ERDC TR-23-10

Engineer Research and Development Center

US Army Corps of Engineers®
Engineer Research and Development Center

ERDC
ENGINEER RESEARCH & DEVELOPMENT CENTER

UGV–Localization in 3D and Path-Planning (U-L3AP)

Docker Containers and Images for Robot Operating System (ROS)–Based Applications

Amir Naser, Osama Ennasr, Ahmet Soylemezoglu, and Garry Glaspell

July 2023

UGV–Localization in 3D and Path-Planning
(U-L3AP)

ERDC TR-23-10
July 2023

Docker Containers and Images for Robot Operating System (ROS)–Based Applications

Amir Naser, Osama Ennasr, and Garry Glaspell

US Army Engineer Research and Development Center (ERDC)
Geospatial Research Laboratory (GRL)
7701 Telegraph Road
Alexandria, VA 22315-3864

Ahmet Soylemezoglu

US Army Engineer Research and Development Center (ERDC)
Construction Engineering Research Laboratory (CERL)
2902 Newmark Drive
Champaign, IL 61824

Final Technical Report (TR)

Prepared for US Army Engineer Research and Development Center (ERDC)
3909 Halls Ferry Road
Vicksburg, MS 39180

Under FLEX-4

Abstract

Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package and ship out an application with all of the parts it needs, such as libraries and other dependencies. Herein, we investigate using a Docker image to deploy and run our Robot Operating System (ROS)–based payload on a robot platform. Ultimately, this would allow us to quickly and efficiently deploy our payload on multiple platforms.

Contents

Figures

Preface

This study was conducted for the US Army Engineer Research and Development Center (ERDC) of the US Army Corps of Engineers (USACE). It was funded by ERDC under FLEX-4.

The work was performed by the Data Representation Branch of ERDC, Geospatial Research Laboratory (GRL). At the time of publication, Mr. Vineet Gupta was branch chief, Mr. Jeff Murphy was division chief, and Dr. Austin Davis was the technical director of the Geospatial Research Laboratory. The deputy director of ERDC-GRL was Dr. Valerie L. Carney, and the director was Mr. David R. Hibner. Work was also performed by the Warfighter Engineering Branch of ERDC, Construction Engineering Research Laboratory (CERL). At the time of publication, Mr. Jeff Burkhalter was branch chief, Dr. George Calfas was division chief, and Mr. Jim Allen was the technical director of CERL. The deputy director of ERDC-CERL was Ms. Michelle Hanson, and the director was Dr. Andrew Nelson.

The authors would like to acknowledge the following individuals for their contributions to this project: Dr. Anton Netchaev, Mr. Steven Bunkley, and Mr. Chuck Ellison.

COL Christian Patterson was commander of ERDC, and Dr. David W. Pittman was the director.

1 Introduction

1.1 Background

Docker, an open platform for developing, shipping, and running applications, is used to run software packages called *containers*. A container is a lightweight and portable package that contains everything needed to run the software. This includes the application code, system tools, libraries, and run time. The only thing preventing a container from being able to stand alone is that it relies on the host's operating system (OS) and kernel for low-level services, such as resource management and network access. Because containers include everything else that the application needs to run, they are easy to move from one environment to another. This makes it possible to run the same application on a developer's laptop, a test server, and a production server without changing the code.

There are several benefits to using Docker:

- *Portability.* You can develop and test your application on your local machine and then, with minimal effort, deploy it to any other machine that is running Docker. The only limitation to portability is that, because containers do not include their own OS, the container can only run on the same OS on which it was created (e.g., if you want to run a Linux-based container, the host machine must have a Linux OS). If you plan to use a Linux-based container on a Windows OS host, or vice versa, you will need to either run your container through a virtual machine (VM) or use Docker Desktop (which runs containers through a Linux VM in the background). With all of this in mind, containers make it easy to move applications from one environment to another. This can be especially useful when working with microservices because containers allows you to build, test, and deploy each service separately (White and Christensen 2017).
- *Isolation.* Each container runs in its own isolated environment. As a result, you can run multiple applications on the same host without them interfering with one another and potentially causing conflicts. For our application, we can install multiple versions of the Robot Operating System (ROS) with different configurations on a single

machine. Each image could be tailored for a specific mission (González-Nalda et al. 2017).
- *Scalability*. Docker makes it easy to scale applications horizontally across multiple hosts. Specifically, you can create multiple containers and distribute the computational load between them (Wendt and Schüppstuhl 2022).
- *Ease of use*. Docker provides a simple and consistent way to package and deploy applications, making it easier for developers to work in different environments. This reduces the delay between writing code and running it in production (Cervera and Del Pobil 2019).

1.2 Objectives

This report addresses the focus areas established in *Army Multi-Domain Intelligence: FY21-22 S&T Focus Areas* (Office of the Deputy Chief of Staff 2020). Specifically, this work addresses this statement from that text: "Wars will be fought at hyper speed and scale, dominated by technologies such as robotics and autonomous systems (RAS), machine learning (ML), and AI [artificial intelligence] capabilities, which are widely available, packaged, and ready for use" (5). Our objective is to leverage containers for rapid deployment of software to multiple robotic platforms. Containers also simplify the process of maintaining robotic platforms and allow for version control.

1.3 Approach and Scope

Containers and VMs are both virtualization methods that allow you to run multiple OSs on a single physical machine. However, they work in slightly different ways.

A container is a lightweight, portable, and executable package. It includes everything (besides the base OS that is needed and provided by the host machine) that an application needs to run, including the application code, libraries, dependencies, and run time. Because containers include only the minimum required components and share the host OS's kernel, they are much lighter and more efficient than VMs. Containers are typically used to deploy and run microservice-based applications, which are made up of small, independent, and modular components that communicate with each other through application programming interfaces (APIs).

A VM is a complete emulation of a physical computer that runs on top of a host OS. A VM includes a full copy of an OS and virtual hardware, such as CPU, memory, storage, and networking devices. Because VMs include a full copy of the OS and virtual hardware, they are much heavier and require more resources than containers. VMs are typically used to run legacy applications, test software in different environments, and isolate applications from one another.

For the purposes of running applications on robots, containers can be used to quickly and safely install the necessary files and dependencies on a robotics platform. This report focuses on creating, running, and deploying a container built around our sensor payload for applications focused on simultaneous localization and mapping.

2 Working with Docker Containers

This chapter provides instructions for installing and running Docker. Mounting volumes and updating and pulling Docker images are also discussed. Finally, a Dockerfile that contains all the software and files necessary to recreate our robot payload is provided.

2.1 Installing Docker for Linux (Ubuntu)

To start using Docker, you will need to install it on your system. Docker is available for Windows, Mac, and Linux. For this document, we will focus solely on how to install Docker for Linux (Ubuntu). Instructions for macOS and Windows are provided on the Docker website.*

Docker stores images in different locations, depending on whether you use root privileges (i.e., using `sudo` on the command line). If the Docker daemon runs as the root user, it has access to the entire file system. Thus, the image is stored in the default Docker image storage location, which is typically `/var/lib/docker`. However, if the Docker image is built without using `sudo`, the image is stored in the current user's home directory, under the `.docker` directory. This is because the current user does not have permission to write to the `/var/lib/docker` directory.

