

Malla Nyrhinen

COMPATIBILITY ISSUES BETWEEN DOCKER AND PODMAN, AND THE ROLE OF OCI

A multivocal literature review

Master's Thesis
Faculty of Information Technology and Communication Sciences
Examiner: Professor Kari Systä
Examiner: University Lecturer Antti Sand
May 2025

ABSTRACT

Malla Nyrhinen: Compatibility Issues Between Docker and Podman, and the Role of OCI
Master's Thesis
Tampere University
Degree Programme of Software Development
May 2025

Containers are used worldwide by companies with a seemingly continuous increase in popularity. Docker is a currently a clear market leader but due to starting to require a subscription from enterprise clients without providing any extra features for the money, or users being unhappy with certain areas in Docker, like security. Podman is marketed as a Docker compatible solution and a rising alternative container technology. Both of these technologies are Open Container Initiative compatible. Open Container Initiative or OCI, creates and maintains standards for containers.

The goal of this master's thesis is to find out if there are compatibility issue areas between Docker and Podman and what is the role of OCI in those possible compatibility issues. Three research questions were selected for this thesis: Does Docker and Podman have compatibility issues, what are the common compatibility issues between Docker and Podman, and How does OCI help in mitigating these issues. The scope of the areas where compatibility issues were searched for was limited to daemon vs daemonless, compose, networking, rootless vs rootful, volumes, logging and monitoring, integration, and swarm and cluster mode.

The research method selected for this thesis is multivocal literature review, MLR. MLR uses on top of academic papers also grey literature. Grey literature is non-academic sources. The selected research method is a good choice as there isn't really published research but data from users is available. All the selected sources do fall into the grey literature scope.

Between Docker and Podman there are compatibility issues in multiple areas. Users have experienced issues in every research area except logging and monitoring, but even that was found out to be a possible compatibility issue. An important discovery is that the issues could be divided into having installations for both Docker and Podman or switching to use Podman from Docker. Most of the issues were if users had installations for both.

OCI has three specifications, runtime, image, and distribution. Compatibility issues found in the research didn't fall into any of those three areas. As all were outside of OCI's scope, so it was concluded that OCI mitigates issues in the areas it covers.

Keywords: Docker, Podman, OCI, Containers, Multivocal Literature Review

The originality of this thesis has been checked using the Turnitin Originality Check service.

TIIVISTELMÄ

Malla Nyrhinen: Compatibility Issues Between Docker and Podman, and the Role of OCI
Diplomityö
Tampereen yliopisto
Tietotekniikka
Toukokuu 2025

Konttitekniologiat eli kontainerit on käytössä yrityksillä ja suosio vaikuttaa nousevan vuosi vuodelta. Docker on tällä hetkellä selvä markkinajohtaja, mutta käyttäjät ovat alkaneet etsimään muita vaihtoehtoja. Tämä johtuu Dockeriin tulleesta maksullisesta versiosta yrityskäyttäjille. Dockerin maksullisen version ei koeta tuovan lisäarvoa verrattuna ilmaisversioon. Yksityiset käyttäjät eivät ole tyytyväisiä joihinkin Dockerin osa-alueisiin, kuten turvallisuuteen. Podmania mainostetaan yhteensopivana ratkaisuna Dockerille. Podman on suosiota kasvattava konttitekniologia. Molemmat näistä konttitekniologioista ovat yhteensopivia OCI:n kanssa. Open container initiative eli OCI luo ja ylläpitää standardeja konttitekniologioille.

Tämän diplomityön tavoite oli selvittää, onko Dockerin ja Podmanin välillä yhteensopivuusongelmia ja mikä on OCI:n rooli mahdollisissa yhteensopivuusongelmissa. Tutkimuksen tutkimuskysymyksiksi valikoitui kolme kysymystä: Onko Dockerin ja Podmanin välillä yhteensopivuusongelmia, Mitä yhteisiä yhteensopivuusongelmia on Dockerin ja Podmanin välillä sekä Miten OCI vähentää yhteensopivuusongelmia. Alueet, mistä yhteensopivuusongelmia etsittiin, rajattiin seuraaviin: daemon vs daemonless, compose, networking, rootless vs rootful, volumes, logging and monitoring, integration ja swarm ja cluster moodit.

Tähän tutkimukseen valittiin tutkimusmenetelmäksi moniääninen kirjallisuuskatsaus eli lyhyemmin MLR. MLR käyttää akateemisten lähteiden lisäksi harmaata kirjallisuutta. Harmaaksi kirjallisuudeksi lasketaan ei-akateemiset lähteet. Valittu tutkimusmenetelmä on hyvä valinta, jos akateemisia lähteitä ei ole paljon, mutta käyttäjien tuottamaa dataa on tarjolla. Kaikki tutkimukseen valitut lähteet sisältyvät harmaan kirjallisuuden tasolle.

Dockerin ja Podmanin välillä löydettiin olevan yhteensopivuusongelmia usealla eri alueella. Käyttäjät kokivat ongelmia kaikilla muilla alueilla paitsi 'logging and monitoring'. Kyseisellä alueella kuitenkin todettiin olevan mahdollisia yhteensopivuusongelmia. Tärkeä löytö tutkimuksessa oli, että yhteensopivuusongelmat johtuivat joko siitä, että käyttäjillä oli asennuksia kummastakin Dockerista ja Podmanista tai käyttäjät vaihtoivat käyttämään Dockerista Podmaniin. Suurin osa yhteensopivuusongelmista ilmeni, jos käyttäjällä oli asennuksia kummastakin.

OCI:lla on kolme spesifikaatiota, runtime, image ja distribution. Tutkimuksessa ilmenneet yhteensopivuusongelmat eivät menneet minkään näiden kolmen spesifikaation alueelle. Kaikki yhteensopivuusongelmat olivat niiden ulkopuolella. Tutkimuksen tuloksista tehtiin johtopäätös, että OCI vähentää yhteensopivuusongelmia alueilla, jotka se kattaa.

Avainsanat: Docker, Podman, OCI, Moniääninen kirjallisuuskatsaus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin Originality Check -ohjelmalla.

USE OF AI IN THESIS

I have utilised AI tools in my thesis:

- No
- Yes

The AI tools utilised in my thesis and their purposes are described below:

Names and versions of AI tools: ChatGPT versions GPT-4o and GPT-4.5.

AI was utilized in checking grammar and bouncing around ideas. Ideas being the format of the thesis and formatting research questions. The biggest use was utilizing AI in providing the research areas for possible compatibility issues between Docker and Podman, it was good at limiting the research into certain areas. Most importantly AI was only used as a tool, not as a replacement for the writer.

ChatGPT was used as a help during the whole process of this thesis for ideas and grammar checking.

I acknowledge that I am fully responsible for the entire content of my thesis, including the parts generated by AI, and accept accountability for any violations of ethical standards in publications.

PREFACE

A special thank you to my thesis supervisor Kari Systä, for not giving up hope on me, even if the start of this process was quite slow. Thank you for checking up periodically and giving valuable feedback and guidance. Thank you to my friends at the university for making the time here fun and fly past quickly. Lastly, thank you to my family and partner for the constant support and encouragement.

Tampere, 23.5.2025

Malla Nyrhinen

CONTENTS

USE OF AI IN THESIS	III
1.INTRODUCTION	1
2.BACKGROUND	3
2.1 Containers versus virtual machines	3
2.2 Containerization	4
2.2.1 Images	4
2.2.2 Containers	4
2.2.3 Registries	5
2.3 Docker	5
2.4 Podman	7
2.5 Open Container Initiative	8
3.BASICS OF DOCKER AND PODMAN	10
3.1 Daemon and daemonless	10
3.2 Compose	10
3.3 Networks	11
3.4 Volumes	11
3.5 Commands	12
4.OCI SPECIFICATIONS	14
4.1 Runtime specification	14
4.2 Image specification	15
4.3 Distribution specification	15
5.RESEARCH METHODOLOGY	17
5.1 Multivocal literature review	17
5.2 Data sources	17
5.3 Research process	19
6.COMPATIBILITY ISSUES	21
6.1 Daemon vs daemonless	21
6.2 Compose	23
6.3 Networking	26
6.4 Rootless vs rootful	28
6.5 Volumes	29
6.6 Logging and monitoring	31
6.7 Integration	31
6.8 Swarm and cluster mode	32
7.RESULTS	33
7.1 Does Docker and Podman have compatibility issues (RQ1)	33

7.2	What are the common compatibility issues between Docker and Podman	
(RQ2)	35	
7.3	How does OCI mitigate these issues (RQ3)	38
8.	CONCLUSIONS	39
	SOURCES	41
	APPENDIX A: MULTIVOCAL RESEARCH SOURCES	44

LIST OF APPREVIATIONS

AWS	Amazon Web Service
CLI	Command-Line Interface
ECR	Elastic Container Registry
MLR	Multivocal Literature Review
OCI	Open Container Initiative
OS	Operating System
SLR	Systematic Literature Review
WSL	Windows Subsystem for Linux

1. INTRODUCTION

It was estimated that by 2025 more than 85% of companies worldwide will use containerized applications in production (The rise of containers 2022). In recent times, Docker is the leading containerization technology with an estimate of 83.21% market share (Top 5 containerization technologies 2024).

Even though Docker is the clear market leader currently, and article by Pandey highlights why there is a switch happening to Podman or other solutions. One big reason is Docker Desktop requiring payments from enterprises that cross a certain limit, without actually providing great improvements. Many started thinking if Docker is a worthy option anymore. Other reasons for switching are Dockers issues with systems not running Linux, security, and architecture. (Pandey 2025) Podman is in many ways like Docker as it can run Docker formatted images and shares commands formatted in same ways (docker pull vs podman pull). Podman has been created as an alternative to Docker and it's compatible with open container initiative (OCI) standards (Aglave 2023; Aleksic 2023).

With the long-lasting popularity of Docker but users switching to alternative solutions, like Podman, this thesis aims to find out answers to these research questions:

- Does Docker and Podman have compatibility issues?
- What are the common compatibility issues between Docker and Podman?
- How does OCI help in mitigating these issues?

The research will be done using a multivocal literature review (MLR) to get experiences and information from both published material and grey literature. On top of that this thesis will contain a comparative analysis from existing material on Docker and Podman.

This thesis is constructed of eight chapters. The first chapter tells what the goal of this thesis is and how to achieve it. The second chapter provides background information overall from container technologies and needed terminology, as well as, telling in little more detail about Docker, Podman, and Open Container Initiative (OCI). In the third chapter Docker and Podman are looked into more on a technical detail. Next in the fourth chapter the OCI specifications are looked at in more detail. The fifth chapter gives information about the research process. More specifically it tells in more detail about the MLR process, what is the used grey literature and about the different sources for the data, and then what the research process looks like. In the sixth chapter the MLR is conducted. In

the chapter especially the possible compatibility issues are focused on by going over the sources selected, what issues users faced whilst also providing background on what users were doing and what was their goal. If the selected sources provided a solution to user's problems, those will be provided as well. The sixth chapter also tells more in depth about the deployment and OCI specifications. Second to last, in the seventh chapter the results are revealed to research questions and discussion is added. Finally in chapter eight, the research is wrapped up by going over the results, limitations, and what would be beneficial in a similar future research.

