A software framework for ROS based Robot control

by

Dinara Rysmakhanova

Submitted to the Department of Robotics and Mechatronics
in partial fulfillment of the requirements for the degree of

Master of Science in Robotics

at the

NAZARBAYEV UNIVERSITY

Apr 2025

Author .....
Department of Robotics and Mechatronics
Apr 29, 2020

Certified by .....
Zhanat Kappassov
Associate Professor
Thesis Supervisor

Accepted by .....
Yelyzaveta Arkhangelsky
Dean, School of Engineering and Digital Sciences

A software framework for ROS based Robot control

by

Dinara Rysmakhanova

Submitted to the Department of Robotics and Mechatronics
on Apr 29, 2020, in partial fulfillment of the
requirements for the degree of
Master of Science in Robotics

Abstract

Robotics education relies heavily on the ROS platform. However, due to open source nature of ROS, installation of robotic packages and libraries require knowledge of dependencies and version control. An educational framework was built by compiling and virtualizing robotic packages to facilitate learning for a novice. A review of the frameworks demonstrates that, although Virtual Machine virtualization and Docker containerization both have advantages, a shift towards containerization can be noticed. Three methods for building the framework, VM virtualization, complete framework containerization, and modular process containerization, were tested. Results show that Docker's features make it a great tool for the framework.

Thesis Supervisor: Zhanat Kappassov
Title: Associate Professor

Contents

List of Figures

List of Tables

Chapter 1

Introduction

ROS (Robot Operating System) – is an open-source toolkit for robotic systems. It plays an important role in the ecosystem of robotic control research and industry development. With the continued integration of already commercially available robots and still developing prototype models, ROS is essential for robotics education curriculum. However, in an open-source environment, parts of the same robotic systems were developed by different researchers. That leads to certain errors due to but not limited to conflicting requirements, versions, and dependencies. Therefore, learning ROS may seem intimidating in the beginning. For students and people new to the robotics software field, the issues and errors arising from simple beginner tasks could hinder the learning progress. Creating an isolated executable program to facilitate the learning process is required. Tools such as virtualization are used to streamline the complicated setup.

Virtualization, creating a ready-to-use operational system in addition to the computer's main operating system (OS), is the current tool used in classrooms and educational environments. Docker containerization is a newer tool that evolved within virtualization. It achieves similar goals with less resource utilization. Docker as a framework for Information Technology Education is investigated and shows great potential [3]. Along with virtual machines, container-based software applications are gaining momentum in educational and commercial spaces. The knowledge of container-based frameworks can be later translated into research and industrial ap-

plications. Research on containerization demonstrates several implementations of the framework for ease of use for beginners as in [2], or more complex solutions for reproducibility in industrial environments [4].

Two containerization methods, monolithic frameworks or modular containers, are used in state-of-the-art frameworks. The first method collects all of the libraries and correct dependencies and integrates them into a single stand-alone software. A container provides a consistent environment throughout systems, but the image needs to be rebuilt for each change. The second method requires a specific environment. Within the environment, modular containers consisting of one or more ROS Nodes will be deployed. Manual version management will complicate the framework, but the system will have lower latency essential for real-time communication with robotic hardware. The method performance also depends on the application area and the setup complexity.

A containerization of a simple setup for coursework in robotics was attempted. A snake robot consisting of five Dynamixel Motors actuates the robot in a rotational manner, while connectors between motors simulate snake-like movement. Complete control of the robot requires Dynamixel package in addition to core control libraries, such as motion planning, visualization, simulation real-time communication. Several applications need to run simultaneously. RViz provides manual motion control through Visualizer and Motion Planner panels. Gazebo Simulator can create a simulated environment that can visualize the robot independently or along with the physical robot.

Chapter 2

Literature Review

Robotics framework using ROS requires a good understanding of the ROS environment. It is essential for a user to understand package libraries and dependencies and their connection to the operating system to create a working framework. Starting from the installation of ROS to collecting all the required versions of dependencies and debugging errors in the process could be difficult for a person new to ROS and robotic systems. Inconsistencies between environment and dependencies are one of the major issues in the ROS framework [5], [6], [7]. For a beginner learning environment, creating a fully functional framework and isolating it from the system using virtualization can be a good solution. Virtualization technology, including a newer subtype: containerization, has been used in the educational and research areas for some time. However, ROS-based frameworks are new and still in the research and development phase. Therefore, in this work, I will be reviewing literature on robotics and computer science frameworks, works comparing virtualization technologies, and analyzing the frameworks and their components for efficiency in resource utilization and ease of use.

2.1 Basic components of the ROS framework

2.1.1 ROS

S ROS contains numerous libraries and packages required for most robotic systems, and compilation and deployment of the robot framework is challenging due to its open-source nature. Challenges include complexity in integrating multiple components within the system [8], [9], [10], [11], inconsistencies in the version of packages and their dependencies [5], [6], [11], and portability between devices with different environments [6], [8], [9]. Therefore, understanding the problem and debugging takes up most of the study and research time. Thus hindering hands-on experience in learning control algorithms and integrating sensors and additional hardware for physical robots [10]. The best way to facilitate learning and research is the compilation of packages and deployment of the software within an isolated virtual environment.

Methods of ROS virtualization vary depending on the application. The base architecture of ROS consists of nodes and communication between them. Every function or process, like model creation, movement, and visualization, is a separate ROS Node. Communication between nodes is carried out by ROS Topics and Services. That creates two viable ways to virtualize ROS. In the first method, we create a virtual environment of a whole OS with a preinstalled ROS inside it [2], [5], [7]. Virtual OS can be modified to suit the needs by adding necessary libraries and packages. As a result, a uniform application can be deployed on multiple machines. The second method is to build a container for each node/process [8], [9]. The software will have increased modularity for varying tasks. The application of the software framework determines which virtualization tool is the most suitable for the task.

2.1.2 Virtualization and Containerization

There is an expected trade-off between performance and convenience when using virtualization technology as opposed to programming on the host OS. Virtualization is slower than running the application directly, especially in real-time or network envi-

ronments, but the differences for simpler applications are small [12]. The advantages of the streamlined learning experience outweigh the delayed response.

The most prominent tools for virtualization are Virtual Machines (VMs) and Containers, with VMs having been used in IT education for some time [3], [13]. Virtualization using VMs creates an isolated hardware subsystem on top of the computer as depicted in 2-1, securing the system OS from the virtual environment, thus providing better isolation and stronger security [1], [14]. VM can provide virtualization of the Linux OS with installed ROS. The required software packages need to be collected, verified for dependency issues, and installed on the system during the setup stage. The virtual environment is then deployed with VMware, which varies depending on the system OS [1]. VM is a reliable and conceptually straightforward way of virtualization for the whole framework.