It is generally recommended to use `sudo` when working with Docker when you need to ensure that the daemon has all the necessary permissions to perform operations (Docker, n.d.). One specific reason to use `sudo` is so that you will have root privileges in order to implement the GUI in the `Docker/ros:noetic` images. However, this could pose a security risk. With all of this in mind, the steps to install Docker are as follows.

1. Add the Docker repository to your system.

```
sudo apt-get update

sudo apt-get -y install apt-transport-https ca-certificates
    curl gnupg-agent software-properties-common
```

* https://docs.docker.com/get-docker/

```
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo
   apt-key add -

sudo add-apt-repository "deb␣[arch=amd64]␣https://
   download.docker.com/linux/ubuntu␣$(lsb_release\␣-cs)
   ␣stable"
```

2. Install Docker.

```
sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io
   uidmap
```

3. Add your user account to the docker group.

```
sudo usermod -aG docker $USER newgrp docker
```

Optional: Download the latest Docker Desktop package. You can install Docker Desktop, which is an application that allows you to run and view your Docker containers and images. However, we ran into issues with Docker Desktop when using GUI applications. Also, Docker Desktop only shows containers and images run without `sudo` privileges. Therefore, if you plan to use GUI interfaces, it is recommended that you do not install Docker Desktop.

```
wget https://desktop.docker.com/linux/main/amd64/docker-
   desktop-4.12.0-amd64.deb
```

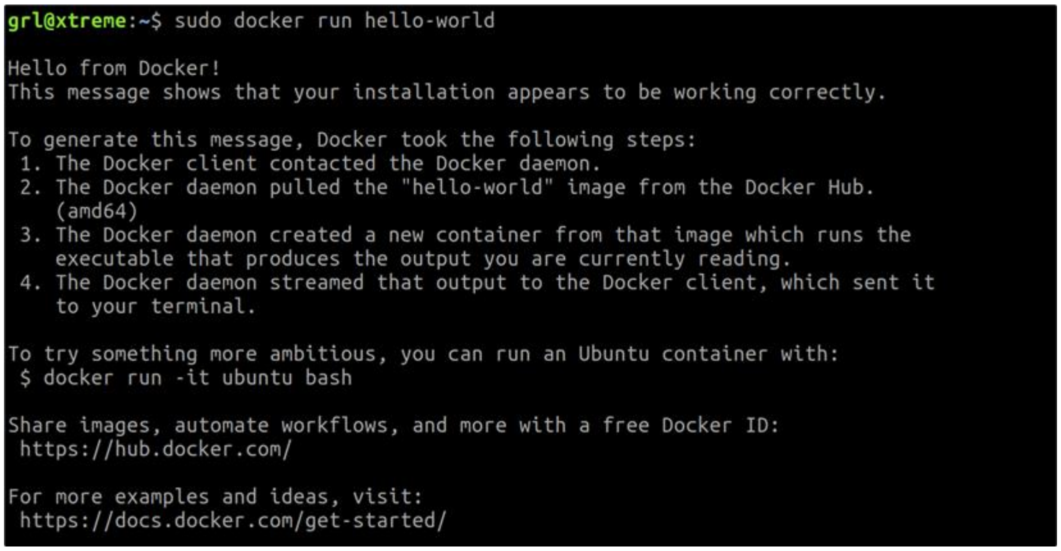
4. Optional: Install Docker Desktop.

```
sudo apt-get install ./docker-desktop-4.12.0-amd64. deb
```

You can verify Docker Desktop installation by running the *hello-world* image. Figure 1 shows the output of running the command that follows:

```
sudo docker run hello-world
```

Figure 1. Terminal output from running the hello-world example.



```
grl@xtreme:~$ sudo docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

2.2 How to Pull and Run a Robot Operating System (ROS) Noetic Image

Rather than building an ROS from scratch, official ROS docker images can be used. Images for popular ROS 1 distributions, such as Melodic and Noetic, as well as ROS 2 images for Foxy and Humble, are available. The following sections focus on pulling ROS 1 Noetic.

2.2.1 Pulling and Running an Image

Pull the ROS Noetic Docker image from Docker Hub with the command that follows:

```
docker pull ros:noetic
```

To run a new Docker container using the ROS Noetic image, use the following command:

```
docker run -it --name my_ros_container ros:noetic bash
```

This will start a new Docker container, with the ROS Noetic image, and open a bash shell inside the container. In this command, `--name` is a flag that specifies a name for the container. The value that follows the flag, `my_ros_container`, is the name that will be assigned to the container. You can now use the ROS Noetic command line tools and libraries inside the container.

2.2.2 Running an Image with GUI Enabled

To use the ROS Noetic GUI tools, the container requires additional options to enable access to the host's display and input devices. The image also needs to be pulled with `sudo`.

```
xhost+
```

```
sudo docker run -it --name my_ros_container --net=host
   --env="DISPLAY" --volume="$HOME/.Xauthority:/root/.
   Xauthority:rw" ros:noetic bash
```

This command allows you to run ROS Noetic GUI tools inside the container and to display them on the host's desktop. This is done by sharing the host's X Server with the container by creating this volume:

```
--volume="\$HOME/.Xauthority:/root/.Xauthority:rw".
```

Then, the host's display environment variable, `--env="DISPLAY"`, is shared to the container. Last, you run the container with the host network driver with `--net=host.`

You can also mount a volume to the container for persisting data. Specifically for our use case, this is our ROS workspace; mounting a volume ensures that our workspace is maintained between sessions. The generic line to mount a volume is to add the following tag to your `sudo docker` run command:

```
--volume="/path/on/host:/path/in/container"
```

or

```
-v /path/on/host:/path/in/container
```

Volumes are covered in greater detail in Section 2.3.

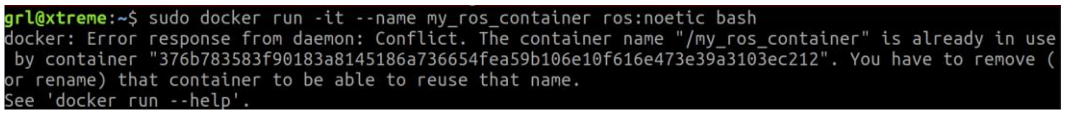
2.2.3 How to Rerun an Already Running Container

If you exit a container after it has been created and run using a command similar to the one that follows, it cannot be reopened by running the same `docker run` command:

```
docker run -it --name my_ros_container ros:noetic bash exit
```

Figure 2 shows the result of attempting to reopen an exited, but not removed, image.

Figure 2. Terminal output resulting from exiting but not removing the image.