2. BACKGROUND

Let's first look briefly at the differences of containers and virtual machines and then move on to containerization and go a little more into detail about images, containers, and registries. Let's also look at more in depth about Docker, Podman, and OCI. These are important subjects in understanding this thesis.

2.1 Containers versus virtual machines

Containers and virtual machines can be compared on two different sides. In packaging, containers are the more lightweight solution. But on the virtual side, the object of virtualization is different. Containers virtualize OS and virtual machines virtualize hardware. This is visualized in the Figure 1 below.

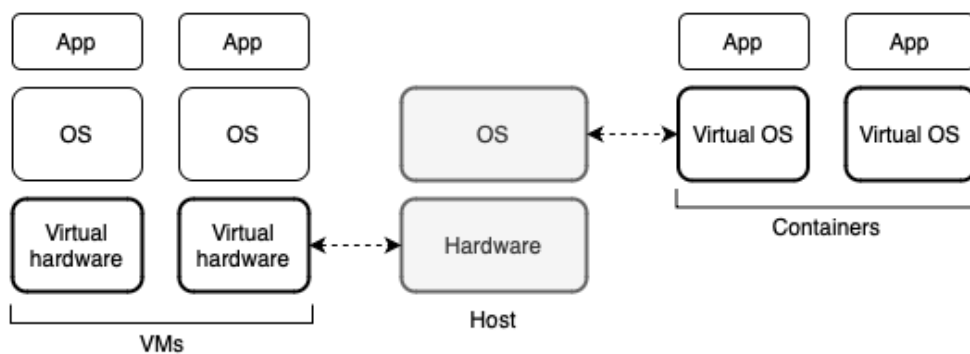


Figure 1. Virtualization from containers and VMs (Poulton 2024)

Using virtual machines, you need a lot more resources from your machine compared to containers as you need to run more operating systems. Compared to containers where you only need the shared OS from the host. (Poulton 2024)

Containers have many advantages over VMs. They are more portable, faster, and use less resources. Containers are good for applications that are portable and scalable. VMs on the other hand are better for isolated applications. (Containers vs VMs (virtual machines): What are the differences?) VMs and containers aren't technologies where you can only choose from one. They can be run together with containers on top of the virtual machines. (Poulton 2024)

2.2 Containerization

In 2018 Schenker said that “There has never been a new technology introduced in IT that penetrated the landscape so quickly and so thoroughly than containers”, and currently, the most popular way to run modern applications is indeed to use containers. In the current age where software is a big part of the world, it is extremely important that new features, updates, etc. can be supplied in the most efficient way as possible, and this is what containers do. Even legacy applications have a significantly reduced time between new releases if it has been containerized. Before containers, for example, the differences in libraries and dependencies between production and development environments could lead to applications failing on production. (Poulton 2024; Schenker 2018)

The issue just mentioned, was fixed by Docker after introducing a standardized way to package and run applications, and now it is much easier to share and run them. From the start to a running container, there are only four steps that needs to be done. Firstly, the application is to be developed or use an existing one. Second the developed app is turned into an image. Third step is not mandatory, but the image is placed into an image registry. Finally, the image can be run as a container. When the application is packaged to an image, the process is called containerization. For companies, most importantly, containers have reduced costs as development has become faster and marketing time has come down (Poulton 2024; Schenker 2018).

2.2.1 Images

An image is made up from multiple layers. These layers contain: a base image, libraries, binaries, dependencies, and configuration files. The base image, or the bottom layer, stores the files for operating system (OS) and filesystem files. After the base image there are layers that contain libraries, binaries, dependencies, and configuration files. (Powell & Smalley 2024) One image can be used to start one or more containers. (Poulton 2024)

2.2.2 Containers

The purpose of containers is to run processes in isolated runtime environments. Containerization is what makes it possible to run programs across platforms and cloud because it packages the application and whatever it needs to run into a single unit: container. (Arrichiello & Salinetti 2022; Smalley & Susnjara 2024) An operating system (OS) can run multiple containers, and each container a different application, but the containers won't know about the existence of others, because they are isolated (Poulton 2024).

2.2.3 Registries

Registries are repositories used for storing and distributing container images. Images are pushed and pulled from registry by a developer, or whoever has the need for it. (Poulton 2024; Powell & Smalley 2024) Registries can be divided into public registries and private registries. Public registries provide images to everyone through a public website. DockerHub is an example of a public registry. The public registries are usually used by individuals and teams that aren't too large in size. Also, when a registry is desired to be setup quickly. But growth will introduce security problems, this is where private registries come into play. Private registries provide advanced security and even technical support, and they are hosted either off-site or on-site. The access to private registries is limited to only authenticated users and images can't be seen from public websites. One example of a private registry is Amazon's ECS. (Docker Registries; What is a container registry 2022) For the user there are many options to choose as a container registry, as many organizations and communities that are involved with containers, have developed their own registries with unique interfaces. Some of the available registries for the users are Docker Hub from Docker, Google offers Google Container Registry and Amazon Web Service (AWS) has Amazon Elastic Container Registry (ECR). (Arrichiello & Salinetti 2022; What is a container registry 2022)

2.3 Docker

After being published in 2013, Docker quickly gained traction in the container community. (Arrichiello & Salinetti 2022)

Docker engine is what orchestrates the functionality for building and running containers from users prompts. The Docker architecture is divided into three main components: client, host and registry. (Arrichiello & Salinetti 2022; What is Docker?) The architecture can be seen in Figure 2 below:

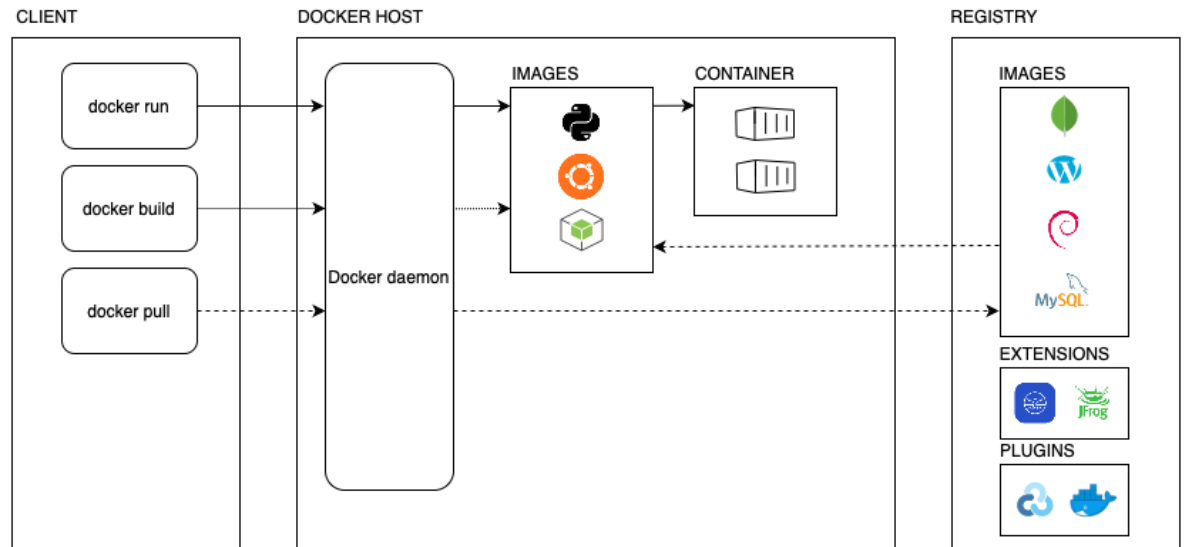


Figure 2. Docker architecture (What is Docker?)

The daemon running in the host is the bridge between components and it is the most important piece to keep containers running. It's a background service that listens to API calls from the client side, handles pulling and pushing to registries, and lastly handles the components in the host component: containers, images, networks, and storage volumes. (Arrichiello & Salinetti 2022) Docker used to be installed with a root, but nowadays it can be used in a rootless mode (Voulgaris *et. al* 2022).

Docker image format release was at the time revolutionary compared to other container technologies. Over time Docker started applying an image specification which then evolved to comply to OCI's specification for images. The Docker images are distributed using Docker registry, and that used to be a standout feature. (Arrichiello & Salinetti 2022)

Docker has its own image registry, Docker Hub, which is the default registry used from Docker. Using the daemon, user is able to push and pull images to the registry. (Arrichiello & Salinetti 2022) The Docker hub is possibly the most popular container registry for users to choose from. This is due to most users using Docker. The free option was very popular for a while, but it was limited due to abusing it. (Pialoux)

As has become clear, Docker adopts a monolithic model, where the goal is to offer a one single solution to manage of the whole container lifecycle. This includes all the tasks related to containerization and images. (Voulgaris *et. al* 2022)

2.4 Podman

Podman, which is short for pod manager, was released four years after Docker in 2017, by RedHat. It's an open-source tool that enables creating, managing, and running containers for many Linux distributions. (Arrichiello & Salinetti 2022; What is Podman? 2024) Podman is also available to macOS and Windows users. Because it requires a Linux kernel, mac users need to use virtual machine, and Windows users need Windows Subsystem for Linux (WSL) 2. Specifically, the WSL 2 version instead of WSL 1, as that won't work. (Walsh 2023) Unlike Docker, Podman needs additional tools for the overall creation and management of containers, as Podman itself doesn't handle the building of containers or moving them to storage systems. Some examples of these tools are Buildah and Skopeo. Buildah is used to build containers from the beginning or with images as starting points. Skopeo is what is used to moving the images between storage systems like registries or in your local system. (Arrichiello & Salinetti 2022; What is Podman? 2024) Where Docker offers a single solution to handle everything, Podman relies on the Unix philosophy "Make each program do one thing well" (Voulgaris et. al 2022).

The whole Podman ecosystem is managed with Lipod library (Arrichiello & Salinetti 2022; What is Podman? 2024). The lipod library was fundamental in building Podman. It contains all the critical logic that is needed for maintaining container's lifecycle, which makes it an irreplaceable piece for Podmans origin, and its development is the reason for what Podman is today. The library has the following scope:

- Container image format management for the whole lifecycle of the image.
- Managing the container lifecycle – from the very beginning to the end and other functionalities in between.
- Container and pod management and managing both together.
- Support for rootless containers and pods, which rootless users can manage.
- Container resource isolation management.
- Support a CLI that can also be used as an alternative to Docker.
- Provide a REST API that is Docker-compatible. (Arrichiello & Salinetti 2022)

Unlike Docker, Podman follows daemonless architecture. The architecture can be seen in the Figure 3 below.