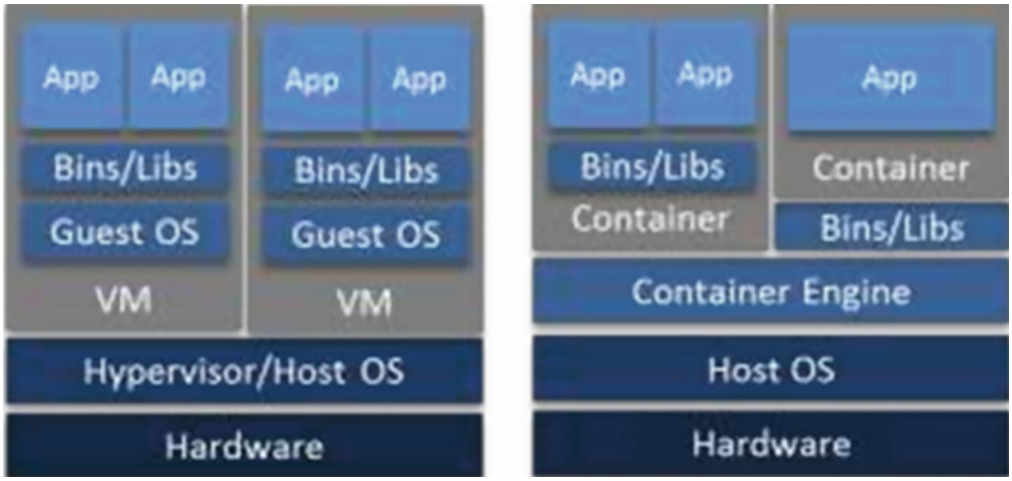

Figure 2-1: VM versus Container architecture. Adapted from [1]

Containerization using Docker, a newer type of virtualization technology that is gaining popularity due to more efficient resource utilization, can virtualize separate nodes/processes in addition to the whole Linux OS with ROS as shown in the right column of 2-1. In the first case, Docker provides an official image for all ROS distributions that can be set up for specific needs [6]. The virtual, ready-to-use container OS can then be deployed on any system. The virtual OS will share the host OS kernel, providing faster deployment and lighter weight, but compromising on less isolation [1], [14]. In the second case, separate packages or nodes can be containerized

for additional modularity [8], [9]. Although that would require manual dependency management, existing frameworks can be improved and updated faster and more easily. This approach can further improve the framework or a stand-alone system for advanced users in the fast-paced research and development field.

2.2 Containers vs Virtual Machines

VMs and containers have different useful features because they virtualize the system on different scales. VMs virtualize on the hardware level, providing a stable and secure virtual environment for the complete software framework while compromising on resource efficiency [1], [15]. Containers can virtualize on both the process level and the OS level. OS-level containerization creates a virtual guest OS by sharing the kernel with the host OS for increased resource efficiency and decreased security [1]. Process-level containerization packages independent applications or processes separately [16]. Along with having better resource utilization, process containers provide modularity for faster upgrades of complex systems [8], [9]. The drawback is that the consistency in dependencies and versioning has to be verified manually or otherwise. Experienced researchers can adapt to the increased complexity of the system and utilize the modularity and speed of process-level containerization, but it would not be as simple for a beginner to resolve and integrate conflicting configurations.

Virtualization of the whole system OS can facilitate learning for a student with zero to little prior experience working with software development. A complete virtual environment with pre-installed ROS and pre-compiled packages would allow students to focus on applying theoretical knowledge while familiarizing them with ROS software controls [5], [10]. Both VMs and Containers provide full system isolation and packaging.

Performance comparison results between Virtualization and Containerization can provide a more fitting option for an education framework. Performance in resource utilization, portability, scalability, and security will be compared.

2.2.1 Resource management

Physical resource utilization is higher for VMs due to the isolation of hardware resources. The boot time of VMs can take up to a minute, while containers boot within 5-10 seconds [1], [7]. Docker images are lighter, with 1.2 GB compared to 13 GB of the VM image for the same Oracle software [3]. CPU and RAM usage are higher for VMs [1]. Compared to the non-virtualized environment, VM, namely KVM, has 27.5% to 55.3% CPU performance overhead for the CPU-intensive tasks, while Docker stays with a 0.36% to 8.02% range, respectively [16]. The CPU and RAM performance of Docker is also better when tested by using the same Ubuntu 20.04 system [17].

The resource utilization also depends on the scale of the task. [18] compares Docker and VM in three different scales: PaaS (Platform as a Service), SaaS (Software as a Service), and IaaS (Infrastructure as a Service). On bigger scales, PaaS and SaaS, Docker and VM perform similarly, with slightly better performance of VM virtualization in demanding and long tasks. VM also shows better results with smaller, more frequent processes within the long task [19]. However, SaaS Docker outperforms VMs with its response time two times faster [18]. Liu proposes a system with FaaS (Function as a Service), granular functions and processes, that further reduces resource utilization for edge devices [20]. Although not all Docker containers support ARM architecture inherently, their power consumption is reduced compared to x86 architecture [21]. The efficiency of containers adjusts to lower-performance devices. Subramaniam et al. successfully deployed containers on Raspberry Pi for a Pololu Romi 32U4 robot [22]. Docker containerization is less resource-intensive and lighter. It is better suited for the low-performance devices that students or less advanced classrooms may have.

A review of latency for real-time operations shows that containerization performs similarly to the bare metal configuration. Average latency in industrial automation was 74 µs for bare metal and slightly slower 117 µs for a containerized system [23]. In an educational space, the difference in µseconds is negligible. [24] reports that Docker containers outperformed a non-containerized system due to optimized resource

usage. Kubernetes can further reduce latency by dividing the containers into pods [25]. Although the lowest latency is not needed for most educational robotics, robots interacting with humans or manipulators performing pickup-and-place tasks for fragile objects can greatly benefit from the methods above. Using the Faas architecture, modular containers ensure that the latency of the systems stays low [20].

Modular architecture with multiple containers necessitates a good network connection between containers for low latency. A bridged isolated network is mostly used for containers. The host network can be used to connect with networks outside of the container, but it reduces the isolation and security that the containerization provides. A scalable cloud computing solution is proposed for Kafka ROS communication with Docker to decrease latency overhead [26]. Additionally, a proxy network is proposed to further improve the latency in a complex containerized and non-containerized system within a host with modular containers [27]. Increasing the complexity of the system increases the latency, but solutions are being developed to minimize its effects.

2.2.2 Scalability

Containerization is considered more portable than VMs due to the varying scale of containers and uniformity in middleware. Docker is available for all platforms, while different VMware is available for varying host OS' [1]. Docker also provides flexibility in modularity by containerizing applications [14]. Containers can run with Docker homogeneously, and many VMware for any system are available.

Containerization allows the generation of multiple containers without increasing physical resource overhead considerably. Setups with multiple robots and computers would require network connectivity for communication. Kubernetes is an orchestration tool to automate and manage deployment of multiple containers across multiple hosts. Industrial field software has been adopting the Kubernetes setup [28]. It increases scalability for complex models while allowing fault tolerance with pod redistribution [29]. Pods, a unit with one or more containers, can be used for clustering containers with similar applications. To take full advantage of Docker's uniform environment, Kubernetes-in-Docker architecture was used to cluster containers in a single

container [29], [30]. Node container clustering is simplified with a fixed environment of the Docker container in KinD. Multiple containers also enable dual booting of different systems such as ROS and ROS2 [31]. Running multiple containers on both smaller scale, e.g. ROS nodes and process, and larger scale, e.g. multiple hosts, is advantageous for intricate robotic systems.

Automation of container generation is essential for complex architectures. Containers can be automatically created on trigger events. For instance, Cervera proposes rebuilding the docker image when a code update is triggered by a GitHub push command [32]. Version control could be simplified for continuous development. Docker-ros tool was made to automate building the docker creation in [33]. Systems with similar environments can be simplified with automated containers. Several functionalities for swarm robots can be organized neatly within container units.

2.2.3 Security

VMs have a better security stance than containers. User privilege attacks are possible in containers because they share the host OS kernel, escalating to container escape attacks [15]. Although VMs can also be affected by VM escape attacks, safety is higher due to isolation at the hardware level [14], [15]. In an industrial and corporate field, where hijacking is possible, along with the research and development field, where privacy is required, the VM could be a better option. However, in a low security risk environment, such as a classroom, VM's security feature doesn't play a big role.

In conclusion, Docker containerization is the most appropriate tool for a non-complex educational framework. VMs higher resource utilization makes it less desirable for devices with unspecified resources [9], [13]. Security is lower than resource utilization on the priority list in the educational environment, making VMs less convenient for the virtualization of the educational software framework.

