continuous-delivery-your-intersystems-solution-using-gitlab-part-i-git.

37. Dagdee, Pravin & C, Praveen & Nigam Vishwa & Nair, Lenin, 2017. Using CI/CD Over Containerization to Drive Down Pre-Production Costs. Container Journal. Cited April 26, 2021. https://containerjournal.com/topics/using-cicd-containerization-drive-pre-production-costs/.

38. Dangi, Sanket 2015. 5 Key Benefits of Docker: CI, Version Control, Portability, Isolation and Security. IOD Cloud Technologies Research Ltd. Cited April 26, 2021. https://iamondemand.com/blog/5-key-benefits-of-docker-ci-version-control-portability-isolation-and-security/.

39. Holmes, Marc 2016. The Modern Software Supply Chain Runs on Docker. Docker blog. Cited April 26, 2021. https://www.docker.com/blog/the-modern-software-supply-chain-runs-on-docker/.

40. Jenkins 2021. Cited April 26, 2021. https://www.jenkins.io/.

41. Jenkins 2021. Getting Started with Pipelines. Cited April 26, 2021. https://www.jenkins.io/pipeline/getting-started-pipelines/.

42. Shetty, Ritesh 2020. Best Jenkins Pipeline Tutorial for Beginners [Examples]. LaptrinhX. Cited April 26, 2021. https://laptrinhx.com/best-jenkins-pipeline-tutorial-for-beginners-examples-699780411/.

43. Bayer, Andrew 2017. Parallel stages with Declarative Pipeline 1.2. Jenkins. Cited April 26, 2021. https://www.jenkins.io/blog/2017/09/25/declarative-1/.

44. Jenkins 2021. Using Docker with Pipeline. Cited April 26, 2021. https://www.jenkins.io/doc/book/pipeline/docker/.

45. Sarai, Aleksa & Suda, Akihiro & Scrivano, Giuseppe, n.d. Rootless containers. Cited April 27, 2021. https://rootlesscontaine.rs/.

46. Tiigi, Tõnis 2019. Experimenting with Rootless Docker. Medium. Cited April 27, 2021. https://medium.com/@tonistiigi/experimenting-with-rootless-docker-416c9ad8c0d6.

47. Red Hat 2019. What is the Linux kernel? Cited April 27, 2021. https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel.

48. Boone, Joseph 2021. HDG Explains: What is Sudo & Root on Linux? Help Desk Geek. Cited April 27, 2021. https://helpdeskgeek.com/linux-tips/hdg-explains-what-is-sudo-root-on-linux/.

49. Docker Docs 2021. Docker security. Cited April 29, 2021. https://docs.docker.com/engine/security/.

50. Mouat, Adrian 2015. Docker security. Cited April 29, 2021. https://www.oreilly.com/content/docker-security/.

51. Campbell, Marc 2017. Processes in Containers Should Not Run As Root. Medium. Cited April 29, 2021. https://medium.com/@mccode/processes-in-containers-should-not-run-as-root-2feae3f0df3b.

52. Pavišić, Vlatka 2019. User privileges In Docker containers. Medium. Cited April 29, 2021. https://medium.com/jobteaser-dev-team/docker-user-best-practices-a8d2ca5205f4.

53. Docker Docs 2021. Post-installation steps for Linux. Cited April 29, 2021. https://docs.docker.com/engine/install/linux-postinstall/.

54. Sasson, Aviv 2020. Rootless Containers: The Next Trend in Container Security. Paloalto Networks. Unit 42. Cited April 29, 2021. https://unit42.paloaltonetworks.com/rootless-containers-the-next-trend-in-container-security/.

55. Kerrisk, Michael 2021. User_namespaces – overview of Linux user namespaces. Linux Programmer's Manual. Man7. Cited April 29, 2021. https://man7.org/linux/man-pages/man7/user_namespaces.7.html.

56. Kerrisk, Michael 2021. Setuid – set user identity. Linux Programmer's Manual. Man7. Cited April 29, 2021. https://man7.org/linux/man-pages/man2/setuid.2.html.

57. Suda, Akihiro 2021. RootlessKit. GitHub. Cited April 29, 2021. https://github.com/rootless-containers/rootlesskit.

58. Docker Docs 2021. Run the Docker daemon as a non-root user (Rootless mode). Cited April 30, 2021. https://docs.docker.com/engine/security/rootless/.

59. Stack Overflow Documentation 2019. Learning Docker. Docker ebook. Cited April 30, 2021. https://riptutorial.com/Download/docker.pdf.

60. Docker Docs 2021. Docker run reference. Cited April 30, 2021. https://docs.docker.com/engine/reference/run/.

61. ihritik, 2019. Python | os.getuid() and os.setuid() method. GeeksforGeeks. Cited April 30, 2021. https://www.geeksforgeeks.org/python-os-getuid-and-os-setuid-method/.

62. GeeksforGeeks 2017. Getpass() and getuser() in Python (Password without echo). Cited April 30, 2021. https://www.geeksforgeeks.org/getpass-and-getuser-in-python-password-without-echo/.

63. Linuxize 2021. How to Create Users in Linux (useradd Command). Cited April 30, 2021. https://linuxize.com/post/how-to-create-users-in-linux-using-the-useradd-command/.

64. Linfo 2007. The /sbin Directory. The Linux information Project. Cited April 30, 2021. http://www.linfo.org/sbin.html.

65. Computer Hope 2021. Linux chmod command. Cited April 30, 2021. https://www.computerhope.com/unix/uchmod.htm.

66. Podman 2021. Getting Started with Podman. Cited April 30, 2021. https://podman.io/getting-started/.

67. Henry, William 2019. Podman and Buildah for Docker users. Red Hat Developers. Cited April 30, 2021. https://developers.redhat.com/blog/2019/02/21/podman-and-buildah-for-docker-users/.

68. McCarty, Scott 2019. Understanding root inside and outside a container. Red Hat Blog. Cited April 30, 2021. https://www.redhat.com/en/blog/understanding-root-inside-and-outside-container.

69. Heon, Matthew 2019. Why can't rootless Podman pull my image? Red Hat. Cited April 30, 2021. https://www.redhat.com/sysadmin/rootless-podman.

70. Kerrisk, Michael 2021. Network_namespaces – overview of Linux network namespaces. Linux Programmer's Manual. Man7. Cited May 01, 2021. https://man7.org/linux/man-pages/man7/network_namespaces.7.html.

71. Containerlabs n.d. Mount namespaces. Cited May 01, 2021. https://containerlabs.kubedaily.com/LXC/Linux%20Containers/Mount-namespaces.html.

72. Kerrisk, Michael 2021. Newuidmap – set the uid mapping of a user namespace. Linux Programmer's Manual. Man7. Cited May 01, 2021. https://man7.org/linux/man-pages/man1/newuidmap.1.html.

73. Kerrisk, Michael 2021. Newgidmap – set the gid mapping of a user namespace. Linux Programmer's Manual. Man7. Cited May 01, 2021. https://man7.org/linux/man-pages/man1/newgidmap.1.html.

74. Sarai, Aleksa & Suda, Akihiro & Scrivano, Giuseppe, n.d. Singularity. Rootless Containers. Cited May 01, 2021. https://rootlesscontaine.rs/.

75. Sylabs Inc n.d. Fakeroot feature. Cited May 01, 2021. https://sylabs.io/guides/3.6/user-guide/fakeroot.html.