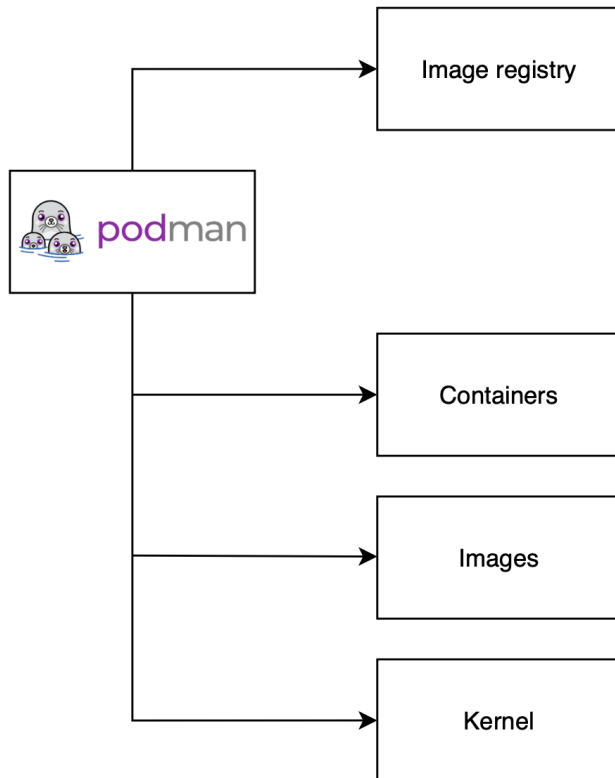


Figure 3. *Simplified Podman architecture (Arrichiello & Salinetti 2022)*

Daemonless meaning that after installation is complete, the user doesn't need to start any services, containers can just be run after the installation is finished. The Podman binary functions as the command-line interface (CLI) and as the container engine, which coordinates the container runtime behind the scenes. (Arrichiello & Salinetti 2022)

Like the scope of Lipod library and Podman's name itself revealed, Podman uses pods. Pods group multiple containers that run together and they share the same resources. This is done similarly to how Kubernetes does it. A pod is built from one infra container and number of containers. The infra container is responsible for keeping the pod running and managing user namespace, isolating the pod from the host. Regular containers have a monitor that tracks its processes and flags dead containers. The dead containers are containers that have stopped functioning but can't be removed due to some of their resources still being in use. Pods are controlled with a CLI and the Lipod library. (What is Podman? 2024)

2.5 Open Container Initiative

Open container initiative, OCI, is a project started in 2015, run under the Linux foundation, and led by the key players in the container industry at that time, such as Docker and

CoreOS. OCI is responsible for creating and maintaining the standards for container formats and runtimes in the container industry. Today there are three specifications that are maintained by OCI: the Runtime Specification, the Image Specification, and the Distribution Specification. The Image specification defines the standards for image formats, like structure, contents, and metadata. The Runtime specification is responsible for giving definitions on how to unpack images and then executed as containers. Lastly, the Distribution specification tells how container images should be distributed using container registries. (About the Open Container Initiative; Poulton 2024).

The most used OCI container runtime is Runc. Runc was donated to OCI in 2015 by Docker, with the purpose of helping establish a standard to the growing container ecosystem. (Arrichiello & Salinetti 2022)

Almost every available container system in the market is compatible with the OCI specification. Docker and Podman both are, and as such they should be able to be used as a drop-in replacement for each other. (Voulgaris *et. al* 2022)

3. BASICS OF DOCKER AND PODMAN

Before going into the research part of this thesis, let's look a little deeper into the basic functions that Docker and Podman have. We learned that on the architecture level there are differences like the daemon vs. daemonless architecture, so to get an even deeper understanding in the workings of these two container technologies let's shortly go over how these container technologies differ on the areas that are important in the research phase. These areas are daemon vs daemonless, compose, networks rootful vs rootless, and volumes. Let's also look at differences on command level.

3.1 Daemon and daemonless

As was said, docker daemon, called dockerd, is the piece that connects client, registry, and host together. The daemon is also responsible for the different Docker objects, like images and containers (Arrichiello & Salinetti 2022; What is Docker?) The interaction between docker daemon and docker client happens with a UNIX socket. This socket is what allows users to act as either root or non-root user and is an important piece in container management. Socket is what transfers the information between Docker client and Docker daemon (Rajyashree, Senthilkumar, Saravanan & Sakthivel 2024)

Even though Podman is daemonless by nature, it is still fully compatible with Docker images. Podman is also able to use the same container runtimes as Docker to launch containers. (Walsh 2023) With no daemon there is also no need for Podman to listen on a UNIX socket for commands. But by running a Podman service and then making it listen for a UNIX socket, users can expose the REST APIs. (Arrichiello & Salinetti 2022) Through the REST API, Docker compatible tools can also communicate with Podman, like docker-compose (Walsh 2023).

Docker's use of daemon often means that it needs root access to the machine. This can then lead to security vulnerabilities. Being daemonless, Podman doesn't require root access and therefore at least in a security sense, is a better option. (Roach 2024)

3.2 Compose

A very popular compose tool for both Docker and Podman is Docker Compose. It's a project started all the way back in 2014 and it was made so that a set of containers can be managed with a YAML definition. The YAML definition was even later given its own specification: Compose spec. (Heon 2022)

At the start of Podman, it didn't support Docker Compose from the beginning. Due to users wanting to use their Compose YAML files, a Podman Compose community project was born. On the downside Podman Compose does not support all the functionality Docker Compose provides. Later Podman added an API which is compatible with Docker Compose, and it remains as the more popular choice amongst users. (Heon 2022)

The command for using compose has two ways of typing it that can be found: `docker-compose` and `docker compose`. The first one is from Compose v1, which no longer receives updates, but can still be used. The v1 was made with Python programming language. The command `docker compose` comes from Compose v2, which was released in 2020, and contains improvements in many different areas. (Migrate to Compose v2) The newer version was made with GO and was a complete rewrite of the Compose v1. For Podman support to v2 was added on 2022. (Heon 2022)

When used with Podman, both Docker Compose and Podman Compose have their positives and negatives. Docker Compose contains more features and has a lot more users and support. Podman Compose on the other hand is better at utilizing Podman's features and a lighter option for Podman. (Heon 2022)

3.3 Networks

Through networking containers are able share data and resources by communicating with other containers or hosts. There is a Container Network Model (CNM) -standard, which was proposed by Docker, a Container Network Interface (CNI) -standard, proposed by CoreOS. (Container Networking: What You Should Know)

There are two network backends that are supported by Podman. The default is Netavark and the other option is CNI. Netavark support was added in Podman version 4.0 and CNI support was deprecated. Support for CNI will be removed when Podman version 5.0 is released. (Podman Network)

3.4 Volumes

Volumes are good for persisting data long term, the data in them isn't removed even if the container using the volume is. Direct access to data inside the volume is not supported, it can lead to breaking the data or even the volume itself. Access needs to happen through a container. (Storage)

For Docker, the Docker daemon controls the volumes (Storage). There are two types of volumes for Docker: named and anonymous. Anonymous volumes are named by the

Docker host, and they are they aren't available to all of the containers automatically. (Bind mounts)

In Podman the volumes are managed by Podman Volume, which is a set of subcommands (How to manage storage with Podman volumes 2024). Podman also supports named and anonymous volumes (Volume).

3.5 Commands

The commands between Docker CLI and Podman CLI are compatible, to make the transition easy between the two tools. The difference in using commands is that for Podman there is no need to have a running daemon that listens and executes the commands. (Arrichiello & Salinetti 2022) Let's next look at what some of the most common commands look like both in Docker and in Podman.

Command's purpose	Docker command	Podman command
Build an image	build	build
Copy files or folders between a container and the local file system	cp	cp
Run a command in a container that is running	exec	exec
Kill a number of running containers	kill	kill
Load an image from container TAR archive or stdin	load	load
Fetch logs of a container	fetch	fetch
Manage pods	n/a	pod
Listen to running containers	ps	ps
Pull/push an image or a repository from a registry	pull/push	pull/push
Remove a number of containers	rm	rm

Remove a number of images	rmi	rmi
Start number of stopped containers	start	start
Stop number of running containers	stop	stop
Create a TARGET_IMAGE tag that refers to SOURCE_IMAGE	tag	tag

As can be seen, all shown of the basic commands are identical for Docker and Podman. The only difference seen in the table is the pod command missing for Docker, it can be only found for Podman. In the bigger scale there are other commands that can only be found for Podman and some only found in Docker. The reason for the similarities between Docker and Podman is due to the fact that the Podman developers wanted to keep the feeling as close as possible to the most famous container technology, Docker, at the time to help in the migration to Podman. It was even said that you can use existing Docker scripts for Podman by just creating an alias with a command: alias docker=Podman. (Arrichiello & Salinetti 2022)

4. OCI SPECIFICATIONS

Let's dive a little deeper on what the defined OCI specifications are. As we found in chapter 2 there are currently three specifications: runtime, image, and distribution specifications.

4.1 Runtime specification

The runtime specification provides definition to containers for configuration, execution environment, and lifecycle. Container's configuration is defined in the configuration file, `config.json`, for platforms that are supported. The specification has five principles with the goal of standardizing containers so that any standard container can be run in an OCI compliant runtime, without having extra dependencies and no matter what the content is. With these principles the specification gives definition to configuration file formats, set of standard operations, and execution environment. (Open Container Initiative 2016)

These five principles are:

1. Standard operations: for standard containers there are a standard set of operations. The operations can be done using a standard set of containers, filesystem, and networks tools.
2. Content agnostic: standard operations behave exactly the same way, no matter the content of the container. Containers are started the same way whether they contain a database, an application, or build files.
3. Infrastructure agnostic: standard containers can be run in any OCI supported platform.
4. Designed for automation: since standard containers use the same operations no matter the content and infrastructure, they work very well for automatic workflows.
5. Industrial-grade delivery: by following the specifications above, organizations of varying sizes are able to make their software delivery process faster and contain more automation.

Standard containers are redefining how the software is built, packaged, and delivered, whether for internal or external development, by providing a consistent, automated, and reliable foundation for modern software deployment. (Open Container Initiative 2016)

4.2 Image specification

The image specification is what defines the OCI compliant images, OCI images. The specification aims to support the creation of interoperable tools for different stages of container image. It's made up of four parts: image manifest, an optional image index, set of filesystem layers, and a configuration. (Open Container Initiative 2024a)

The image manifest sets three goals. The first one is that images are content-addressable. A unique id is generated to image and its components by hashing the images configuration. The second goal is allowing multi-architecture images. This is done through something called a "fat-manifest", which references multiple architecture-specific manifests. OCI codifies it in image index. The last goal is for the image to be translatable to OCI runtime specification. These goals define configuration and layers for a single container image targeted at a specific architecture and OS. (Open Container Initiative 2024b)

The image index contains information for a set of images that can spread across multiple architectures and OS's (Open Container Initiative 2024b). It points to specific image manifests and the image index itself is a higher-level manifest (Open Container Initiative 2024c).