2.3 State-of-the-art frameworks

2.3.1 Components

Educational software frameworks are mostly containerized with pre-determined components so that they can run as a stand-alone executable application [2], [5], [34], [35]. The image is assembled after collecting and compiling all the necessary components, [5] uses the YAML file to specify the necessary components and their versions to automate the process. The image is then considered final, and all the changes are done by rebuilding the whole image. Changes can be monitored with the version control system, with some images prebuilt for specific scenarios [34]. The task-specific images can be used across different application areas, including isolated laboratory experiments. Images can also be divided into layers to include non-essential, rare functions in separate images without increasing the complexity of the framework as in [2]. The stand-alone image provides simplicity in utilization but increases the latency in communication with other devices.

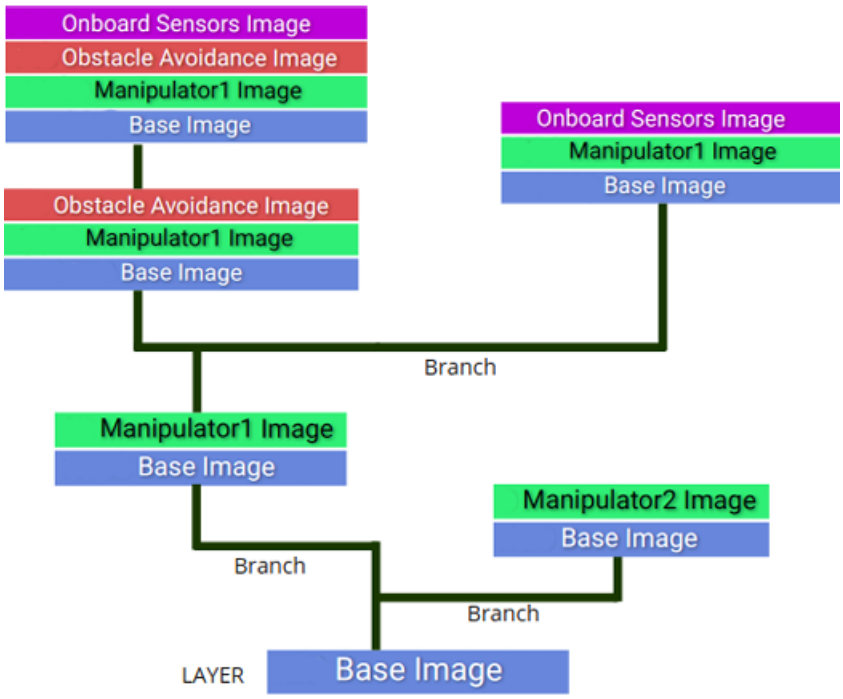

Figure 2-2: Layered container images architecture of P&P framework. Adapted from [2]

Containerizing ROS Nodes allows faster communication between nodes and various devices [8], [9]. In a multi-device setup with robots and simulated environments, Kubernetes, a container orchestration tool, reduces network latency preventing network bottleneck [9]. The method is useful for complex robotic system applications in research [9], or cloud computing where network communication is essential [8]. It's also applied in education with complex images, where Docker Compose runs four containers with a single command [36]. Although low latency for real-time communication is great for controlling a physical robot, a buffer can be acceptable within laboratory conditions.

Docker-in-Docker (DinD) allows a Docker container to be run within a Docker container. Several task-specific containers can be run in an educational framework presented in [35] and prototyping application presented in [2]. Complex systems can be reduced to simpler tasks, the framework can be upgraded with temporary components. DinD adds a layer of functionality expanding the applications of containers.

2.3.2 Application areas and Results of deployment and testing

Containerized frameworks have different useful features for varying application areas. Education applications in IT [3], [13], [35], [36] and Robotics [5], [9], [10], [11], [37] containerize the software for easier learning. Research fields deploy containerized applications for faster development [2], [7], [8]. Industrial robotics use containers to automate and streamline complex control systems [4].

Containerized, ready-to-use educational frameworks were developed in [5], [10], [11], [37] as shown in Table 1. Components of the application were collected and compiled before containerization. [10] and [37] developed learning platforms with exercises and tutorials, while [5] and [11] provided a virtualized environment for a more ambiguous learning path. A combination of both could help students navigate learning basic ROS controls and work on creative solutions for more complex projects independently. Layered images proposed by [2] incorporate both methods by using more specific images for tutorial tasks and general images for a broader range of capabilities required for exploration.

Table 2.1: Methods and results from related works

Source	**Containerization scale**	**Tested robots**	**Proposed Method**
Cervera and Del Pobil [2]	Full Stack	Simulation	Using JupyterLab to save documentation along with process execution code
Indri, Sibona and Russo [3]	Full Stack with layers	EEZYbotARM MK2 robot arm	Layering container images for separate applications
Aldegheri et al. [4]	Modular	Robotnik Kairos mobile robot	HW-in-the-loop verification of real-time constraints through network communication of Node containers
Lumpp et al. [12]	Modular	Robotnik Kairos mobile robot	Clustering nodes into containers
Melo, Arrais and Veiga [14]	Modular	UR10 robot arm	Automating Docker image generation with a multi-stage pipeline
Cañas et al. [15]	Full Stack	TurtleBot	A platform for learning robotics with gamified exercises
El Hafi et al. [17]	Full Stack	Simulation	Software Development Environment (SDE) for collaborative work
Erdogmus and Yayan [18]	Full Stack	Multiple mobile robots	Uplat platform for learning with exercises for provided mobile robots

ROS Nodes containerization has better resource utilization providing better performance in real-time robot control. [9] further optimizes the framework by clustering nodes for specific applications. Hardware-in-the-loop, verification of physical constraints in real-time communication with robot hardware, proposed by [8] can add safety to physical robot experiments. Laboratory with consistent computer hardware and environment can benefit from additional optimized functionality despite being more difficult to manage.

2.4 Conclusion

The review of virtualization methods, particularly the containerization of ROS, establishes that the containerization of the complete ROS with precompiled packages is the best method for the educational framework. The containerized framework will provide a uniform environment across all possible devices with reliable results. Additionally,

ROS Node containerization can expand and improve the framework without increasing complexity disproportionately. The analysis of several existing and proposed frameworks reveals inconsistency in framework components and that the methods of containerizing the frameworks vary depending on the resulting application. The best components and containerization method for an educational framework for a robotics course are yet to be determined.

Chapter 3

Framework Methodology

An educational framework for a robotics course is needed to facilitate the learning experience for beginners. Working with ROS, the most common platform for robotics tools, requires knowledge of dependencies and version management. The optimal method to assemble and isolate the software framework is virtualization. A virtualized framework with proper documentation could help ease the newer students into the robotics field.

Analysis of the current solutions in virtualization technologies provides a variety of methods to build the educational framework. These methods have benefits and flaws depending on the application area, software environment, and task scope. For the robotics course, a more thorough comparison of performance and ease of use is carried out. For this purpose, a framework was virtualized with both VMware and Docker for a practical comparison. After identifying the best method, a framework is constructed for the snake planar robot used in the robotics course.

3.1 Framework design

The framework is intended for teaching a robotics course for beginners, therefore, necessary design constraints must be applied. It must be easy to install and use with simple instructions that would be easy to follow for inexperienced users. Resource utilization must be low so that the framework can run on an average personal computer.

Although the highest efficiency with immediate response time and high graphics is not necessary for learning, performance needs to be good enough to motivate the students and streamline the process.

The framework architecture starts with ROS at the base. It contains all the dependencies and libraries for the framework, including the ones for the robot model and its motors' packages. The snake planar robot model and its control configurations define the robot system used in the course. Motion control is executed with the MoveIt motion planner. It can be done either in CLI or interactively as an RViz plugin. Additionally, Gazebo for simulation is added. After all components of the robot software are collected and verified for dependencies, ROS is virtualized with VM and Docker to compare performance.