```
grl@xtreme:~$ sudo docker run -it --name my_ros_container ros:noetic bash
docker: Error response from daemon: Conflict. The container name "/my_ros_container" is already in use
 by container "376b783583f90183a8145186a736654fea59b106e10f616e473e39a3103ec212". You have to remove (
or rename) that container to be able to reuse that name.
See 'docker run --help'.
```

There are a couple of options available for avoiding this issue. You can either stop and remove the container and repeat the initial `docker run` command, or you can use the `docker start` command.

To stop, remove, and rerun a running container named `my_ros_container`, use the following commands:

```
docker stop my_ros_container
docker rm my_ros_container
docker run -it --name my_ros_container ros:noetic bash
```

However, by doing so, you eliminate any changes made to the initial container or image if you have not already committed. The preferred option, then, is to use the `docker start` command. This command effectively reruns the container without requiring you to stop and remove the previous container. For example, if you want to start a container named `my_ros_container`, you can use the following command:

```
docker start my_ros_container
```

This command will start the container and run it in the background. If you want to attach to the container and access its terminal, you can use the `-a` flag to attach to the container. However, the Docker daemon sometimes freezes and does not work properly when using this method:

```
docker start -a my_ros_container
```

The `docker run` command can also be used to start a new container based on an image. You can start a new container by giving it a different name, via the `-name` flag, to specify the name of the container. For example, you could use the command that follows:

```
docker run --name my_new_container my_image
```

This command starts a new container based on the `my_image` image and gives it the name `my_new_container`.

Overall, to ensure that the updates in your Docker container do not disappear when you stop and remove a container, you can either use a different name for the new container or make sure you commit all your changes to the container or image prior to stopping or removing it. Updating and committing changes is discussed in depth in Section 2.6.

2.2.4 How to Pull ROS 2

This document is focused on ROS 1 (specifically Noetic), but ROS 2 can also be used (Martinez 2022). ROS 1 (Noetic) is currently supported until May of 2025. ROS 2 (Humble) is supported until May of 2027. At the time of this writing, ROS 1 possesses a larger user base than ROS 2 (Scott and Foote 2022). ROS 2 Humble can be accessed using the command that follows:

```
docker pull ros:humble
```

It is also possible to bridge between ROS 1 and ROS 2. This is outside the scope of this report, but more information on this topic can be found on the Docker website.*

2.3 Volumes

In Docker, a volume is a persistent storage location that is outside of a container's image and can be used to share data between the host and the container or between multiple containers. Volumes are managed by Docker and can be used to store data that need to persist even if the container is stopped or deleted.

Volumes are useful in a number of situations, including

- sharing configuration files, logs, or other data between the host and the container;

* https://hub.docker.com/_/ros

- storing data that are generated by a container, such as a database, cache, or uploaded files; and
- sharing data between multiple containers; for example, when using a volume as a message queue or shared cache.

2.3.1 Using Volumes in a Docker Container

To use a volume in a Docker container, specify the `-v` flag when starting the container and then add the host path and the container path, separated by a colon.

```
docker run -v /host/path:/container/path my_image bash
```

As stated in Section 2.2.3, you can also use this command:

```
docker run --volume="/host/path:/container/path"\ my_image bash
```

This will create a volume that is mounted at `/container/path` within the container and is linked to `/host/path` on the host. Any data written to the volume by the container will persist on the host, and any changes made to the data on the host will be visible to the container.

You can also use named volumes to manage volumes more easily. Named volumes are created using the `docker volume create` command and can be referenced by name when starting a container.

```
docker volume create my_volume
docker run -v my_volume:/container/path my_image bash
```

This will create a named volume called `my_volume` and mount it within the container at `/container/path`. Named volumes can be managed and reused across multiple containers, thus making it easier to share data between containers.

Last, you can use the `docker cp` command to copy a directory and its contents from the host to a running Docker container. The general syntax for the `docker cp` command is as follows:

```
docker cp <src> <container>:<dest>
```

In the `docker cp` command, `<src>` is the path to the directory or file on the host that you want to copy, `<container>` is the identity or name of the running container, and `<dest>` is the destination path within the container where you want to copy the files.

For example, to copy the directory `/path/on/host/` to the container's `/path/in/container/` directory, you would run the following command:

```
docker cp /path/on/host/ my_running_container:/path/in/
   container/
```

This will copy the entire directory, including all files and subdirectories, from the host to the container. To access the host directory and write to the destination path within the container, however, you will need to have the appropriate permissions.

2.3.2 Mounting Devices

Sensors, such as cameras, are essential to a robot payload. To get the cameras to work within the Docker container, they must be mounted. Although it functions similarly to a volume, slightly different syntax is used when mounting a device from your host to the container. A volume could be mounted as follows:

```
docker run -v /host/path:/container/path my_image bash
```

To mount a device, however, you would need to change the `-v` flag to a `--device` flag, as follows:

```
docker run --device /dev/path:/container/dev/path my_image bash
```

If you plan to create udev rules for a device, the udev rules only need to be applied on the host machine. Udev rules allow for symbolic links. This is important if you have two identical sensors. With symbolic links, the sensors can be differentiated as front or back (or left and right) cameras. You can then mount the device as previously shown, and it will follow the udev rules created on the host machine.

2.4 Docker Compose to Run Containers

2.4.1 Description and Basics

When trying to start a container that includes many different inputs and flags, you can use Docker Compose. Docker Compose is a tool for defining and running multi-container Docker applications. It is used to define an application's services, networks, and volumes in a single file, called a `docker-compose.yml file`, and then to start and stop the services using a single command.

Docker Compose is useful because it allows developers to define and run multiple containers for their applications as a single unit, making it easier to manage, scale, and deploy applications. With Docker Compose, developers can easily manage the configuration of multiple containers and avoid setup processes that are prone to manual errors.

Using Docker Compose instead of a bash script to run containers has several benefits. First, Docker Compose provides a simple and declarative way to define the containers and their configuration, making it easier to understand and manage the setup of a multi-container application. Second, Docker Compose automates the process of setting up and connecting containers, freeing developers from having to manually manage the low-level details of the containers. Finally, Docker Compose provides built-in support for scaling, rolling updates, and health checks, making it easier to manage the containers over time.

To ensure Docker Compose is available on your host machine, you can install it using the commands that follow:

```
sudo apt-get update
sudo apt-get install -y docker-compose
```

2.4.2 How to Create and Implement a Docker Compose File

The list that follows provides step-by-step instructions for creating and implementing a Docker Compose file.

1. Create a docker-compose.yml file. This file will define all the services, networks, and volumes used in your application.

2. Define services. In the docker-compose.yml file, define each service using the "services" key. For each service, specify the image name, ports to be exposed, environment variables, and so on.
3. Start the services. To run the container and to start the services defined in the docker-compose.yml file, use the following command:
   ```
   docker-compose up
   ```
4. Stop the services. To exit the container and to stop the services, use the following command: docker-compose down
5. Scale services. Scaling a service in Docker Compose allows you to increase or decrease the number of replicas of a particular service running in your application. This can be useful when you need to handle increased load on your application and need more instances of a service to handle the traffic. To scale a service, use the following command:
   ```
   docker-compose up --scale <service-name>=<number of replicas>
   ```
6. Connect to a service. Connecting to a service in Docker Compose allows you to run commands inside a running container of a particular service. This can be useful for debugging, checking logs, or making changes to the service. To connect to a running service, use the following command:
   ```
   docker-compose exec <service-name> <command>
   ```

The `docker run` command that follows was used to provide an example of how to create a `docker-compose.yml` file:

```
docker run -it --name my_ros_container --network=host
    --env="DISPLAY" -e FILE_PATH=/root/launch_files -e
    ROS_MASTER_URI=http://$HOSTNAME:11311 -v /etc/hosts
    :/etc/hosts --volume="$HOME/.Xauthority:/root/.
    Xauthority:rw" -v ~/Documents/git/Docker/
    directory_example:/root/launch_files --device /dev/
    device_example:/dev/device_example --rm my_image bash
```

The `docker-compose.yml` file can be seen here:

```
version: '3'
services:
  ros_container:
    image: my_image
    environment:
      DISPLAY: ${DISPLAY}
      FILE_PATH: /root/launch_files/
      ROS_MASTER_URI: http://${HOSTNAME}:11311
    volumes:
      - /etc/hosts:/etc/hosts
      - "${HOME}/.Xauthority:/root/.Xauthority:rw"
```

```
      -/Documents/git/Docker/directory_exam-
      ple:/root/launch_files
    devices:
      - /dev/device_example:/dev/device_example
    network_mode: host
    command: bash
    tty: true
    stdin_open: true
    restart: always
    container_name: my_ros_container
```

If you are including any environmental names from the host machine, you will need to ensure that they are exported prior to trying to run the `docker-compose.yml` file. For example,

```
export DISPLAY=:0
export HOSTNAME=localhost
```

Last, after all of the preceding steps are complete, you can implement the `docker-compose.yml` file by typing

```
docker compose up
```

Then, you can enter the container in another terminal window with the following command:

```
docker exec -it my_ros_container bash
```

When you are done with the container and want to stop and remove it, you can use this command:

```
docker compose down
```

2.5 Dockerfiles

2.5.1 Description and Basics

A Dockerfile is a text file that contains instructions for building a Docker image. It is used to automate the process of creating a Docker image so that you can build the same image without having to manually perform the steps.

A Dockerfile typically starts with a base image to use for the container (e.g., an OS), then adds additional files and packages to the image, and then configures the image to run a specific application or service. For example, the base `ros:noetic` image can be used as the base image, and then certain dependencies or libraries can be added to it via a Dockerfile. You can also include any additional commands that need to be run to set up your application.

To create a new Dockerfile, you can use any text editor to create a file with the name Dockerfile (no file extension). Then, you can add the necessary instructions to build your Docker image. Once you have a Dockerfile, you can use the `docker build` command to build a Docker image. You can then use the `docker run` command to run the image as a container.

Here is an example of a simple Dockerfile that installs the Apache Web server on an Ubuntu base image:

```
FROM ubuntu:20.04
RUN apt-get update && apt-get install -y apache2
CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

This Dockerfile has three instructions:

- `FROM` specifies the base image to use. In this case, it is using the ubuntu:20.04 image.
- `RUN` is used to execute a command in the container. In this case, it is updating the package manager index and installing the Apache Web server.
- `CMD` specifies the command to run when the container is started. In this case, it is starting the Apache Web server.

To build the image using a Dockerfile, you can use the `docker build` command while in the same directory as your Dockerfile. For example, you can use the following:

```
docker build -t my_image .
```

This will build an image called `my_image` using the Dockerfile in the current directory.

Each `RUN` command used in a Dockerfile creates a new layer in the updated Docker image. The more layers an image has, the more storage it uses and, therefore, the more space is taken up. It can be beneficial to group multiple terminal commands within one `RUN` command to limit the number of layers created. This can be done by using `&&` or `;` between terminal commands within one `RUN` command. For example, you could use what follows:

```
RUN apt-get update && apt-get install -y apache2
```

The preceding command would be used instead of either of the commands that follow:

```
RUN apt-get update
RUN apt-get install -y apache2
```

Do not, however, incorporate every single terminal command you want to implement within a singular `RUN` command to save space and storage. It is a good practice to incorporate a new `RUN` command in a Dockerfile when you want to separate different parts of the build process into separate layers. This helps to optimize the image size and improve its maintainability. Having multiple smaller `RUN` commands instead of one large command makes it easier to identify the cause of potential problems, revert changes, and make updates to specific parts of the image. The trade-off is that having more `RUN` commands will result in more layers and thus a larger image size. Ultimately, the choice between having fewer but larger `RUN` commands or more numerous but smaller ones will depend on the specific requirements of your project, so it is important to make an informed decision based on these factors. One practical example of incorporating many terminal commands within multiple, singular `RUN` commands can be seen in Section 2.4.2.

2.5.2 Dockerfile to Get Started with the Catkin Workspace

The catkin workspace in ROS-based software is where you build, modify, and install ROS-based packages. In this case, it is where we installed the simultaneous localization and mapping software.

What follows is a good starting Dockerfile to use after pulling the `ros:noetic` image, especially if you plan to use a catkin workspace (i.e., `catkin_ws`) in your Docker container.

```
# Start with a base ros:noetic image.
# If you rename your image in future and want to build
```

```
↪    on top of it, you have to use "FROM my_Image"
FROM ros:noetic

# Sets frontend to noninteractive for Dockerfile
ENV DEBIAN_FRONTEND=noninteractive
# Makes sure shell is using bash
SHELL ["/bin/bash", "-c"]
# Sets the directory on container start-up to /root/
↪   (~/)
WORKDIR /root/

# ====================================================
### Initial updates/needed base packages

## rosdep
# Fix permissions for rosdep
RUN sudo rosdep fix-permissions; \
# Update rosdep
rosdep update; \
##
## Set frontend to noninteractive in container
sudo echo 'debconf debconf/frontend select Noninteractive' |
  ↪ debconf-set-selections; \
##
## Essential packages
# Update and Upgrade apt-get
sudo apt-get update && sudo apt-get upgrade -y; \
# Install dialog apt-utils package to remove warnings
↪    for using apt-get while building
sudo apt-get install -y dialog apt-utils \
# Install net-tools package
net-tools \
# Install iputils-ping package
iputils-ping \
# Install git package
git \
# Install nano package
nano \
# Install curl package
curl \
# Install wget package
wget \
# Install terminator package
terminator \
# Install software-properties-common package
software-properties-common \
##
## Python3
# Install python3-rosdep package
python3-rosdep \
# Install python3-rosinstall package
python3-rosinstall \
# Install python3-rosinstall-generator package
python3-rosinstall-generator \
# Install python3-wstool package
python3-wstool \
# Install build-essential package
```

```
build-essential; \
# Update apt-get
sudo apt-get update; \
##
## Catkin_ws
# Source the setup.bash script
source /opt/ros/noetic/setup.bash; \
# Add source command to ~/.bashrc
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc;
↪  \
# Source ~/.bashrc
source ~/.bashrc; \
# Create catkin_ws directory
mkdir -p ~/catkin_ws/src; \
# Run the setup.bash script from the /opt/ros/noetic
↪    directory and change directory to ~/catkin_ws.
↪    Build the packages in the catkin workspace using
↪    catkin_make
. /opt/ros/noetic/setup.bash; cd ~/catkin_ws;
↪    catkin_make; \
# Add source command to ~/.bashrc
echo "source ~/catkin_ws/devel/setup.bash" >>
↪    ~/.bashrc; \
# Source ~/.bashrc
source ~/.bashrc
```