OCI details how filesystem and filesystem changes can be serialized into a layer. Those layers are then stacked on top of each other to make a complete filesystem. (Open Container Initiative 2024d)

Lastly, the configuration defines the JSON structure to describe the container images for runtime and execution tools, and explains how it relates to filesystem changesets, which are detailed in layers. (Open Container Initiative 2025a)

4.3 Distribution specification

The last specification, distribution, defines a standard API for sharing and moving container content between different systems. The distribution specification is affiliated with the other two specifications (Open Container Initiative 2023). The moving and sharing of content is done in a standardized way, the process is the same everywhere, whatever the tool or platform is. The design is also generic and can be used as a distribution mechanism for other types of content as well. (Open Container Initiative 2023; Open Container Initiative 2025b)

The API works over HTTP. Users can check if a registry supports the distribution specification by doing a GET request to endpoint `/v2/`, and with a 200 answer, the specification is supported. (Open Container Initiative 2023)

The distribution specification has four different use cases. It can be used for content verification, resumable push and pull, and de-duplication of layer upload. (Open Container Initiative 2025b)

5. RESEARCH METHODOLOGY

In this chapter we will look deeper into the selected research method, as well as justify why it has been selected. We will also explain the types of data sources that are selected, the inclusion and exclusion criteria for the data, the process of collecting data, and finally what method is used in analyzing the data. The selected research method for this thesis is multivocal literature review (MLR).

5.1 Multivocal literature review

MLR is like Systematic Literature Review (SLR), but with the inclusion of grey literature. The software engineering field is heavily focused on practice and applications; therefore, it is beneficial to include also data from sources outside of academia. (Garousi, Felderer & Mäntylä 2019) For the topic of this research, there isn't that much existing research. Therefore, it's beneficial that we use sources that have been produced a lot in the form of grey literature, and this makes the MLR an excellent choice as a research method for the selected topic of this research.

5.2 Data sources

For this research, data from varying types of sources is used. The sources can be either formally published data i.e. "white literature" or part of the grey literature. Where we can trust the white literature to be scientific and creditable, that is not the case with grey literature. The grey literature can be divided into three tiers, as can be seen from Figure 4 below, and the scale goes from known to unknown in both axes:



Figure 4. *Tiers of grey literature (Garousi et al. 2019)*

The expertise-axle tells how well the contents producer actually know about what they have said in the source. The axle for outer control tells the degree to which the content is produced, moderated, or edited according to criteria for knowledge creation. Then inside the modal the content is divided into white literature and different shades of grey to correspond how creditable the source is, and the grey literature is divided into tiers that go from one to three. In the first tier, which is the lightest grey, are sources that are highly creditable, and the outlet is highly controlled. The first tier contains sources such as books and government reports. The second tier for grey literature has only moderately controlled output and the sources are only moderately creditable. The shade has gone more darker from the first tier to describe this. This tier contains sources like news articles, Q/A sites, and wiki articles. Lastly, the third and darkest tier, has the least controlled output and creditability. This tier contains sources that are for example blogs, emails, and tweets. (Garousi et. al. 2019)

For this research the grey literature comes mostly from the second tier. Many of the selected sources are from Q/A sites, such as GitHub issue posts and then from sources like Stackoverflow. The reason for this is that they directly tell that users are experiencing issues in certain areas and also where the issue might come from and if there is a solution for it.

5.3 Research process

The process for the multivocal research done for this thesis, can be seen in the Figure 5 below. It was formed based on the instructions on how to conduct a multivocal literature review from 2019 by Garousi et. al.

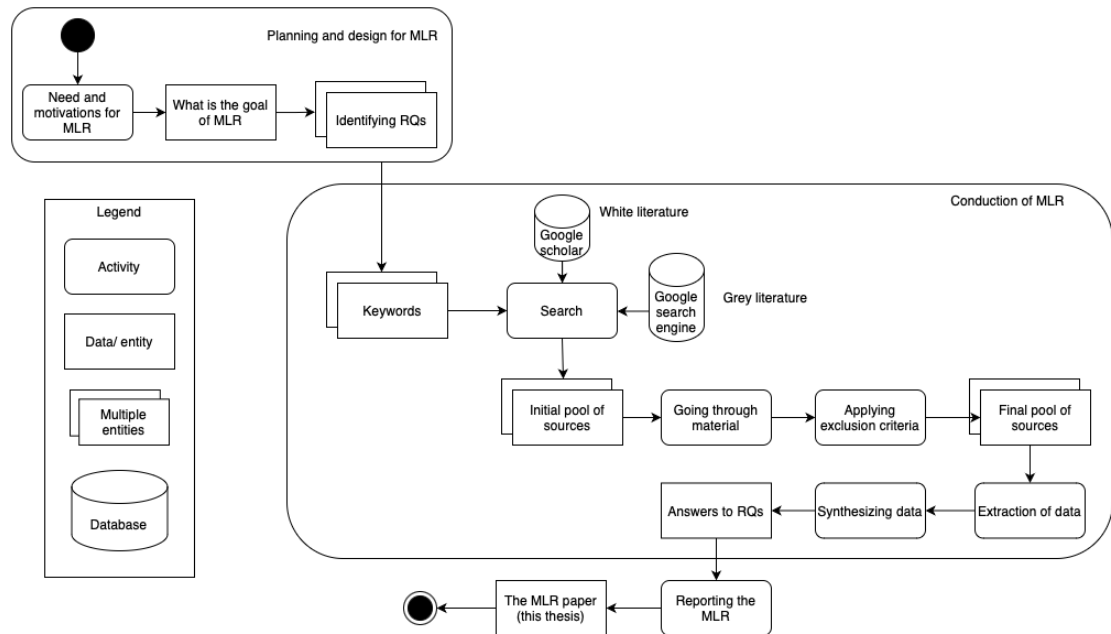


Figure 5. MLR process diagram (Gerousi et. al. 2019)

The process was started with the planning and design phase. The motivation was determined as well as what are the goals. With those the three research questions were formed.

With the research questions in mind, the next step in the process is conducting the multivocal literature review. First the keywords are formed. For this research the keywords are for example: Docker, Podman, issues, problems, and the area where the problems might occur, like compose or networking. With the keywords, search expressions are formed and used in different search engines. For this research, the white literature is searched from Google scholar, and grey literature from Google search engine. Next after the initial pool for sources is gathered, the process of reading through it and exclusion is applied. For this research the following exclusion criteria is selected:

- The source can't be older than from 2020
- If the problem is caused by a clear user error
- No duplicate sources
- Issue not about compatibility, but rather only from Docker or Podman

- Error due to inexperienced user

The duplicate source meaning, that if a user has posted about getting the same error to multiple different sites, only one is selected for this research as a source. Sources where user is learning about either containers or Docker or Podman have been left out if it's not clear that if the issue is real or just user not knowing what they are doing. After the exclusion criteria is applied, the final pool of resources is selected. Then from that pool data is collected and synthesized, and finally the research questions can be answered.

The last part of the process is reporting this research creating a paper, which this thesis will serve as.

6. COMPATIBILITY ISSUES

In this chapter let's look at the sources found for the different types of compatibility issues between Docker and Podman. The different kinds of issues users have posted in forums, bug posts or as a blog post, are explained and if there has been a solution or a work-around, it's added as well.

For researching the different possible compatibility issues between Docker and Podman, we first discussed with an AI about what different areas the issues might fall in. The result list of the areas can be found in Appendix B. This limited the area of research as well to keep this research in a good size. Then based on the result the search expressions were formatted. So, for example, one result was possible compose problems, then the search expression was formed like: 'Docker Podman issues compose'. First, we look at all the different compatibility issues that have risen between Docker and Podman. After the initial pool of resources was gathered, the data was read through and in the previous chapter mentioned exclusion criteria was added. From that we are left with the final pool of resources and those are opened up more in the following minor chapters. The chapters are divided into areas where the compatibility issues were found in.

6.1 Daemon vs daemonless

The first area where compatibility issues might come is the difference in Docker having a daemon and Podman following a daemonless architecture. As was discussed in chapters 2 and 3, Docker adopts rootful mode by default and runs it with a daemon and the daemon communicates with a client using a socket. The socket is created at location

```
/var/run/docker.sock
```

by default (Dockerd). Whereas Podman is daemonless, and it's capable in running either as rootful or rootless. For Podman to enable docker compatible tooling through REST APIs, it also uses sockets (Arrichiello & Salinetti 2022; Walsh 2023). Depending on the mode the socket paths are defined as follows:

```
rootful: /run/podman/podman.sock
```

```
rootless: /$XDG_RUNTIME_DIR/podman/podman.sock.
```

The rootless socket path is for example:

```
/run/user/1000/podman/podman.sock. (Podman System Service)
```

The different expectation for the socket paths when it comes to Docker and Podman seem to then

ultimately lead to some issues for the users. Let's next look at the issues, marker with "I" and following with an issue number, that can arise from mismatching socket paths.

- **I1: Multiple sockets**

Even though user is able to run basic Docker commands like pull, build, and commit, after installing Podman-Docker, querying Docker socket for images results into an empty result. Directly using Podman socket doesn't help either, the result stays the same. Pinging the socket tells that the socket is running and results into OK. And, when listing images, the result is as expected. So, what is wrong then? The socket was set differently than what the user thought it would be. Instead of being the rootful version, corresponding what Docker would be, it was set as the rootless version:

```
/run/user/1000/podman/podman.sock.
```

After setting the DOCKER_HOST environmental variable to correspond the rootless version, the problem was fixed. [S1] This issue already highlights how delicate the difference is, and though the problem was small, it led into a broken workflow.

- **I2: Mounting socket inside container**

When trying to mount podman.sock into a container there can be an issue where Podman changes the context of files in the container to 'container_file_t', and Docker doesn't do that. The issue was fixed after user changed the context on Podman socket on the host, but the situation felt frustrating for the user. They even stated that they were able to get it to work with Docker and mounting Docker socket, but when Podman is running on host the problem started. With Docker the efforts to make it work took less than an hour but with Podman the efforts spanned to multiple days. [S2]

- **I3: Symlink trouble**

For users having a symlink between rootful Docker and rootless Podman can lead to 'Permission denied' – errors [S3, S4].