The final framework is a virtualized image. In the case of VM virtualization, VMware is used to work with the framework. After installation of VMware, an image of the framework can be downloaded and opened on the user's computer. An image will open the Ubuntu OS desktop with all the necessary applications and packages preinstalled. ROS can then be run as usual through a terminal. For containerization, Docker Desktop runs the image. The GUI required for RViz and Gazebo is not native to Docker, thus, the image is modified to include the GUI. The user can then run the image in either CLI or Docker Desktop itself to open the framework in an isolated environment in the form of a container. Documentation on the installation of VMware or Docker and the employment of the image will be provided in both cases to improve the user experience.

3.2 Design implementation

The development process of the framework started from the outermost layer, establishing virtualization tools. It was followed by adding the layers in the following order: Ubuntu OS (for VM), ROS, GUI, planar robot packages, and its test cases. Adding components to the framework required debugging and dependency management, so the framework was tested and verified after each layer.

VM virtualization setup is reasonably straightforward as seen in 3-1. There are several Hypervisors available for building and using virtual machines. VMware, VirtualBox, and Hyper-V are among the most well-known ones. VMware and VirtualBox are type 2 hypervisors that run on top of an OS, instead of hardware [1]. The demand for the education robotics framework is not high, so a hypervisor that is easier to use and set up was chosen. Between the two, Wazan et al. demonstrate that VMware outperforms VirtualBox [15]. A free version of VMware is available for both Windows and MacOS platforms. After installing the VMware Workstation Pro, the official Ubuntu 18.04 OS image was downloaded. Subsequent steps are similar to using ROS on Ubuntu natively. In the provided virtual desktop environment, ROS Melodic was installed and set up with instructions from the official site. ROS packages for snake robot are provided in Github, and were downloaded with git clone. After installing all additional packages such as Dynamixel motors and MoveIt, the framework was ready for the testing process.

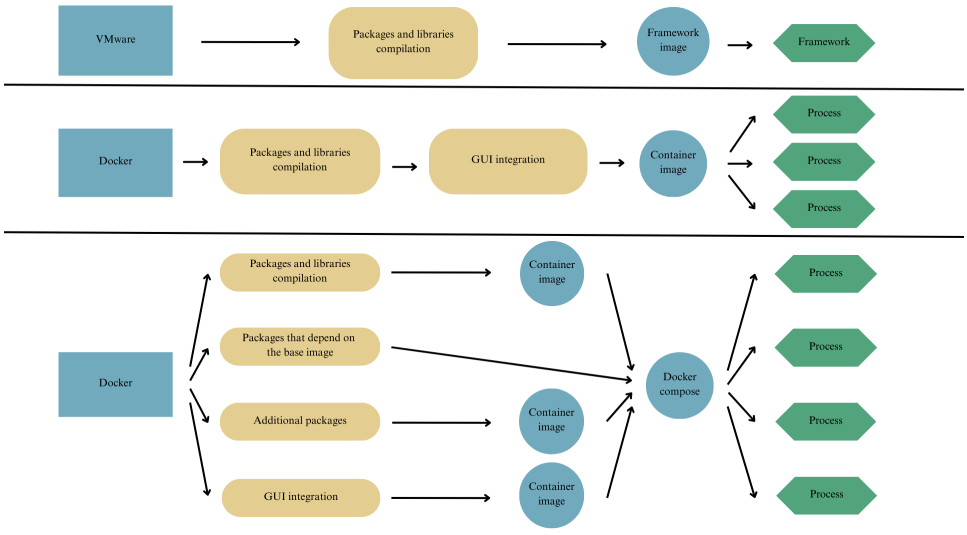

Figure 3-1: Virtualization process pipeline

Docker containerization, on the other hand, requires knowledge of Docker CLI [13]. Docker Desktop, a GUI that uses Docker Engine to virtualize, provides a GUI for image and container management. After installing the Docker Desktop application, ROS Melodic was pulled from the integrated Docker Hub repository. Two frameworks

were built using the image to test functionality performance and setup difficulty.

All framework components were collected into one image in the first case. All the necessary packages and libraries were installed when building the image. Dockerfile was created to create build instructions. The image is then built and run with docker and a single container was created. The single container architecture was easy to build and use but the changes required rebuilding the whole image.

In the second case, the framework components were also collected into a base image. However, the third method breakdown in 3-1 shows that this method provides more flexibility. The base image was used to run ROS nodes in different containers. As an example, roscore, which runs the ROS Master required for all ROS communication, was set up as a service in docker-compose and it launches automatically when the framework is run. Other ROS nodes were run in their separate containers and refer to the roscore container for communication. This allows framework scaling by adding the packages to the base image without having to rebuild it. Additional modifications can also be added with their own images for isolation so that the changes made in these images don't affect the base image. Modules can be added and removed depending on the functionality requirement. The GUI was separated and the noVNC image from Docker Hub was used. Communication between containers was set up by creating a docker network. All module and network instructions were written in the docker-compose.yaml file.

3.3 Framework implementation

3.3.1 Framework tools and components

Although Windows, the host OS, has a built-in application for virtualization Hyper-V, performance comparison shows slightly better results with VMware Workstation [17]. Therefore, VMware Workstation Pro 17, which is free for non-commercial use, was used to deploy the image for VM virtualization. For containerization, Docker Desktop for visual image and container management and Windows Powershell for

CLI were used. ROS is not native to Windows and requires Ubuntu, so Docker used WSL2 (Windows Subsystem for Linux) to run containers that require Linux.

ROS Melodic-desktop-full version was used in both cases. It comes with the RViz and Gazebo preinstalled for visualization and simulation respectively. The snake planar robot package, ros_snake_robot [38], was downloaded from GitHub. The project is in C++, so the test files for the robot were also written in it. Additionally, noVNC GUI image for Docker was used.

3.3.2 Dockerfile

Docker builds the image using instructions from Dockerfile. The ROS with snake planar robot image is built with the following commands. The osrf/ros:melodic-desktop-full-bionic image is used as a base and the dependencies that can be directly downloaded within ROS were installed first. The workspace was then set up for the installation of dependencies and the ros_snake_robot package from github. A simplified sample of the Dockerfile is provided below.

```
# Base image
FROM osrf/ros:melodic-desktop-full-bionic

# Install dependencies
RUN apt-get update && apt-get install -y \
        ros-melodic-ros-control \
        ros-melodic-ros-controllers \
        ros-melodic-moveit \
        ros-melodic-dynamixel-sdk

# Set up and initialize the workspace
RUN mkdir ~/ws/src
WORKDIR ~/ws/src
RUN source  /opt/ros/melodic/setup.bash && \
```

```
    catkin_init_workspace ~/ws/src && \
    cd ~/ws && \
    catkin_make

# Install Snake Robot Package
RUN cd ~/ws/src && \
    git clone https://github.com/fenixkz/ros_snake_robot.git
      && \
    source devel/setup.bash
```

3.3.3 Docker Compose

Building a more complicated container setup as shown in the third pipeline in 3-1 requires an orchestration of containers. Docker Compose creates services specified in yaml configuration file. The image from the previous section was used as a main container image for ROS. Roscore was run as an individual container so that all separate ROS node containers could connect to ROS Master. With the environment of roscore specified, the ros_node service could run ROS node containers that can communicate through one ROS Master. Volume mount could be defined if changes in the ros_node service need to be saved. Novnc GUI with its base images is run as a service and the ros containers that require GUI could access it through exposed ports by adding depends-on command. A Docker network was used for communication between containers and was specified for all services.

```
services:
# First service. Runs the ROS Master
  roscore:
    image: ros_snake_robot:latest
    container_name: roscore
    networks:
      - ros
```

```
    command: ["roscore"]

# ROS Node service that can run a GUI with noVNC
# The last two lines are to make the container interactive
  ros_node:
    image: ros_snake_robot:latest
    container_name: ros_container
    environment:
      - DISPLAY=novnc:0.0
      - ROS_MASTER_URI=http://roscore:11311
    depends_on:
      - novnc
    volumes:
      - ./catkin_ws:/root/catkin_ws
    networks:
      - ros
    stdin_open: true
    tty: true

# A sample of a noVNC web display
  novnc:
    image: theasp/novnc:latest
    container_name: vnc_container
    environment:
      - DISPLAY_WIDTH=1600
      - DISPLAY_HEIGHT=968
      - RUN_XTERM=no
    ports:
      - "8080:8080"
    networks:
```

— ros

The YAML file demonstrates the example of the services that can be created with docker compose.