2.5.3 Full Dockerfile to Install Various Drivers and GitHub Repositories

At this time, we have a Docker image that contains the starting dependencies and catkin workspace that are required to install various drivers and GitHub repositories. The Docker image contains Rover-Pro driver, Microstrain driver, FLIR-Boson USB thermal camera driver, Velodyne driver, Ouster driver, SC-LIO-SAM, Elevation Mapping, Movebase, Traversability Estimation, Real-Time Appearance-Based (RTAB) Map, and RTAB-Map ROS.

To optimize the size and maintainability of this image, we used a singular `RUN` command for each additional resource added (i.e., one `RUN` command for the Rover-Pro driver, one for the Microstrain driver, and so on). This limited the total number of layers included in the image, thus making the image smaller, while also allowing the image to be easily maintained because each `RUN` command put each additional resource into a separate layer.

All of the resources added to this image were tested within the Docker image and were successfully run along with two separate robot payloads.

The overall Dockerfile that was used to create the image is included in Appendix A.

If starting from scratch, it could take up to two weeks for one of our payloads to be implemented with all of these necessities. However, with this Dockerfile, the overall image can be built, and the robot can be up and running, in around 30 minutes. Because this image has already been built and pushed onto a public repository on Docker Hub, the entire image can be pulled in approximately two minutes. The image can be pulled via the following command:

```
docker pull afnaser/grl-robot
```

More information on pulling images and on Docker Hub in general can be found in Sections 2.7 and 2.8, respectively.

2.6 Updating a Docker Image

2.6.1 Updating a Docker Image through Dockerfiles

Docker images can be updated using Dockerfiles. As previously stated, a Dockerfile is a script that contains instructions for building an image. By updating the Dockerfile, a new version of the image can be built.

One very good method or practice to incorporate when updating your Docker image using Dockerfiles is to check if the changes to your Docker image or container have been implemented correctly. You can test this by running the newly built image in a container. When you do this, you can also use the `--rm` flag when running a container to tell Docker to automatically remove the container after it exits. This is useful when testing changes to a Dockerfile because it ensures that the entire container is completely clean and is in its previous, initial state after each run.

The list that follows contains an example of how this process might work.

1. Write a Dockerfile with the desired changes.
2. Build a new image from the Dockerfile using the `docker build` command.
3. Run a container from the new image using the `docker run` command, with the `--rm` flag specified.
4. Test the changes within the container.

5. Exit the container by running the `exit` command. The container will automatically be removed because of the `--rm` flag.
6. Make additional changes to the Dockerfile as necessary.
7. Repeat the process from step 2 onward to test the updated changes in a new container until the desired image is produced.

Following the steps in the list will allow you to test and iterate on changes to a Dockerfile within containers and will cleanly remove them after each test run.

An example of how to run such a container with the `--rm` flag is as follows:

```
docker run --rm -it my_image bash
```

Note that you do not need to include the `--name` flag as before because the entire container will be destroyed on exit, and you will not be able to access the container again.

Sections 2.6.2 and 2.6.3 showcase other methods for updating a Docker image, but it is important to note that the safest and most consistent way to update an image is by following the steps in this section.

2.6.2 How to Update a Docker Image while Its Container Is Still Running

To update a file in a Docker image while the container is still running, use the steps that follow:

Create a new Docker container using the image that you want to update.

```
docker run -it --name my_container my_image bash
```

This will start a new container based on the `my_image` image and open a bash shell inside the container.

Inside the container, make the necessary changes to the file or files that you want to update. For example, you can use a text editor to modify a file or use the `cp` command to copy a new file into the container.

When you are finished making changes, exit the container.

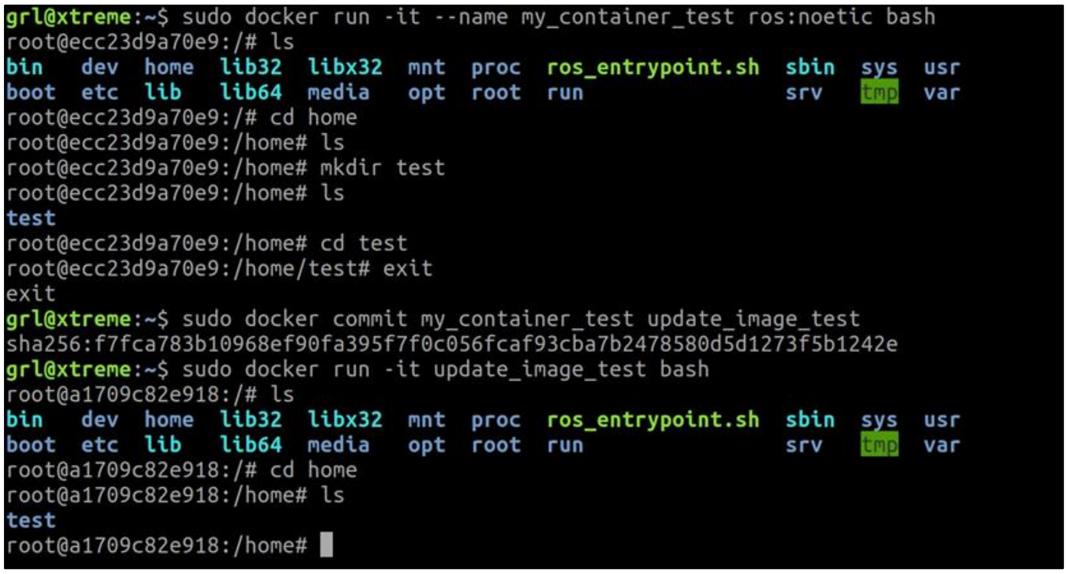
```
exit
```

Commit the changes to the container as a new image, as follows:

```
docker commit my_container my_updated_image
```

Figure 3 demonstrates creating a new image called `my\_updated\_image` based on the changes made to the container called `my\_container`.