Firstly, an error can be encountered when running Podman in rootless mode and creating a symlink from /var/run/docker.sock to /run/podman/podman.sock. This creates an access error for user. They don't have permission to access the Podman socket, even after trying to run in a privileged mode. Docker is most of the times rootful, but Podman can be either, which can lead to confusion. In this case it seems like running Podman in rootless mode, doing the symlink from rootful

socket to a rootless socket leads to the issue. It is even stated by the developers that Docker based tools expect the Dockers default rootful socket at location `/var/run/docker.sock`, therefore a patch or an approach is proposed where the rootful and rootless issue would be more properly informed for the user. [S3]

Secondly, when user tries to use docker-compose as root, it works as expected, but when they change to user, permission error happens. Even after some troubleshooting curl commands reveal that as a root user works:

```
$ sudo curl -H "Content-Type: application/json" --unix-socket
/var/run/docker.sock http://localhost/_ping
OK
```

But, as non-root user, trouble persists:

```
$ curl -H "Content-Type: application/json" --unix-socket
/var/run/docker.sock http://localhost/_ping
curl: (7) Couldn't connect to server
```

Again, the problem seems to be that the Podman socket is at the wrong location, when trying to run as rootless, the symlink from `var/run/docker.sock` to `/run/podman/podman.sock` doesn't work. For this discussion the proposed solution was to reverse running Podman with `'systemctl enable podman'` therefore breaking the link. Instead, the socket should be created at the rootless location and just like in Issue 1, the `DOCKER_HOST` environment variable should be set as such. [S4]

6.2 Compose

Using Docker Compose for Podman is a good choice for users, as it offers more features and therefore also more versatility. It's compatible with both Docker and Podman and its large user base means it has been tested more extensively, making it a stable option to use. Podman compose exists but it is a community project, it's not affiliated with Podman, so for now let's keep our focus on Docker compose when searching for compatibility issues. (Heon 2022) Even though Docker compose is usable with Podman, there still are some issues that can come be, so let's look at those next.

- I1: Docker binary

If Docker binary is installed, there were at least couple of issues reported where Podman can't be used by docker compose. The issues are similar to each other, but the error message is different. For the setup user uses system wide Podman service enabled, and environment variable set to point to the rootless Podman

socket path, discussed in chapter 5.1.1. [S5, S6] So let's see what the different issues are.

On the first source, the bug comes right away when trying to run

```
docker-compose up
```

An error of server not providing an image id and writing to location pops up for the user. One proposed solution is to use

```
COMPOSE_DOCKER_CLI_BUILD=0
```

for build, and for run/exec to stop using the docker binary use

```
COMPOSE_INTERACTIVE_NO_CLI=1.
```

The suggested problem seems to be that docker-compose tries to use the docker binary, so the proposed solution would avoid it. It remains unclear if this was able to resolve the problem for the user. [S5]

For the second one docker binary is unable to communicate with an API provided by Podman, after it has been invoked by docker-compose. The user receives a 'Bad request for API route' -error.

```
request returned Bad Request for API route and version ...
```

```
... check if the server supports the requested API version
```

```
ERROR: Service 'http' failed to build: Build failed
```

The error indicates that either the request was bad or there is an API mismatch. For this issue the discussion didn't contain any recommended steps to try and fix the issue. [S6]

As seen, there was a suggested solutions and workaround for the first issue, but not even that for the second one. The first one would require not having both Docker and Podman on the machine, which was an undesired solution for the user. Luckily both of these were identified as bugs and then fixed on release of Podman version 4. But it is an undesired situation that users expect that Docker and Podman would work seamlessly, therefore it is expected that Docker binary should work with Podman, and then there are bugs preventing smooth operation. [S5, S6]

- I2: Docker compose

User experiences an issue on Fedora 40, where trying to install docker-compose after having Podman and podman-docker already installed, resulted into a conflicting request problem [S5]. Fedora is an operating system. It contains the most modern tools available and is a crucial part in the development of Red Hat Enterprise Linux (RHEL). Linux foundation funds the OCI project and for container management Fedora uses Podman. (Callajo 2023)

Even though the discussion is on a forum dedicated for Fedora, the problems arise from docker-compose and Podman. To resolve them there was a pull request submitted for the docker-compose package, to remove the dependency on docker-cli, enabling installation of Podman and optionally podman-docker as well. For Podman there was a pull request for a bug related to not being able to use docker-compose-switch. If docker-compose wanted to be used with Podman, then user had to either use podman compose or docker compose with podman-docker installed, not docker-compose. User confirmed after waiting for a while for the updates to be published that the problem was solved and they successfully installed docker-compose. [S7]

- **I3: Registry credentials**

Third issue reported by users is related to credentials. When trying to pull an image from a private registry, compose extension is not able to access registry credentials that are configured in Podman. A workaround exists where users are able to manually pull the images after first running command

```
podman pull privateregistry.com/image:tag
```

and then

```
podman compose up
```

as then the local image cache is seeded. There is also a proposed solution that the issue might be because user has previously had an older Docker Desktop installed, and that their Docker configuration might try to get the credentials from there. Any information stored in `~/.docker/config.json` would override compose information. Even though the guess was correct in the user having an old Docker Desktop installation and being able to successfully login with podman login, the original issue prevails. The issue post contains multiple people saying that they are facing the same issue and that this is something that even entire teams are facing in their work environment. The issue seems to be quite tricky because some users do come out and say that they are not able to reproduce it, but that

the issue is possibly a Podman issue and is related to Podman's CLI. A not ideal solution was then found where the contents of

```
~/.config/containers/auth.json
```

were copied to

```
~/.docker/config.json.
```

Multiple users confirmed that this is what works for them. The issue does still remain open, even though users are able to work around it. [S8]

- **I4: Using Podman and Docker Compose**

Even though this source is not a user posted issue on a discussion forum, we can find possible caveats related to compose between Docker and Podman in here.

If swarm is used with your Docker Compose instance, it will not work with Podman. Podman does not support the swarm feature and there is no plan for implementation for it to be supported in the future, leading to a direct compatibility issue between the two technologies. [S9]

Another point that comes up is that to enable Docker Compose to work with Podman user does need to run

```
sudo systemctl start podman.socket
```

to activate RESTful API, that Docker Compose talks to [S9]. These highlight well the fact that maybe Podman is not as "Drop-in replacement" as is advertised.

6.3 Networking

Here we look at possible issues with Docker and Podman that related networking.

- **I1: Blocked internet access for Podman rootful container**

After upgrading from Fedora 40 to Fedora 41, user found out that they lost internet connection from their rootful Podman containers when Docker is installed. For rootless Podman the containers still have their internet connection. It was identified to be a Docker issue which is during the writing of this, still in effect. [S10]

The cause for this issue is that an iptables rule is added by Docker that blocks all forwarding. Docker sets a DROP policy to the forward chain of the iptables. Iptables are used to make firewall rules for containers. Firewalls and interfaces are managed by Netavark, which Podman uses for configuring networking for containers. [S10, S11] The additional iptables rule leads to rootful Podman containers to lose all connection externally. To go around this issue, users need to add iptables rule to allow traffic or revert the iptables driver. [S10, S11]

A user suggested workaround for the issue is to disable Docker completely. Of course, if Docker is needed, then this will not be possible. It was also stated that in the best-case Docker and Podman would be able to coexist with each other straight out of the box. For now, the issue is only documented, and users have to themselves navigate through it if they want to use both Docker and Podman. [S10]

- **I2: Network sharing**

User wishes to get containers in Docker's network and Podman's network to communicate with each other by name. The reason for the two networks is that the user found it difficult to port some docker containers to Podman. Also, the root-based containers were placed into the Docker's network. What user wishes to do is to find out if there is a way to treat the two detached networks as a one network by bridging them together. It would have to happen without damaging their individual configurations. [S12]

As an alternative way to go around the communication issue, user thought that forwarding to port 127.0.0.1:<port> would reach the host and connect to a correct port, but that is not the case. Using port 127.0.0.1:<port> in a container, is always "this container". To make it work, user needs to use an IP-address that is assigned to one of interfaces on the host. Referring to a host can be run by using

```
host.docker.internal
```

which is available to Podman by default for the recent versions, and for Docker if either Docker Desktop is used on Windows or MacOS or adding a host to Docker run on Linux. [S12]

The desired way of using a name resolution between the Docker and Podman is not easy to achieve. It is suggested that the easiest solution would be to use either Docker or Podman. It remains unclear if this truly is an incompatibility issue or just an issue of having two separate networks running side by side. [S12]

- I3: Default network management

There is a bug report where Podman's default network management is different from Docker. It's expected that podman-compose is able to create a default network even if an external network exists, like Docker is able to do. In compose.yml file, there is a manually created network named shared. The creation happening with a command

```
docker network create shared
```

for Docker, and for Podman respectfully

```
podman network create shared.
```

User notices that if first a command

```
docker compose up -d
```

is run, it will create a –default network, container called expose is connected to networks: default and shared, and the container called other is implicitly connected to the default network. When trying to do the same with podman-compose the results are not desired. An error of missing networks is risen. User says that this is because podman-compose doesn't create the default network and connect it to the two containers. There exists a workaround for this issue. If the default network is defined at the top-level in networks section in compose.yml, then podman-compose is able to connect the containers to the default network. Another suggested solution is that at the networks section in compose.yml contains an external: true definition. But it's still wondered that why does it differ between hosts. [S13]

Based on the last comment on the threat, the problem actually isn't what it seems by the error message. Instead, the message is misleading. When creating a network with a same name more than once, it should only work on the first time, and then after that the message should be network already exists. [S13]

6.4 Rootless vs rootful

As was discussed, Docker can nowadays run rootless, but is rootful by default. Podman on the other hand is rootless by default but capable of running rootful. With one running rootful and one rootless it's possible that issues rise, and user's need to do some extra configurations or workarounds to make things run smoothly. Let's next see what possible issues can be from the mismatch of rootful and rootless.

- **I1: Linux server image**

User runs into problems when they try to run a rootless Podman container using the LinuxServer.io Beets container. After starting the container Beets writes a config file. But, when trying to modify that file, user gets an error of the file being unmodifiable. Another issue is not having correct permissions for files, even though user is the owner for them. [S14]

The issue seems to be that many of Linux server images are made specifically for rootful Docker. It's even stated in multiple sites that there is no testing, support, or making any effort to ensuring that their images or mods would work in a rootless environment. This does mean that even though they have been built with rootful Docker in mind, they would most likely not work for rootless Docker, even as that is now also available to choose as an option. For some there is a possibility that they would still work for a rootless environment, but it's not a guarantee. [S14, S15, S16]

Luckily there is a workaround for this one as well, and users are able to modify the file they want. After modifying PUID and PGID to 0 and then changing the ownership back to 1000, user is able to modify the files they want. [S14]

6.5 Volumes

Volumes in Docker and Podman are quite similar, and they serve the same function. Like we found out in chapter 3, in Docker volumes are controlled by Dockers daemon, and for Podman volumes are controlled by Podman Volumes. Despite the similarities, there are compatibility issues found.