3.4 Evaluation

Frameworks were compared on the criteria of convenience, setup time, run speed, and software weight. The possibility of modular integration, reproducibility on different platforms, and efficiency of the framework versus time and effort constraints were also considered.

Three frameworks were tested as shown in 3-1. VM is a straightforward way to virtualize a system as a whole and has been used in education for a while [4]. It was used as a benchmark for the convenience of setup and utilization. Docker's learning curve for both single and multiple container architecture was compared to VM.

The completed frameworks will run two tasks with several processes and GUI applications. The first task will integrate RViz's interactive motion control for the simulation. The second task run a simple coded movement of a simulated planar robot in Gazebo to get a baseline performance comparison. CPU and RAM usage measurements will be compared for three frameworks. Real-time motion control will provide a comparison for more complex scenarios, while coded movement will provide scalable results for identical scenario.

Chapter 4

Framework performance

4.1 Framework setup

Setting up the framework was the major part of the thesis work. VM setup was straightforward, it was an easier way to set up an additional OS on the host OS. The process included downloading VMware and the Ubuntu image. After configuring image resources, 16 GB RAM and 8 CPU processors, a virtual environment was created. Within the virtual OS, ROS and the packages for the planar robot were installed. VM didn't require additional knowledge beyond installation instructions.

Docker framework setup, on the other hand, required learning some docker syntax. Windows PowerShell was used to create and manage images. For the single container architecture, a Dockerfile syntax was used. Basic steps for the framework included choosing the appropriate image for the base, creating and initializing the workspace, and downloading and installing the ROS packages. Additionally, volume can be mounted to save the progress from the working container if necessary. Most importantly, a GUI for displaying ROS applications was added. As the image was created and tested on Windows, the WSL display was chosen for the framework due to latency. However, additional GUI options, such as noVNC, were configured and can be provided for users with different operating systems. The multiple container setup involved three more steps. First, we divided the processes into separate containers. A network between containers was created and set up in the next step. Lastly, a YAML

configuration file for Docker Compose was created. Most of the variable requirements were fulfilled by the single container architecture setup, so the complexity of the framework setup was increased mostly by network creation and service declaration. Setting up the Docker image is also more difficult due to the lack of a user interface for image setup, as Docker Desktop does not explicitly support image creation.

4.1.1 Scalabilty

One of the motives for the framework is an easy, reproducible setup. Although the work is tested on one robot, the possibility of building a framework for any robotic system is important.

VM's setup process consists of common, familiar steps of ROS robotic system installation. Each VM is a fixed framework, and any modifications made to the framework require a redistribution of the image. Considering that the image size for the ROS snake robot used in this work is 14.4 GB, distributing the image frequently to students involves downloading over 14 GB image each time. Therefore, adding a module or even a simple task to the framework becomes an arduous task. Stacking the images for different purposes would increase the needed space, which confirms the scalability issues in [17].

Docker containerization has a learning curve [13], especially in a multi-container environment. Building the image consists of listing all the steps of the installation process in a step-by-step manner in a configuration file. Rebuilding the image on the same host is faster because only the changes are processed. In other words, if the previous image is present, the image is upgraded to a newer version with modifications instead of creating a new one each time. Correspondingly, students can update their images from the repository. The framework weight, which is 4 GB for the planar robot, is maintained at a similar level. Moreover, additional isolated containers from the same image can be created to work on several separate tasks for the robot in the same environment [6]. It is also possible to boot multiple robots in different environments, as suggested by [31]. Stacking multiple robots is lighter, and a network between them could be created for robots to communicate. Docker is efficient for

simple setups and essential for complex ones.

Multi-constainer architecture has a similar setup process. Modularity of the method adds more functionality to the framework. For educational purposes, it can be used to build an immutable application where all of the processes are started for the user. Listing of the Docker compose configuration code demonstrates automating ROS Master creation. All processes and tasks can be defined as services. Applications for the immutable framework can vary from tasks for absolute beginners to the automation of industrial robots. Modularity of the architecture can also be used in rapid research and development. The Docker framework for rapid prototyping was also validated in [39]. New modules for peripheral devices like sensors or additional software like a GUI can easily be added to the container network. Scalability of microservice architecture from [24] corroborates the efficiency of the modular structure for the framework.

4.1.2 Framework utilization

Docker Desktop facilitates utilization and management. A container image can be easily shared with Docker Hub, and containers can be run from the Docker Desktop intuitively. Users can work with the existing tasks or create their own in the same way as they would with a native installation through a terminal.

Containerization does not provide a user interface for interaction with the image [3]. User experience could be worsened due to the lack of a user-friendly interface. However, after adapting to CLI, the framework utilization is streamlined by reducing the GUI load times. Working directly with the terminal can eventually help to adapt to ROS faster.

VM, on the other hand, provides a familiar interface. The desktop interface of Ubuntu is convenient for a novice because you can explore the system explicitly. VM provides the same process of working on the project as the bare metal configuration. The frameworks simplify the installation without interfering with the learning process.

4.2 Performance results

Three frameworks were tested for performance measurements. The metrics for comparison were startup time, CPU and RAM usage, and frame rate.

The startup time of a VM image was 23 seconds, and loading and opening a terminal took an additional 11 seconds. In total, to start working with ROS, VM took 34 seconds. Docker foregoes the need to load a graphical interface and opens a bash in 2 seconds from a CLI. The boot-up time is the same as the values in [1] and [7]. VM is slower because it loads a full desktop interface for the framework. Docker focuses on running the process, so it executes commands faster. The container's environment is loaded from WSL. Running containers is faster, but there is a boot-up time of 15 seconds for the Docker Desktop with its engine. Although the time to start the framework is close, repeated opening is much faster for Docker containerization. The difference is negligible for an educational framework, but in research and industry fields, Docker is better adapted for faster work processes.

The only GUI used in Docker was for the ROS visualization and simulation applications, and their frame rate was restricted only by the applications themselves. VM, on the other hand, loads an Ubuntu desktop interface, and the frame rate for the VM averaged at 13.07 for the interactive task. Docker provides a smoother GUI experience with the help of the WSL, limited to the Windows platform on which the frameworks were tested. It displays the GUI through the X server installed with WSL. It limits cross-platform compatibility. The solution would be to install the X server on the user device, which complicates the framework's utilization, or to use a custom GUI container available in the Docker repository. The noVNC container was used in a multi-container architecture to test its responsiveness. The GUI is opened in the browser and shows a similar frame rate to the VM.