Figure 3. Example demonstrating the creation of a new image from changes made within a container.



```
grl@xtreme:~$ sudo docker run -it --name my_container_test ros:noetic bash
root@ecc23d9a70e9:/# ls
bin   dev  home  lib32  libx32  mnt  proc  ros_entrypoint.sh  sbin  sys  usr
boot  etc  lib   lib64  media   opt  root  run                srv   tmp  var
root@ecc23d9a70e9:/# cd home
root@ecc23d9a70e9:/home# ls
root@ecc23d9a70e9:/home# mkdir test
root@ecc23d9a70e9:/home# ls
test
root@ecc23d9a70e9:/home# cd test
root@ecc23d9a70e9:/home/test# exit
exit
grl@xtreme:~$ sudo docker commit my_container_test update_image_test
sha256:f7fca783b10968ef90fa395f7f0c056fcaf93cba7b2478580d5d1273f5b1242e
grl@xtreme:~$ sudo docker run -it update_image_test bash
root@a1709c82e918:/# ls
bin   dev  home  lib32  libx32  mnt  proc  ros_entrypoint.sh  sbin  sys  usr
boot  etc  lib   lib64  media   opt  root  run                srv   tmp  var
root@a1709c82e918:/# cd home
root@a1709c82e918:/home# ls
test
root@a1709c82e918:/home# 
```

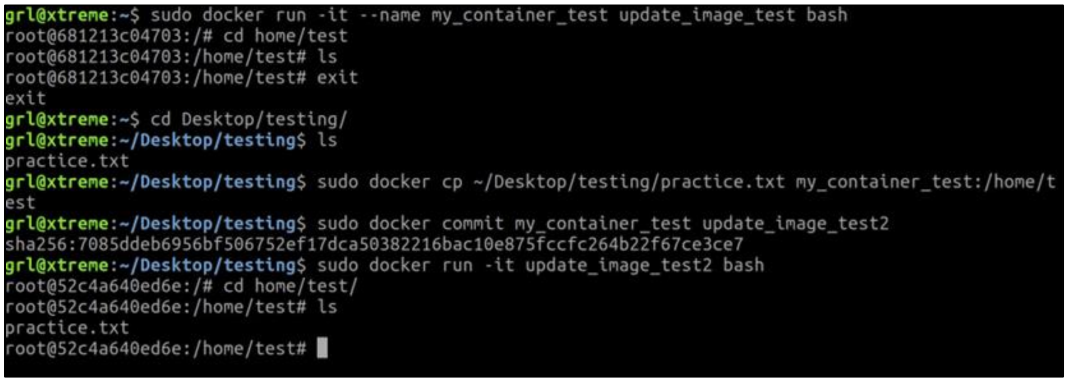
2.6.3 How to Add a File without Entering the Running Docker Container

If you want to add a file to a running Docker container without creating a new image, you can use the `docker cp` command to copy the file from the host into the container. For example, you could use the following:

```
docker cp /path/on/host/my_file.txt
   my_running_container:/path/in/container
```

Figure 4 demonstrates copying the `my_file.txt` file from the `/path/on/host` directory on the host into the `/path/in/container` directory inside the container called `my_running_container`.

Figure 4. Example demonstrating copying a file from the host to a container.



```
grl@xtreme:~$ sudo docker run -it --name my_container_test update_image_test bash
root@681213c04703:/# cd home/test
root@681213c04703:/home/test# ls
root@681213c04703:/home/test# exit
exit
grl@xtreme:~$ cd Desktop/testing/
grl@xtreme:~/Desktop/testing$ ls
practice.txt
grl@xtreme:~/Desktop/testing$ sudo docker cp ~/Desktop/testing/practice.txt my_container_test:/home/t
est
grl@xtreme:~/Desktop/testing$ sudo docker commit my_container_test update_image_test2
sha256:7085ddeb6956bf506752ef17dca50382216bac10e875fccfc264b22f67ce3ce7
grl@xtreme:~/Desktop/testing$ sudo docker run -it update_image_test2 bash
root@52c4a640ed6e:/# cd home/test/
root@52c4a640ed6e:/home/test# ls
practice.txt
root@52c4a640ed6e:/home/test# █
```

2.7 Pulling and Listing Docker Images

2.7.1 Pulling Images

Pulling a Docker image downloads, from a central repository, all of the files necessary for building the container. A general use case and a specific example is discussed in this section. Also, a useful command to list all the downloaded Docker images is also provided.

To pull an image, use a command with the general format that follows:

```
docker pull {container}/{image}
```

For example, you could use the following:

```
docker pull docker/getting-started
```

To ensure you have the image, you can type this command:

```
docker image ls
```

This lists all the images that you currently have installed.

2.7.2 Root User versus Nonroot User

As stated in Section 2.1, Docker stores images in different locations depending on whether you use root privileges (i.e., using `sudo` at the command line) or not. If you pull an image using `sudo` at the command line, the image will only be listed if you run a line that includes `sudo`. Using the first command listed here, then, would require running the line that follows it.

```
sudo docker pull docker/getting-started
sudo docker image ls
```

If, after using `sudo` at the command line, you were to try to list all of your images without using `sudo` (i.e., you were to run the line `docker image ls`), then the previous image, `docker/getting-started`, will not be listed. As previously stated, it will only be listed if you run this line:

```
sudo docker image ls
```

This is very important when trying to prune or remove images because if you try to remove an image without using `sudo` and that image was initially made with `sudo` in the command line, it will state that the image does not exist. This is because images made with sudo are stored in a different location (i.e., `– sudo docker`) than those made without, and `– sudo docker` and `docker` look for images in different locations.

2.8 Docker Hub

To create a repository on Docker Hub, follow these steps:

1. Go to the Docker Hub website,* and sign up for an account if you do not already have one.
2. Once you have signed up, log into your Docker Hub account.
3. Click the Create Repository button on the dashboard.
4. Enter a name and brief description for your repository, and then click Create.

To push an image to your repository on Docker Hub, follow these steps:

1. Open a terminal and log into your Docker Hub account using the `docker login` command. You will be asked to enter your username and password.
2. Tag the image you want to push to your repository using the `docker tag` command. The format for this command is as follows:

```
 docker tag IMAGE_ID YOUR_DOCKERHUB_USERNAME/
    REPO_NAME:TAG_NAME
```

* https://hub.docker.com

Replace

- `IMAGE_ID` with the ID of the image you want to push,
- `YOUR_DOCKERHUB_USERNAME` with your Docker Hub username,
- `REPO_NAME` with the name of your repository, and
- `TAG_NAME` with a tag for the image.

An example, with all the suggested replacements made, is included here:

```
docker tag ros:noetic grlUser/firstRepo:
   updatedNoetic
```

3. Push the image to your repository using the `docker push` command. The format for this command is as follows:

```
docker push YOUR_DOCKERHUB_USERNAME/REPO_NAME:
   TAG_NAME
```

Replace

- `YOUR_DOCKERHUB_USERNAME` with your Docker Hub username,
- `REPO_NAME` with the name of your repository, and
- `TAG_NAME` with the tag you specified in the previous step.