- **I1: Podman specific volume options**

When running Podman in rootless mode, user notices that when using the option U with docker-compose, it's not working as expected. The U option is a Podman specific volume mount. This mount option tells Podman to match the owner of the mounted volume, UID and GID, to the UID and GID inside a container (Podman Run). User is asking if it is possible to use the U option for docker-compose, or if it would be better to use podman-compose instead. User reaches permission errors, because the U option is not taken into account when running. Instead of data being created from the container it is created from the user. [S17]

Another Podman specific option Z, needed special handling by the developers to make it work in similar scenarios. It was also stated that it should not be too difficult to make the U option supported as well. Even though it was stated that support for the U option would be easy to implement for docker compose, the original poster opted to use podman-compose instead. [S17] Having these different options be only specific to one tool or the other makes it harder for the user to make their projects work if both are used.

- **I2: Mounting defaults to root**

User has configured Podman as was instructed in the S4 solution. This time the problem is that Podman keeps mounting volumes as a root. Even though user has tried to make sure in their Dockerfile that the mounting would happen as a user instead. They have explicitly defined user and group, manually assigned UID and GID to user (1000) and made sure that the permissions inside the container are correct as well as the folder permissions outside as well. The volume mount

volumes:

```
- type: bind
  source: repos/xml
  target: /patches/xml
```

still mounts as root. [S18]

There are two solutions to make the volume mount as the user in question. User can either use Podman unshare to run commands within the same user namespace as the containers or run as root inside the container. In the second option the container will still remain rootless. User opted out of the first option, stating that unshare would require changing the owner on the host system, which can cause permission issues. The second option leads to a more desired solution, the only problem is a security tradeoff. In the second solution user needs to first run as the root inside the container and then mount with either option :z or :Z. For example,

```
podman run -it --rm --name nexus2 \
  -v /home/tom/myshares/nexus2:/sonatype-work:Z \
  -u root \
  sonatype/nexus /bin/sh. [S18]
```

Both of the options change the label in container context. The lower z instructs Podman that multiple containers share the volume content, whereas the capital

Z, instructs Podman to assign a private label, restricting the access only to the current container. (Podman Run)

- **I3: Unable to write to volume**

After having experience with Docker, user decided to give Podman a chance. The problem is that user seems to be running into access and permission issues even after setting the user as 1000:1000, i.e. as user on the host system whom the container user should have access of and having the used directory inside volumes also be owned by UID 1000. [S19]

The first solution suggestion is to use the :z or :Z options. At first this is something the user does not know about, and possibly after looking into it and trying it, the outcome doesn't change. User found out that after running a command

```
chmod -R o=rwx /srv/podman/ldap
```

it starts working. The issue is that using this command doesn't restrict access like the user wants, and therefore does not like this solution. Another suggestion is to check

```
podman unshare
```

command as with-it user can change the ownership to the desired one (chown). In the end user does confirm that creating a folder at the mountpoint of subvolume and then chowning to the correct user at the end leads to succeeding in what user was trying to do. [S19]

6.6 Logging and monitoring

Logs are a very important piece of containers. They are used in ensuring the program is running correctly, monitoring user behavior, and seeing errors. [S20]

For Podman and Docker any compatibility issues faced by users were not found. But there are some areas where they are possible. As Podman supports pods and is rootless by default, some permission issues in logging are possible [S20].

6.7 Integration

As was found already in sub chapter 5.1.4 in the first issue, Docker and Podman can have integration issues, where something is made specifically one technology in mind, and then it won't work on the other. In the case in 5.1.4, the issue was thought to be that Linux server images are made specifically for rootless Docker [S14]

- **I1: Docker plugin**

After installing Podman and VSCode, enabling access to podman socket, and packages needed for compose, user tested that everything works. When user installed a Docker plugin to VSCode and updating a needed path, the VSCode extension wasn't able to connect. The issue is that the extension hasn't had Podman support for a couple of years now. Switching to use pod manager extension works instead for the user. [S21]

Even if some tools once have supported both Docker and Podman, the support can stop, and then users start facing issues.

6.8 Swarm and cluster mode

In subchapter 5.1.2 Compose, issue 4, we found out that using Docker compose with swarm, won't work with Podman, as Swarm is not supported by Podman.

There are no plans to make similar functions as swarm to Podman at the time [S22]. Also, regarding cluster mode, Podmans focus is only on single-node environments. Instead for orchestrations users could use Kubernetes' K8s yaml -files, as Podman has support for it. [S22, S23]

So, for swarm and cluster mode users can't really face compatibility issues per se, because Podman does not even have support for them. As a whole the no support is a compatibility issue in itself.

7. RESULTS

Here we will go over the results found during the multivocal research done in the chapter before this. We will tell how the results answer the research questions selected for this research.

For the research we used 23 sources, and every source was from grey literature and fell into the second grey tier. In the Figure 6 we can see how the sources divided into different source sites.

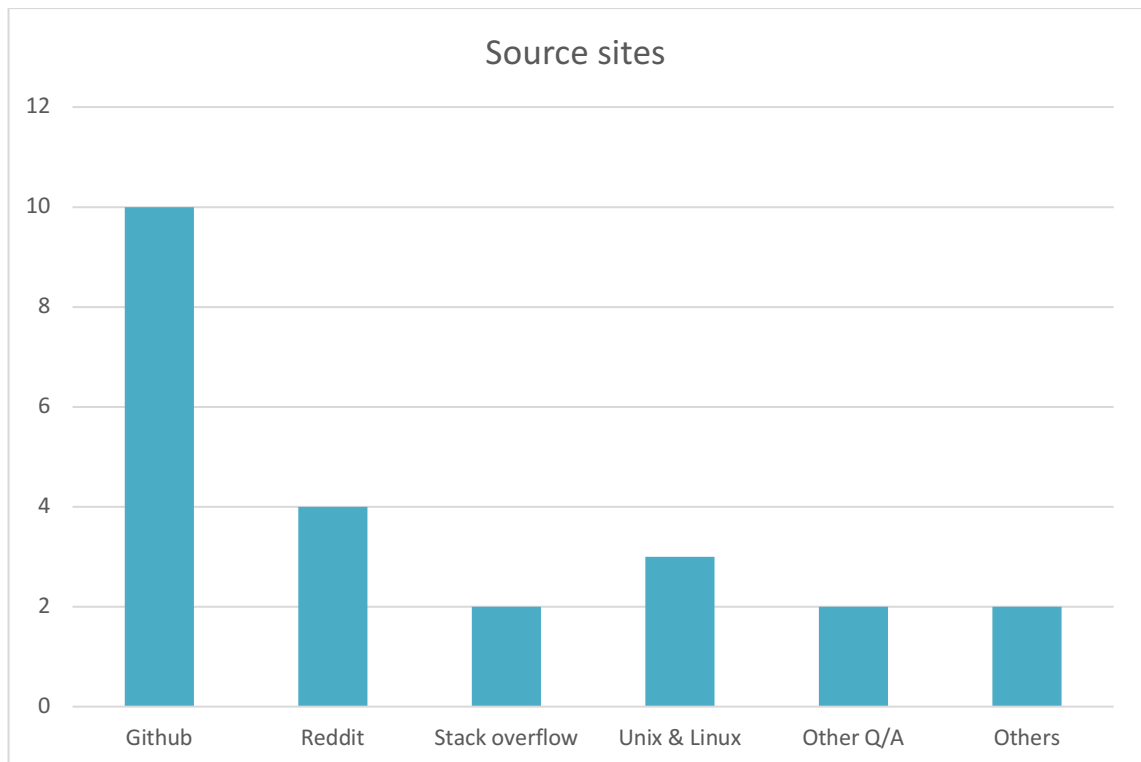


Figure 6. Source sites division

Out of the sources 21 sources were from Q/A sites, and only 2 sources used were not. Out of those 21 sources, 10 were a GitHub issue post. Second most issues were found from Reddit discussion forum.

7.1 Does Docker and Podman have compatibility issues (RQ1)

From the MLR it can be concluded that Docker and Podman do have compatibility issues between each other. The compatibility issues can be found in many different areas. The areas where there are issues are: sockets, compose, networking, rootless vs rootful,

volumes, integrations, and swarm vs cluster mode. No issues were found in logging and monitoring, but it is a possible compatibility area.

Out of the 23 sources used, seven were not an issue post, but the rest can be divided if the issue was solved, there was a workaround, or not solved. This can be seen in Figure 7 below:

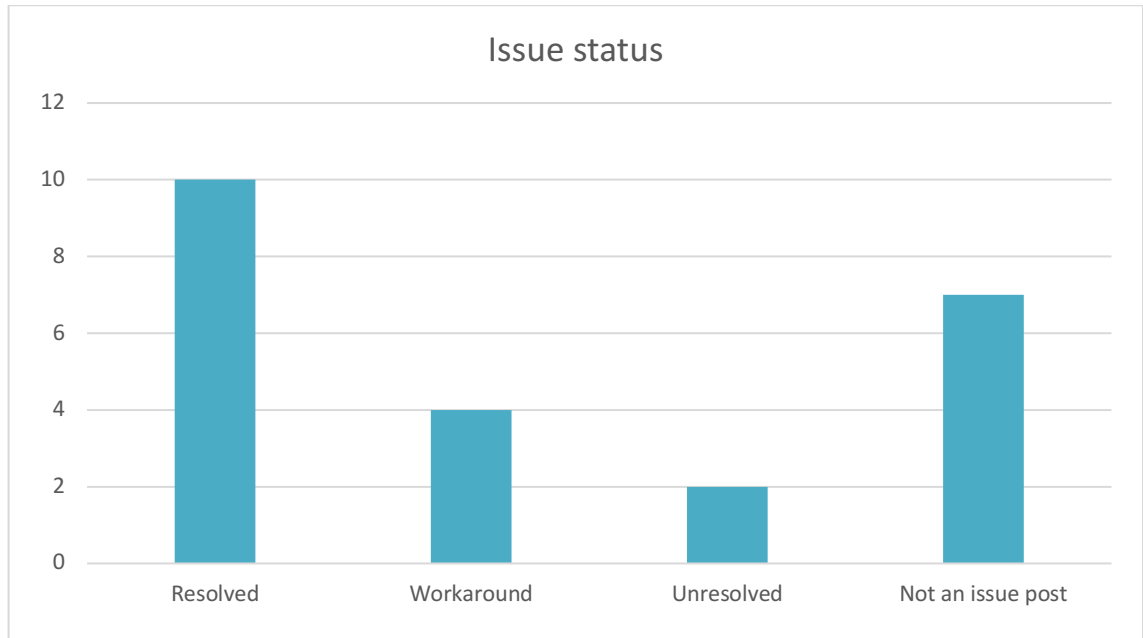


Figure 7. Issue statuses

As can be seen, most of the issues could be resolved and only two were left as unsolved. Relatively many, four, needed to have a workaround instead of a real fix.