4.2.1 CPU and RAM usage

Two tasks were used to test the CPU and RAM usage for all three frameworks. The first test involved an interactive control of the planar robot in the rViz. The planar

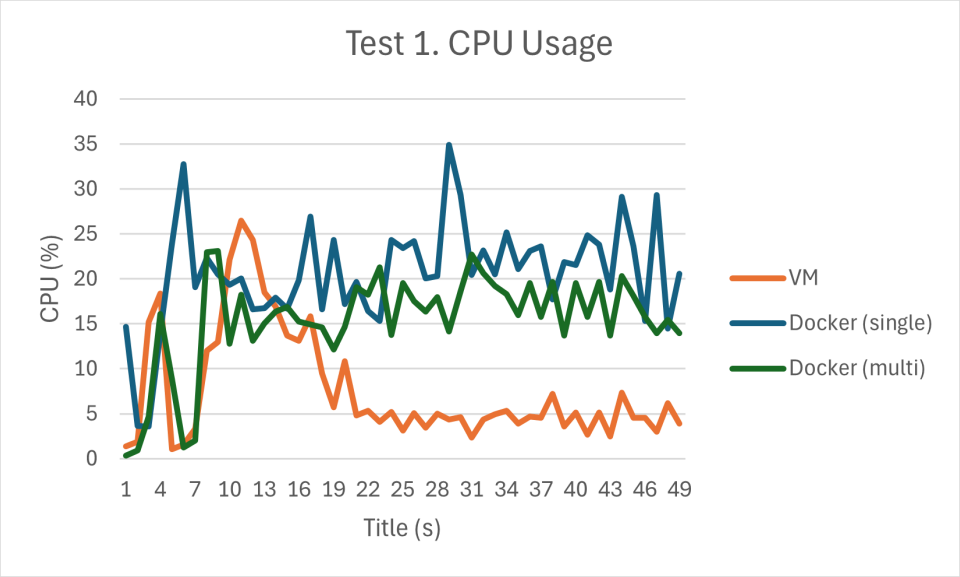

Figure 4-1: CPU usage during Task 1

Table 4.1: Average CPU Usage during Task 1

Framework	Avg CPU Usage
VM	7.67%
Docker (single)	20.67%
Docker (multi)	15.35%

robot movement was controlled by dragging the robot in the visualization scene and executing the movement through rViz controls. Three consecutive position changes were executed. The task tests resource utilization during complex communication in a simple task. The ROS topic receives position values from the display, while the display visualizes the movement.

4-1 shows the CPU Usage of three frameworks for task 1. All three frameworks had a spike during the launch of rViz. Both Docker architectures remained at the same level with occasional spikes. VM CPU usage, however, decreased as sharply as it increased and leveled off for an overall average of 7.67%. Average CPU Usage for single and multiple container Docker frameworks was 20.67% and 15.35%, respectively. Docker had a higher CPU usage overall, with a single container being the highest.

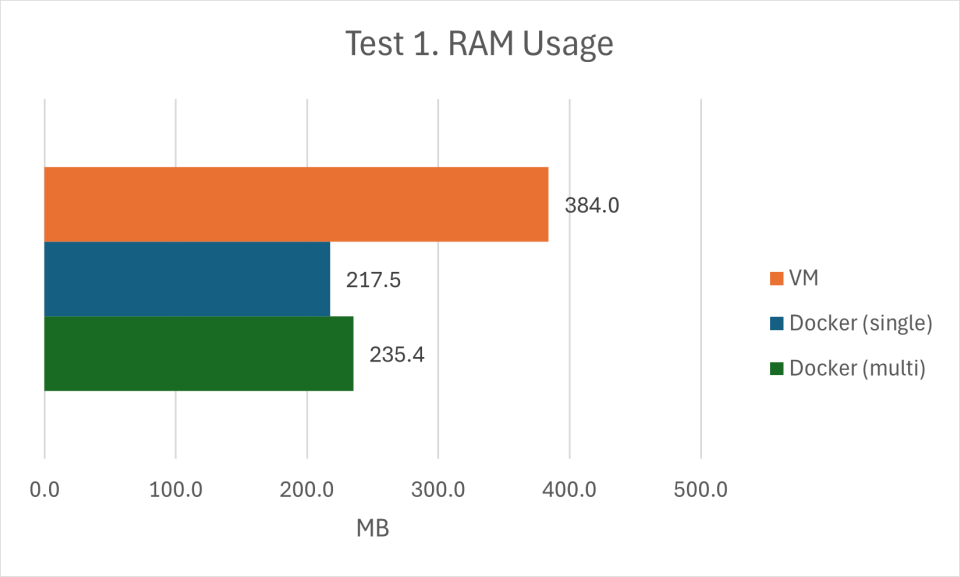

Figure 4-2: RAM usage during Task 1

RAM usage for the interactive task is depicted in 4-2. VM RAM usage was highest with 384 MB. Docker RAM usage stayed in the 200s, single 217 MB and multi 235 MB.

Docker had a higher CPU utilization than VM. In contrast to the majority of the previous works, VM resource utilization was better than Docker's. The reason for this could be in the interactive nature of the task. VM contains the GUI it uses, so all of processes are happening inside of it. Docker, on the other hand, uses the GUI on the host through WLS. The higher usage might be explained by the additional communication between the host and the Docker containers. There is also a difference between single-container and multiple-container architecture. In the single container, all process are executed in the same container environment. The GUI access is given to the whole container, meaning all processes can use it. Multi-container environment separates the processes. Thus, only the container running the visualization has access to the GUI, while containers containing non-visual processes are simplified. Overall, the CPU usage of this task depended more on the interactive communication rather than on the simplicity of the three movements task. It is also important to note that the motion control on the GUI is not identical due to human interaction, but it is similar. It should not affect the CPU values drastically.

The second task was a coded movement. 500 trajectory points were sent to a planar robot at a 10 Hz frequency. The planar robot completed 50 full side sweep movements and finished by returning to the original position. The task took around 52 seconds to complete. The purpose of this task case is to test the frameworks in identical, longer executions.

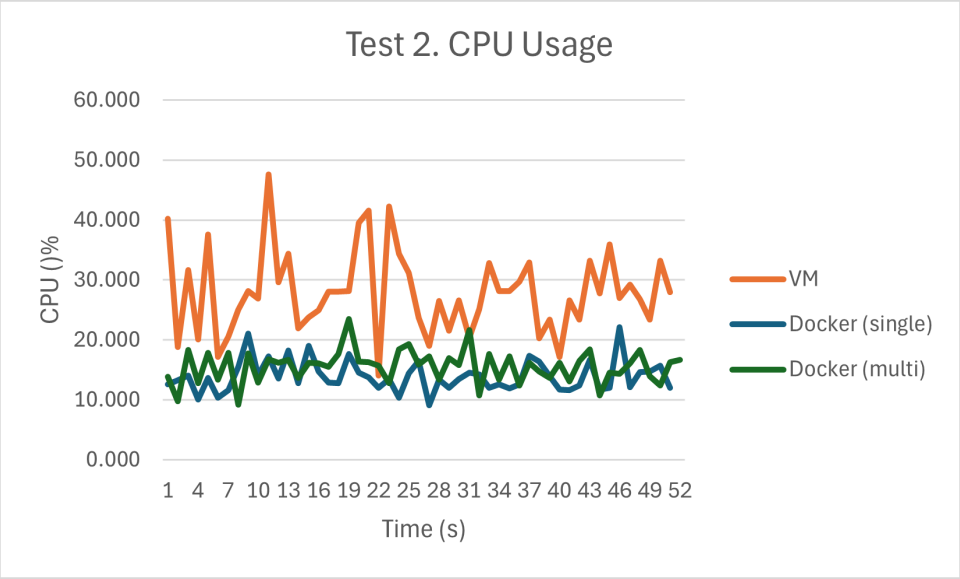

Figure 4-3: CPU usage during Task 2

Table 4.2: Average CPU Usage during Task 2

Framework	Avg CPU Usage
VM	27.93 %
Docker (single)	14.00%
Docker (multi)	15.54%

4-3 shows the CPU Usage for task 2. VM CPU was the highest in this task. It averaged at 27.93% while fluctuating erratically. Docker, contrary to the first task performance, was steadily low for both cases. While the multiple container architecture had almost the same average value with 15.54% for the second task, the single container decreased to an average of 14%.