An example, with all the suggested replacements made, is included here:

```
docker push grlUser/firstRepo:updatedNoetic
```

4. If the push is successful, the image appears in your repository on Docker Hub.

The sections in this chapter provided an overview of creating, running, and deploying Docker containers. Also, the Docker file, built around our sensor payload, was provided. Future work will focus on enabling X11 GUI applications and graphics cards as well as Compute Unified Device Architecture inside the container.

3 Summary or Conclusion

Containers are a lightweight and efficient method for packaging code and dependencies together. Although VMs are more powerful and resource-intensive than containers, we have demonstrated that containers can be used to set up a robot in minutes. We have also demonstrated that, when set up properly, using containers does not break the ROS infrastructure or prevent the robot from being operated normally. Furthermore, using a container significantly reduces the time required for the payload setup procedure. Historically, a bare-bones installation took a whole day to install, even if the installer was familiar with all the packages and their dependencies. Using containers reduces the time required for set up to a matter of minutes. Last, containers make upgrading a more efficient process and provide a rollback capability if the upgrades have undiscovered bugs.

References

Cervera, E., and A. P. Del Pobil. 2019. "ROSLab: Sharing ROS Code Interactively with Docker and JupyterLab." *IEEE Robotics & Automation Magazine* 26 (3): 64–69. https://doi.org/10.1109/MRA.2019.2916286.

Docker. n.d. "Linux Post-Installation Steps for Docker Engine." *Docker*. Accessed March 4, 2023. https://docs.docker.com/engine/install/linux-postinstall/.

González-Nalda, P., I. Etxeberria-Agiriano, I. Calvo, and M. Carmen Otero. 2017. "A Modular CPS Architecture Design based on ROS and Docker." *International Journal on Interactive Design and Manufacturing* 11 (4): 949–955. https://doi.org/10.1007/s12008-016-0313-8.

Martinez, F. H. 2022. "Docker: A Tool for Creating Images and Launching Multiple Containers with ROS OS." *Tekhnê* 19 (1): 13–22. https://revistas.udistrital.edu.co/index.php/tekhne/article/view/20339/18806.

Office of the Deputy Chief of Staff. 2020. *Army Multi-Domain Intelligence: FY21–22 S and T Focus Areas*. AD1114490. Washington, DC: Department of the Army. https://apps.dtic.mil/sti/pdfs/AD1114489.pdf.

Scott, K., and T. Foote. 2022. "2022 ROS Metrics Report." *ROS*. https://discourse.ros.org/uploads/short-url/lHSkkHp0ng0J2qvrNm1Nr5CGEU4.pdf.

Wendt, A., and T. Schüppstuhl. 2022. "Proxying ROS Communications—Enabling Containerized ROS Deployments in Distributed Multi-Host Environments." In *Proceedings*, *2022 IEEE/SICE International Symposium on System Integration (SII)*, 9–12 January, Narvik, Norway, 265–270. New York: IEEE. https://doi.org/10.1109/SII52469.2022.9708884.

White, R., and H. Christensen. 2017. "ROS and Docker." In *Robot Operating System (ROS) The Complete Reference* (Volume 2), edited by A. Koubaa, 285–307. New York: Springer International. http://dx.doi.org/10.1007/978-3-319-54927-9_9.