The research also revealed that out of the 16 issue posts the problems can be divided into two. Either users have both installations for Docker and Podman, or they are trying to replace Docker with Podman. The division can be seen from the Figure 8.

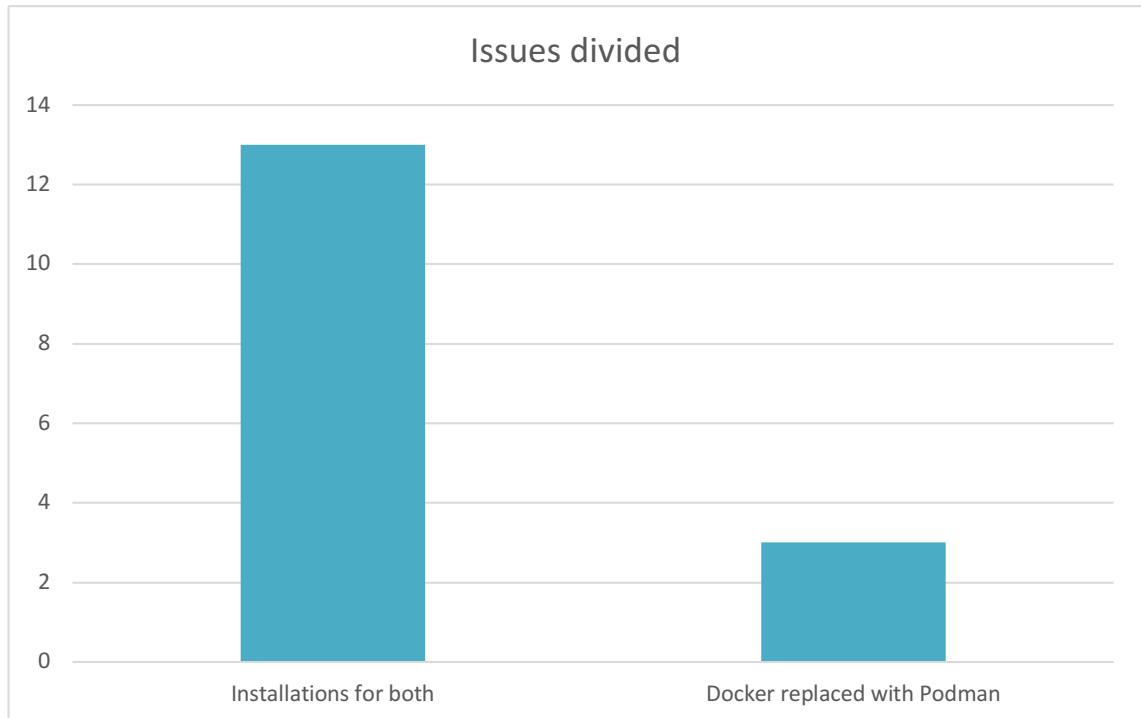


Figure 8. Issue division

The clear majority of the found issues happened when users had installations for both Docker and Podman. Only few were when users tried to replace Docker with Podman. No issues were found for trying to replace Podman with Docker.

7.2 What are the common compatibility issues between Docker and Podman (RQ2)

So, what are the compatibility issues that were found. Let's go through the different compatibility issue areas and their issues.

The first area that was looked into was daemon centric and daemonless architecture, and more specifically because of that difference; possible socket mismatches. The different locations for sockets lead to different issues. Mainly because the expectations are that they would work similarly and then they don't, it leads to bad experiences for users. Because the sockets can be set to different locations, it might not be where user expects for it to be or if there is a symlink between Docker and Podman, and the link goes from rootful to rootless socket, issues arise. Secondly when mounting a socket, Podman changes file context, Docker doesn't. These lead to unexpected results and permission errors.

Having both Docker and Podman simultaneously can lead to multiple incompatibility issues when using Docker compose. If Docker binary is installed, compose can try to de-

fault to it instead of Podman. This leads to failures when trying to interact with the Podman service. Another issue with docker-compose for users was that it couldn't be installed for Podman if podman-docker was already installed. This issue was resolved only after an update was released for Podman. Problems also were risen when trying to access registry credentials configured in Podman. This was possibly due to having previous Docker Desktop installations. Users found that there is a workaround for this, but no real solution was provided. For compose, there was one possible issue found related for Docker being replaced with Podman. Features, like Docker Swarm, which is supported by Docker compose, are not necessarily supported by Podman. Additionally, to make Docker compose work with Podman, additional steps are required.

For networking all the found compatibility issues happen when both Docker and Podman are installed on the user's machine. Again, there was multiple different possible compatibility issues found. In rootful Podman containers internet access can be lost because Docker modifies iptables rules, blocking all forwarding. Problem can be averted if user adds a corrective rule manually. The next one is a bit unclear if it's a true compatibility issue or by design. Trying to enable communication between containers in Docker and Podman networks, is not an easy task. This proved to be quite difficult to user and containers were not placed where they were wanted to. Again, it was suggested to just use either Docker or Podman instead of trying to get this to work. Lastly, Docker and Podman manage their default networks in different ways. Docker compose can automatically create and connect to default network, while Podman compose does not always do so, leading to errors. Workarounds for this included defining the default network in compose.yml-file or marking it as external.

There are not many issues in the area of rootless and rootful Docker and Podman, only one was found. This didn't stem from having both technologies installed but trying to replace Docker with Podman. Problem can be when using rootless Podman and a Linux server image. These can cause issues as many of the Linux server images are made specifically rootful Docker in mind.

Volumes can cause incompatibility issues again for having installed for both Docker and Podman as well as when replacing Docker with Podman. The incompatibility issue is technology specific volume mounts. When using rootless Podman with Docker compose, some volume mounts might not work, if they are Podman specific. In one user case option -U didn't. Another Podman specific volume mount -Z, was handled previously by developers so for the other volume mount that should be possible as well. If user used Podman compose instead, then there are no issues. Another issue is that Podman

mounts volumes as owned by the root user, even after explicitly trying to set it as rootless. This can be solved with couple of different ways, but both have tradeoffs. Additionally, when trying to replace Docker with Podman, permission issues can arise when trying to write to volumes. Even Podman specific volume mount options won't work. For this again couple of workarounds exist, user just needs to decide which is suitable for their user case.

Logging and monitoring were an area where experienced compatibility issues were not found. It was brought up in an article that due to Podman's pod support and rootlessness, some permission errors in logging are possible.

Finally, we looked at the integration area and then swarm and cluster areas. For integration it was already found in the rootful vs rootless section that Linux server images are made for rootful Docker, and possibly won't work for Podman. Another issue was that a Docker plugin that had previously supported Podman, doesn't anymore and user needed to use another one. For swarm and cluster, Podman has not and doesn't plan on adding support. Users can instead use K8s yaml files with Kubernetes, which Podman supports.

Most of the issues were something that users couldn't move forward without asking for help. That required that users had to post about them, wait for help or even a patch in a form of a new release. The issues could be divided into either built in compatibility problem areas or bugs, in figure 9.

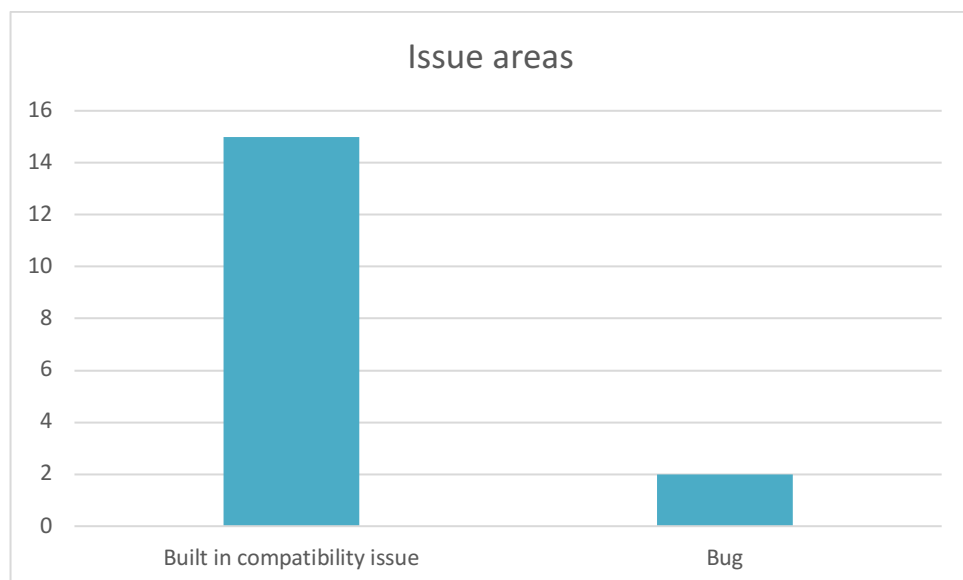


Figure 9. Issue areas

Majority of the issues fall into being built in, only two were identified as bugs.

Even though it is advertised that Podman would be a 'drop-in' replacement, it's not necessarily so black and white, as was proved. Often users felt frustration that using only

one or the other technology their programs would work without problems, but when using both, or trying to replace Docker with Podman, issues arose.

7.3 How does OCI mitigate these issues (RQ3)

In chapter 3 or 5 the scope of the specifications for OCI was determined. The issues found during the MLR fall outside of the scope of OCI. From that could be concluded that in the areas OCI does cover, the problems are mitigated well, as they don't exist. On the other hand, there are issues, so perhaps the scope of the OCI specifications isn't big enough. Though, it might be difficult to make specifications so that they would go deep into technical details. This could lead into problems due to technologies being made in different ways. Like Docker and Podman being rootful and rootless.

8. CONCLUSIONS

Using the multivocal research method this thesis found out that there are many possible compatibility issues between Docker and Podman. Compatibility issues were found in every area except logging and monitoring and swarm and cluster mode. But for logging and monitoring it was found out that compatibility issues are possible and as Podman doesn't support swarm and cluster mode, it is a compatibility issue in itself. Most of the issues happen when users have both Docker and Podman related installations, some issues if users try to switch from Docker to Podman. No issues were found for this thesis where users tried to switch from Podman to Docker.

This research also found out that the standards provided by OCI, don't cover the areas where the compatibility issues were found from. OCI provides specifications on the higher level, when the problems were most likely from lower level, related to Docker and Podman's technicalities. Based on that, in the areas the specifications are, OCI mitigates the issues well. As both are OCI compatible and therefore should be possible to use as drop-in replacement, based on the results, this isn't always the case.

Even though this research was able to provide many different areas for compatibility issues, the number of sources was quite low. This might result in not providing the complete picture. One reason for a low number of sources can either be that users don't really face that many compatibility issues, not many users in this context where compatibility issues may arise, research queries might not have been formatted well enough, to receive good results, and finally, the areas for searching was limited to 8 different possible compatibility issue areas.