Docker had a lower CPU Usage than the VM. Results of this task were consistent with the expected values. The related works show a similar difference with Docker

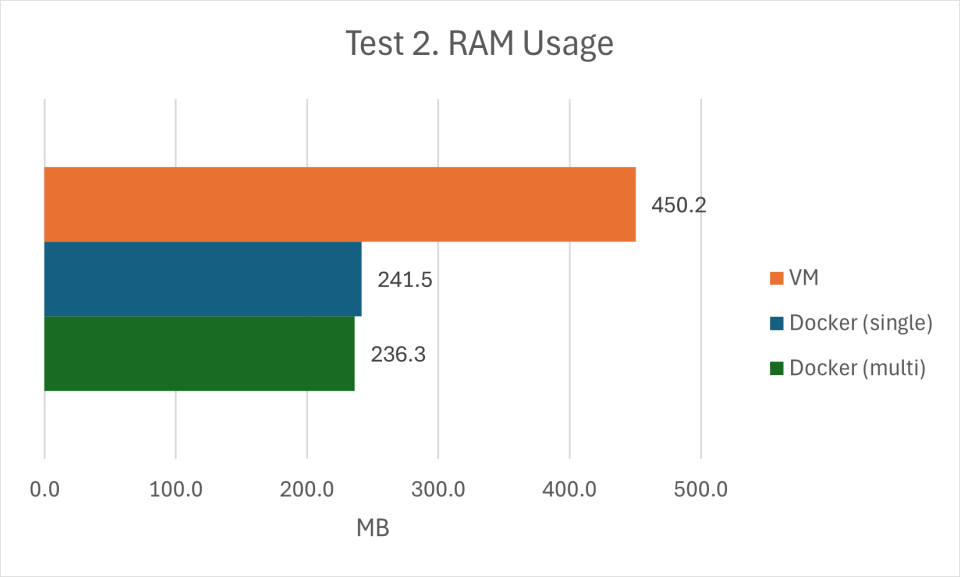

Figure 4-4: RAM usage during Task 2

being more efficient [1], [16], [24], [40]. VM's CPU usage increased with the extended task complexity. Docker's single container approach decreased to 14%, slightly outperforming the multi-container. The GUI in this task was used only for visualization, so Docker container didn't encounter additional communication needs. Multi-container approach stayed steadily at 15%. This suggests that Docker containers don't increase resource consumption steeply with a slight increase in complexity. The further analysis revealed that the Docker measurements included the Docker Desktop values. The CPU usage of individual containers was 3-5% less than the reported.

RAM usage was similar for both tasks. VM had an average of 450.2 MB, Docker's usage was 241.5 and 236.3 MB.

Although the second task values were consistent with expected results, the first task with interactive control showed different results. VM outperformed Docker with less resource utilization. However, it spiked during the longer task. Docker was consistent in its resource consumption. It had optimized results when the task increased in complexity. Analysis of results shows that all frameworks have advantages. VM is user-friendly, but can drastically increase resource consumption. Docker is a bit more complex. The single container framework is fairly easy to master. The multiple container framework is the most diverse in application. It can be used to simplify the

framework to simple tasks, in rapid development environments for fast updates and modifications, or it can be used in automated complex systems.

The initial scope of the software framework is for a full snake planar robot control. More testing is needed to determine the efficiency of the framework in other application areas.

Chapter 5

Conclusion

ROS is an essential toolkit for the robotics field. It is open-source, so the volume of robotic packages available makes it an important part of robotic systems. However, it also creates dependency and version management issues. Learning robotics with ROS for a new student, therefore, could be difficult and unnecessarily complex. The solution is to create a software framework with ROS packages preinstalled. Virtualization technologies were used to compile and isolate the framework. The framework was built by collecting and compiling all the ROS packages for the snake planar robot with its dependencies. The collected packages were virtualized by VM and Docker to create an easy, ready-to-use framework. VM virtualization and two methods of Docker containerization were compared. The VM image consisted of Ubuntu with ROS preinstalled. It's a user-friendly framework with a desktop interface for a familiar way of using ROS.

Two architectures of Docker containerization were tested. In a single container approach, the image builds one container where all the processes are executed. The framework image is created by all the necessary installation steps into one configuration file, Dockerfile. It contains the steps for the installation of dependencies and the ROS package, and the initialization of the workspace. In a multiple container approach, the framework is built upon the single container architecture. It improves the architecture by separating processes into services for modularity. It also demonstrates the possibility of adding extra packages for easy upgrade and modifica-

tion. Performance comparison was conducted between the three architectures. VM setup was easier, but it had a slower boot-up time and was heavier. It is also more user-friendly than Docker. Docker had an initial adaptation period for learning the Docker structure and syntax. The setup of Docker is more space-efficient. Update and modification of images can be carried out more easily, with less space needed. Multi-container setup, particularly, revealed a potential for scalable development and deployment in different areas like research and industry.

Two test tasks were performed to compare CPU and RAM utilization. The first task included interaction motion control through RViz visualization. The second task was a longer coded movement. In the first part, VM unexpectedly outperformed Docker. It is possible that it was due to the use of GUI through WSL in the Docker containers. Notwithstanding the first task, Docker outperformed the VM in the second part.

In conclusion, VM is a more user-friendly approach with higher resource consumption. Docker is more difficult, but efficient. Docker can be a great tool in educational environments. It provides cross-platform compatibility, good portability with Docker repository, modularity for fast update and upgrade, and scalability for complicated setups.

Bibliography

[1] A. K. Yadav, M. L. Garg, and Ritika, "Docker containers versus virtual machine-based virtualization," *Advances in Intelligent Systems and Computing*, p. 141–150, Sep 2018.

[2] M. Indri, F. Sibona, and L. O. Russo, "Pp - standard architecture to enable fast software prototyping for robot arms," *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, p. 721–728, Sep 2018.

[3] K. Jiang and Q. Song, "A preliminary investigation of container-based virtualization in information technology education," *Proceedings of the 16th Annual Conference on Information Technology Education*, p. 149–152, Sep 2015.

[4] C. A. Garcia, M. V. Garcia, E. Irisarri, F. Perez, M. Marcos, and E. Estevez, "Flexible container platform architecture for industrial robot control," *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, p. 1056–1059, Sep 2018.

[5] E. Cervera and A. P. Del Pobil, "Roslab: Sharing ros code interactively with docker and jupyterlab," *IEEE Robotics amp; Automation Magazine*, vol. 26, no. 3, p. 64–69, Sep 2019.

[6] F. H. Martinez, "Docker: A tool for creating images and launching multiple containers with ros os," *Tekhnê*, vol. 19, no. 1, p. 13–22, May 2022.

[7] P. Melo, R. Arrais, and G. Veiga, "Development and deployment of complex robotic applications using containerized infrastructures," *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, p. 1–8, Jul 2021.

[8] S. Aldegheri, N. Bombieri, F. Fummi, S. Girardi, R. Muradore, and N. Piccinelli, "Late breaking results: Enabling containerized computing and orchestration of ros-based robotic sw applications on cloud-server-edge architectures," *2020 57th ACM/IEEE Design Automation Conference (DAC)*, Jul 2020.

[9] F. Lumpp, M. Panato, F. Fummi, and N. Bombieri, "A container-based design methodology for robotic applications on kubernetes edge-cloud architectures," *2021 Forum on specification amp; Design Languages (FDL)*, p. 01–08, Sep 2021.

[10] J. M. Cañas, E. Perdices, L. García-Pérez, and J. Fernández-Conde, "A ros-based open tool for intelligent robotics education," *Applied Sciences*, vol. 10, no. 21, p. 7419, Oct 2020.

[11] L. El Hafi, G. A. Garcia Ricardez, F. von Drigalski, Y. Inoue, M. Yamamoto, and T. Yamamoto, "Software development environment for collaborative research workflow in robotic system integration," *Advanced Robotics*, vol. 36, no. 11, p. 533–547, Jun 2022.

[12] M. Sollfrank, F. Loch, S. Denteneer, and B. Vogel-Heuser, "Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation," *IEEE Transactions on Industrial Informatics*, vol. 17, no. 5, p. 3566–3576, May 2021.

[13] K. Fernalld, T. OConnor, S. Sudhakaran, and N. Nur, "Lightweight symphony: Towards reducing computer science student anxiety with standardized docker environments," *The 24th Annual Conference on Information Technology Education*, p. 15–21, Oct 2023.

[14] O. Bentaleb, A. S. Belloum, A. Sebaa, and A. El-Maouhab, "Containerization technologies: Taxonomies, applications and challenges," *The Journal of Supercomputing*, vol. 78, no. 1, p. 1144–1181, Jun 2021.

[15] A. S. Wazan, M. A. Kuhail, K. Hayawi, and R. Venant, "Which virtualization technology is right for my online it educational labs?" *2021 IEEE Global Engineering Education Conference (EDUCON)*, p. 1254–1261, Apr 2021.

[16] A. Bhardwaj and C. R. Krishna, "Virtualization in cloud computing: Moving from hypervisor to containerization—a survey," *Arabian Journal for Science and Engineering*, vol. 46, no. 9, p. 8585–8601, Apr 2021.

[17] Z. Li, "Comparison between common virtualization solutions: Vmware workstation, hyper-v and docker," *2021 IEEE 3rd International Conference on Frontiers Technology of Information and Computer (ICFTIC)*, p. 701–707, Nov 2021.

[18] M. Aniruddh, D. Anubhav, S. C. Mouli, B. Sahana, and A. A. Deshpande, "Comparison of containerization and virtualization in cloud architectures," *2021 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT)*, p. 1–5, Jul 2021.

[19] Z. Li, M. Kihl, Q. Lu, and J. A. Andersson, "Performance overhead comparison between hypervisor and container based virtualization," *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*, p. 955–962, Mar 2017.

[20] C. Liu, H. Ren, N. Zhou, G. Li, X. Liang, W. Gui, and C. Yang, "Industry-oriented lightweight simulation system," *2023 IEEE 29th International Conference on Parallel and Distributed Systems (ICPADS)*, p. 1099–1106, Dec 2023.

[21] S. Kaiser, M. S. Haq, A. S. Tosun, and T. Korkmaz, "Container technologies for arm architecture: A comprehensive survey of the state-of-the-art," *IEEE Access*, vol. 10, p. 84853–84881, 2022.

[22] A. Subramaniam, D. H. Prajapati, K. Alremeithi, and W. Sealy, "Enhancing autonomous robotics through cloud computing," *2024 IEEE International Conference on Industry 4.0, Artificial Intelligence, and Communications Technology (IAICT)*, p. 259–265, Jul 2024.

[23] R. Queiroz, T. Cruz, J. Mendes, P. Sousa, and P. Simões, "Container-based virtualization for real-time industrial systems—a systematic review," *ACM Computing Surveys*, vol. 56, no. 3, p. 1–38, Oct 2023.

[24] T. Betz, L. Wen, F. Pan, G. Kaljavesi, A. Zuepke, A. Bastoni, M. Caccamo, A. Knoll, and J. Betz, "A containerized microservice architecture for a ros 2 autonomous driving software: An end-to-end latency evaluation," *2024 IEEE 30th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, p. 57–66, Aug 2024.

[25] A. S. Seisa, S. G. Satpute, and G. Nikolakopoulos, "Comparison between docker and kubernetes based edge architectures for enabling remote model predictive control for aerial robots," *IECON 2022 – 48th Annual Conference of the IEEE Industrial Electronics Society*, p. 1–6, Oct 2022.

[26] S. S. Policharla, S. N, T. A, P. Auradkar, and P. N. Anantharaman, "Ros-kafka gateway for scalable, remote and cross-platform robotic system communication," *2023 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, p. 9–15, Nov 2023.

[27] A. Wendt and T. Schuppstuhl, "Proxying ros communications — enabling containerized ros deployments in distributed multi-host environments," *2022 IEEE/SICE International Symposium on System Integration (SII)*, p. 265–270, Jan 2022.

[28] S. Hardikar, P. Ahirwar, and S. Rajan, "Containerization: Cloud computing based inspiration technology for adoption through docker and kubernetes," *2021 Second International Conference on Electronics and Sustainable Communication Systems (ICESC)*, p. 1996–2003, Aug 2021.

[29] L. Mickey, C. Robert, and R. Arthur, "Starling: Containerisation architecture for scalable local development, deployment and testing of multi-uav systems," 2022. [Online]. Available: http://raaslab.org/rss2022/assets/contributed_papers/RSS2022_Li_Clarck_Richards.pdf

[30] B. Lampe, L. Reiher, L. Zanger, T. Woopen, R. van Kempen, and L. Eckstein, "Robotkube: Orchestrating large-scale cooperative multi-robot systems with kubernetes and ros," *2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC)*, Sep 2023.

[31] S. Shibuya, K. Kobayashi, T. Ohkubo, K. Watanabe, K. Tian, N. J. Sebi, and K. C. Cheok, "Seamless rapid prototyping with docker container for mobile robot development," *2022 61st Annual Conference of the Society of Instrument and Control Engineers (SICE)*, p. 1063–1068, Sep 2022.

[32] E. Cervera, "Run to the source: The effective reproducibility of robotics code repositories," *IEEE Robotics amp; Automation Magazine*, vol. 31, no. 2, p. 125–134, Jun 2024.

[33] J.-P. Busch, L. Reiher, and L. Eckstein, "Enabling the deployment of any-scale robotic applications in microservice architectures through automated containerization," *2024 IEEE International Conference on Robotics and Automation (ICRA)*, p. 17650–17656, May 2024.

[34] J. P. Hacker, J. Exby, D. Gill, I. Jimenez, C. Maltzahn, T. See, G. Mullendore, and K. Fossell, "A containerized mesoscale model and analysis toolkit to accelerate classroom learning, collaborative research, and uncertainty quantification," *Bulletin of the American Meteorological Society*, vol. 98, no. 6, p. 1129–1138, Jun 2017.

[35] W. Wang, T. Wang, and G. Yin, "Container-based complex programming skills training platform," *2019 IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, p. 1–5, Oct 2019.

[36] M. F. Ashari, A. Bhawiyuga, and A. Basuki, "The development of hands-on lab platform using container-based virtualization technology," *Proceedings of the 8th International Conference on Sustainable Information Engineering and Technology*, p. 304–310, Oct 2023.

[37] A. K. Erdogmus and U. Yayan, "Virtual robotic laboratory compatible mobile robots for education and research," *2021 International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*, p. 1–6, Aug 2021.

[38] A. Mazhitov, A. Adilkhanov, Y. Massalim, Z. Kappassov, and H. A. Varol, "Deformable object recognition using proprioceptive and exteroceptive tactile sensing," in *2019 IEEE/SICE International Symposium on System Integration (SII)*, 2019, pp. 734–739.

[39] L. El Hafi, Y. Zheng, H. Shirouzu, T. Nakamura, and T. Taniguchi, "Serket-sde: A containerized software development environment for the symbol emergence in robotics toolkit," *2023 IEEE/SICE International Symposium on System Integration (SII)*, p. 1–6, Jan 2023.

[40] S. Giallorenzo, J. Mauro, M. G. Poulsen, and F. Siroky, "Virtualization costs: Benchmarking containers and virtual machines against bare-metal," *SN Computer Science*, vol. 2, no. 5, Aug 2021.