The results might also not reflect the current situation. Both Docker and Podman are technologies that are constantly updated and improved on. Some of the problems that were found aren't viable as a compatibility issue today. Or there might be new ones not yet found by any of the users. To get an accurate result on the current compatibility issues, a multivocal research method is not the best option. It does however provide a clear picture that there have been and there still are compatibility issues. Most likely in the future there might be new ones. The results do also give an idea for the user of where a compatibility issue might come from.

For future research a wider approach is recommended. Now only areas were covered which were provided by an AI, but based on the results, they aligned to be in the given areas. Even if some sources provided material to other areas, they didn't fall into any

areas that weren't not provided. This would most likely require some deeper research to find more possible compatibility issue areas.

SOURCES

About the Open Container Initiative. *Open Container Initiation*. Retrieved January 9, 2025, from <https://opencontainers.org/about/overview/>

Aglave, S. (2023). Most Popular Container Runtimes. *Cloudraft*. Retrieved June 17, 2024, from <https://www.cloudraft.io/blog/container-runtimes>

Aleksic, M. (2023). Podman vs. Docker: Everything you need to know. *Phoenixnap*. Retrieved June 17, 2024, from <https://phoenixnap.com/kb/podman-vs-docker>

Arrichiello, A. & Salinetti, G. (2022). Podman for DevOps. *Packt Publishing*.

Bind mounts. *Dockerdocs*. Retrieved May 13, 2025, from <https://docs.docker.com/engine/storage/bind-mounts/>

Callejas, A. (2023). Fedora Linux System Administration: Install, Manage, and Secure Your Fedora Linux Environments. *Pact Publishing*.

Container Networking: What You Should Know. *Tigera*. Retrieved May 12, 2025, from <https://www.tigera.io/learn/guides/kubernetes-networking/container-networking/>

Containers vs VMs (virtual machines): What are the differences? *Google Cloud*. Retrieved May 14, 2025, from <https://cloud.google.com/discover/containers-vs-vm>

Dockerd, *Dockerdocs*. Retrieved March 11, 2025, from <https://docs.docker.com/reference/cli/dockerd/#daemon-socket-option>

Docker Registries. *Docstore Documentation*. Retrieved May 22, 2025, from <https://docs.dockstore.org/en/stable/advanced-topics/docker-registries.html>

Garousi, V., Felderer, M. & Mäntylä, M. V. (2019). Guidelines for including grey literature reviews in software engineering. *Information and Software Technology*, 106, 101–121.

How to manage storage with Podman volumes. (2024). *GeeksForGeeks*. How to manage storage with Podman volumes. Retrieved May 13, 2025, from <https://www.geeksforgeeks.org/manage-storage-with-podman-volumes/>

Heon, M., (2022). Podman Compose or Docker Compose: Which should you use in Podman? *Red Hat Blog*. Retrieved March 13, 2025, from <https://www.redhat.com/en/blog/podman-compose-docker-compose>

Migrate to Compose v2. *Dockerdocs*. Retrieved May 4, 2025, from <https://docs.docker.com/compose/releases/migrate/>

Open Container Initiative. (2024a). Image Format Specification. *GitHub*. Retrieved March 18, 2025, from <https://github.com/opencontainers/image-spec/blob/main/spec.md>

Open Container Initiative. (2024d). Image Layer Filesystem Changeset. *GitHub*. Retrieved March 18, 2025, from <https://github.com/opencontainers/image-spec/blob/main/layer.md>

Open Container Initiative. (2023). OCI Distribution Specification. *GitHub*. Retrieved May 18, 2025, from <https://github.com/opencontainers/distribution-spec/blob/main/README.md>

Open Container Initiative. (2024b). OCI Manifest Specification. *GitHub*. Retrieved March 18, 2025, from <https://github.com/opencontainers/image-spec/blob/main/manifest.md>

Open Container Initiative. (2025a). OCI Image Configuration. *GitHub*. Retrieved March 18, 2025, from <https://github.com/opencontainers/image-spec/blob/main/config.md>

Open Container Initiative. (2024c). OCI Image Index Specification. *GitHub*. Retrieved March 18, 2025, from <https://github.com/opencontainers/image-spec/blob/main/image-index.md>

Open Container Initiative. (2025b). Open Container Initiative Distribution Specification. *GitHub*. Retrieved March 19, 2025, from <https://github.com/opencontainers/distribution-spec/blob/main/spec.md>

Open Container Initiative. (2016). The 5 principles of Standard Containers. *GitHub*. Retrieved March 18, 2025, from <https://github.com/opencontainers/runtime-spec/blob/main/principles.md>

Pandey, M., (2025). Why Companies are Moving Away from Docker. *Aim*. Retrieved May 5, 2025, from <https://analyticsindiamag.com/ai-features/why-companies-are-moving-away-from-docker/>

Pialoux, F. How to Choose a Container Registry: The Top 9 Picks. *Bluelight*. Retrieved May 20, 2025, from <https://bluelight.co/blog/how-to-choose-a-container-registry>

Podman Network. *Podman*. Retrieved May 13, 2025, from <https://docs.podman.io/en/stable/markdown/podman-network.1.html>

Podman Run. *Podman*. Retrieved March 17, 2025, from <https://docs.podman.io/en/latest/markdown/podman-run.1.html>

Podman System Service. *Podman*. Retrieved March 11, 2025, from <https://docs.podman.io/en/latest/markdown/podman-system-service.1.html>

Volume. *Podman*. Retrieved May 13, 2025, from <https://docs.podman.io/en/v4.3/markdown/options/volume.html>

Poulton, N. (2024) Getting Started with Docker. *Birmingham; Packt Publishing, Limited*.

Powell, P. & Smalley, I. (2024). What is a container image? *IBM*. Retrieved January 8, 2025, from <https://www.ibm.com/think/topics/container-images>

Rajyashree, R., Senthikumar, M., Saravanan, G. & Sakthivel, M. (2024). An Empirical Investigation of Docker Sockets for Privilege Escalation and Defensive Strategies. *Procedia Computer Science*, 233. 5th International Conference on Innovative Data Communication Technologies and Application, 660-669.

Roach, J. (2024). Docker vs Podman: Which Containerization Tool is Right for You. *Datacamp*. Retrieved May 13, 2025, from https://www.datacamp.com/blog/docker-vs-podman?dc_referrer=https%3A%2F%2Fwww.google.com%2F

Schenker, G. N. (2018) Learn Docker: fundamentals of Docker 18.x: everything you need to know about containerizing your applications and running them in production. 1st edition. *Birmingham; Packt Publishing*.

Smalley, I. & Susnjara, S. (2024). What is containerization? *IBM*. Retrieved June 17, 2024, from <https://www.ibm.com/topics/containerization>
Storage. *Dockerdocs*. Retrieved May 13, 2025, from <https://docs.docker.com/engine/storage/>

The Rise of Containers. *Innova*. (2022). Retrieved June 17, 2024, from <https://www.innova.com.tr/en/blog/the-rise-of-containers>

Top 5 Containerization technologies in 2024. (2024). Retrieved October 5, 2024, from <https://6sense.com/tech/containerization>

Voulgaris, K., Kiourtis, A., Karabetian, A., Karamolegkos, P., Poulakis, Y., Mavrogiorgou, A., & Kyriazis, D. (2022). A Comparison of Container Systems for Machine Learning Scenarios: Docker and Podman. 2022 2nd International Conference on Computers and Automation (CompAuto), 114–118.

Walsh, D. (2023). Podman in action: secure, rootless containers for Kubernetes, microservices, and more. *Manning Publications Co*.

What is a container registry? (2022). *Redhat*. Retrieved January 8, 2025, from <https://www.redhat.com/en/topics/cloud-native-apps/what-is-a-container-registry#public-vs-private-registries>

What is Docker? *Dockerdocs*. Retrieved May 6, 2025, from <https://docs.docker.com/get-started/docker-overview/>

What is Podman? (2024). *RedHat*. Retrieved January 14, 2025, from <https://www.redhat.com/en/topics/containers/what-is-podman>

APPENDIX A: MULTIVOCAL RESEARCH SOURCES

- [S1] <https://stackoverflow.com/questions/72690495/interact-with-podman-docker-via-socket-in-redhat-9>
- [S2] https://www.reddit.com/r/podman/comments/1akij4e/mounting_podmans-ock_into_a_container_so_the/
- [S3] <https://github.com/containers/podman/issues/18480>
- [S4] <https://unix.stackexchange.com/questions/731645/podman-w-docker-compose-run-as-user>
- [S5] <https://github.com/containers/podman/issues/12206>
- [S6] <https://github.com/containers/podman/issues/16938>
- [S7] <https://discussion.fedoraproject.org/t/conflicts-when-trying-to-install-docker-compose-having-podman-and-podman-docker-already-installed/132760/14>
- [S8] <https://github.com/containers/podman/issues/22682>
- [S9] <https://www.redhat.com/en/blog/podman-docker-compose>
- [S10] <https://github.com/containers/podman/issues/24486>
- [S11] <https://fedoraproject.org/wiki/Changes/NetavarkNftablesDefault>
- [S12] <https://unix.stackexchange.com/questions/787069/podman-and-docker-sharing-a-network-and-or-hostname-resolution-between-services>
- [S13] <https://github.com/containers/podman-compose/issues/675>
- [S14] https://www.reddit.com/r/podman/comments/1db3gnp/podman_rootless_issues/
- [S15] <https://github.com/linuxserver/docker-jellyfin/issues/184#issuecomment-1382578624>
- [S16] <https://github.com/containers/podman/issues/15313#issuecomment-1312661506>
- [S17] <https://github.com/containers/podman/issues/16477>
- [S18] <https://stackoverflow.com/questions/75817076/no-matter-what-i-do-podman-is-mounting-volumes-as-root/76497033#76497033>
- [S19] https://www.reddit.com/r/podman/comments/w8nymk/container_can_not_write_to_volume/
- [S20] <https://betterstack.com/community/guides/scaling-docker/podman-logging/>
- [S21] https://www.reddit.com/r/podman/comments/1hx2cv9/podman_vscode_docker_plugin_not_working/

- [S22] <https://lists.podman.io/archives/list/podman@lists.podman.io/thread/7NN6SF5BAF4553CXHKOFHWCSQ7IQR3M3/>
- [S23] <https://github.com/containers/podman/discussions/12886>

APPENDIX B: POSSIBLE INCOMPATIBILITY AREAS BETWEEN DOCKER AND PODMAN BY AI

1. Daemon vs daemonless
2. Docker vs podman compose
3. Networking differences
4. Rootless vs rootful
5. Volume management
6. Logging and monitoring
7. Integration
8. Swarm vs cluster mode