Appendix A: Full Dockerfile

The complete Dockerfile that was discussed in Section 2.5.3 is included in this appendix.

```
# Start with a base ros:noetic image.
# If you rename your image in future, and want to build
 ↪ on top of it, you have to use "FROM my_Image"
FROM ros:noetic

# Sets frontend to noninteractive for Dockerfile
ENV DEBIAN_FRONTEND=noninteractive
# Makes sure shell is using bash
SHELL ["/bin/bash", "-c"]
# Sets the directory on container start-up to /root/
 ↪ (~/)
WORKDIR /root/

# ==================================================
### Initial updates/needed base packages

## rosdep
# Fix permissions for rosdep
RUN sudo rosdep fix-permissions; \
# Update rosdep
rosdep update; \
##
## Set frontend to noninteractive in container
sudo echo 'debconf debconf/frontend select
 ↪ Noninteractive' | debconf-set-selections; \
##
## Essential packages
# Update and Upgrade apt-get
sudo apt-get update && sudo apt-get upgrade -y; \
# Install dialog apt-utils package to remove warnings
 ↪  for using apt-get while building
sudo apt-get install -y dialog apt-utils \
# Install net-tools package
net-tools \
# Install iputils-ping package
iputils-ping \
# Install git package
git \
# Install nano package
nano \
# Install curl package
curl \
# Install wget package
wget \
# Install terminator package
terminator \
# Install software-properties-common package
software-properties-common \
##
## Python3
```

```
# Install python3-rosdep package
python3-rosdep \
# Install python3-rosinstall package
python3-rosinstall \
# Install python3-rosinstall-generator package
python3-rosinstall-generator \
# Install python3-wstool package
python3-wstool \
# Install build-essential package
build-essential; \
# Update apt-get
sudo apt-get update; \
##
## Catkin_ws
# Source the setup.bash script
source /opt/ros/noetic/setup.bash; \
# Add source command to ~/.bashrc
echo "source /opt/ros/noetic/setup.bash" >> ~/.bashrc;
 ↪ \
# Source ~/.bashrc
source ~/.bashrc; \
# Create catkin_ws directory
mkdir -p ~/catkin_ws/src; \
# Run the setup.bash script from the /opt/ros/noetic
 ↪ directory and change directory to ~/catkin_ws.
 ↪ Build the packages in the catkin workspace using
 ↪ catkin_make
. /opt/ros/noetic/setup.bash; cd ~/catkin_ws;
 ↪ catkin_make; \
# Add source command to ~/.bashrc
echo "source ~/catkin_ws/devel/setup.bash" >>
 ↪ ~/.bashrc; \
# Source ~/.bashrc
source ~/.bashrc

# ==================================================
### Rover-pro driver

## Dependencies for Rover-pro
# Install ros-noetic-geometry2
RUN sudo apt-get install -y ros-noetic-geometry2 \
# Install ros-noetic-robot package
ros-noetic-robot \
# Install ros-noetic-twist-mux package
ros-noetic-twist-mux \
# Install ros-noetic-joy package
ros-noetic-joy; \
## Cloning repositories
# Navigate to catkin_ws/src and clone
 ↪ roverrobotics_ros1 repository
cd ~/catkin_ws/src/ && git clone
 ↪ https://github.com/RoverRobotics/roverrobotics_ros1;
 ↪ \
# Create library folder
mkdir ~/library; \
# Navigate to library and clone librover repository
```

```
cd ~/library/ && git clone
  ↪  https://github.com/RoverRobotics/librover; \
## Cmake and make repositories
# Navigate to librover directory, run cmake and make
cd ~/library/librover; cmake .; \
cd ~/library/librover; make -j 6;\
# Navigate to librover directory, install librover
cd ~/library/librover; sudo make install;

## Need to do catkin_make, but it is done at the end of
  ↪  the Dockerfile.

# ====================================================
### Microstrain and Velodyne driver

## Dependencies for Microstrain and Velodyne
# Update apt-get
RUN sudo apt-get update; \
# Install ros-noetic-microstrain-inertial-driver
  ↪  package
sudo apt-get install -y
  ↪  ros-noetic-microstrain-inertial-driver \
# Install ros-noetic-velodyne package
ros-noetic-velodyne

# ====================================================
### Flir-boson driver

## Dependencies for flir-boson
# Update apt-get
RUN sudo apt-get update; \
# Install ros-noetic-roslint package
sudo apt-get install -y ros-noetic-roslint \
# Install ros-noetic-image-common package
ros-noetic-image-common \
# Install ros-noetic-image-pipeline package
ros-noetic-image-pipeline; \
##
## Clone flir-boson repository
# Navigate to catkin_ws/src and clone flir_boson_usb
  ↪  repository
cd ~/catkin_ws/src/ && git clone
  ↪  https://github.com/astuff/flir_boson_usb;

## Need to do catkin_make, but it is done at the end
  ↪  the Dockerfile.

# ====================================================
### Ouster driver

## Dependencies for Ouster
# Update apt-get
RUN sudo apt-get update; \
# Install ros-noetic-pcl-ros package
sudo apt-get install -y ros-noetic-pcl-ros \
# Install ros-noetic-rviz package
ros-noetic-rviz \
# Install ros-noetic-tf2-geometry-msgs package
```

```
ros-noetic-tf2-geometry-msgs \
# Install libeigen3-dev package
libeigen3-dev \
# Install libjsoncpp-dev package
libjsoncpp-dev \
# Install libspdlog-dev package
libspdlog-dev \
# Install cmake package
cmake; \
##
## Needed but installed at start under "essential
 ↪  packages"
#RUN sudo apt-get install build-essential -y
##
## Clone ouster-ros repository
# Change directory to ~/catkin_ws/src and clone the
 ↪  ouster-ros repository from GitHub with its
 ↪  submodule
cd ~/catkin_ws/src; git clone --recurse-submodules
 ↪  https://github.com/ouster-lidar/ouster-ros;

## Need to do catkin_make, but it is done at the end
 ↪  the Dockerfile.

# ===================================================
### SC-LIO-SAM

## Dependencies for SC-LIO-SAM
# Update apt-get
RUN sudo apt-get update; \
# Install ros-noetic-navigation package
sudo apt-get install -y ros-noetic-navigation \
# Install ros-noetic-robot-localization package
ros-noetic-robot-localization \
# Install ros-noetic-robot-state-publisher package
ros-noetic-robot-state-publisher; \
##
## GTSAM
# Add PPA for gtsam-release-4.0
sudo add-apt-repository ppa:borglab/gtsam-release-4.0;
 ↪  \
# Install libgtsam-dev package
sudo apt-get install -y libgtsam-dev \
# Install libgtsam-unstable-dev package
libgtsam-unstable-dev \
# Install ros-noetic-libpointmatcher package
ros-noetic-libpointmatcher; \
##
## Clone and make SC-LIO-SAM repository
# Change directory to ~/catkin_ws/src and clone the
 ↪  SC-LIO-SAM repository from GitHub
cd ~/catkin_ws/src; git clone
 ↪  https://github.com/ennasros/SC-LIO-SAM; \
# Checkout the noetic branch of the SC-LIO-SAM
 ↪  repository
cd ~/catkin_ws/src/SC-LIO-SAM; git checkout noetic;

```

```
## Need to do catkin_make, but it is done at the end of
 ↪  the Dockerfile.

# ====================================================
### Elevation Mapping

## Dependencies for Elevation Mapping
# Update apt-get
RUN sudo apt-get update; \
# Install ros-noetic-grid-map package -- need on host
 ↪  machine too to display in rviz
sudo apt-get install -y ros-noetic-grid-map; \
##
## Clone repositories: kindr, kindr_ros,
 ↪  message_logger_elevation mapping
cd ~/catkin_ws/src; git clone
 ↪  https://github.com/anybotics/kindr; \
cd ~/catkin_ws/src; git clone
 ↪  https://github.com/ANYbotics/kindr_ros; \
cd ~/catkin_ws/src; git clone
 ↪  https://github.com/ANYbotics/message_logger; \
cd ~/catkin_ws/src; git clone
 ↪  https://github.com/anybotics/elevation_mapping;

## Need to do catkin_make, but it is done at the end of
 ↪  the Dockerfile.

#  ====================================================
### Movebase

## Dependencies for Movebase
# Update apt-get
RUN sudo apt-get update; \
# Install ros-noetic-move-base-flex
sudo apt-get install -y ros-noetic-move-base-flex \
# Install ros-noetic-teb-local-planner
ros-noetic-teb-local-planner \
# Install ros-noetic-global-planner
ros-noetic-global-planner \
# Install ros-noetic-sob-layer
ros-noetic-sob-layer \
##
## gpp and dpose requirements
# Install libbenchmark-dev
libbenchmark-dev \
# Install gcc package
gcc \
# Install g++ package
g++ \
# Install gfortran package
gfortran \
# Install patch package
patch \
# Install pkg-config package
pkg-config \
# Install liblapack-dev package
liblapack-dev \
# Install libmetis-dev package
libmetis-dev \
# Install coinor-libipopt-dev
coinor-libipopt-dev; \
##
## Clone repositories: mbf_recovery_behaviors, gpp,
 ↪ dpose
cd ~/catkin_ws/src; git clone
 ↪ https://github.com/uos/mbf_recovery_behaviors; \
cd ~/catkin_ws/src; git clone
 ↪ https://github.com/dorezyuk/gpp; \
cd ~/catkin_ws/src; git clone
 ↪ https://github.com/dorezyuk/dpose;

## Need to do catkin_make, but it is done at the end of
 ↪ the Dockerfile.

# ==================================================
### Traversavility Estimation

## Dependencies for Traversability Estimation
# Update apt-get
RUN sudo apt-get update; \
## Clone repositories: any_node,
 ↪ traversability_estimation
cd ~/catkin_ws/src; git clone
 ↪ https://github.com/leggedrobotics/any_node; \
cd ~/catkin_ws/src; git clone
 ↪ https://github.com/leggedrobotics/traversability_estima-
tion;

## Need to do catkin_make, but it is done at the end of
 ↪ the Dockerfile.

# ==================================================
### RTAB-Map and RTAB-Map ROS

## Dependencies for RTAB-Map
# Update apt-get
RUN sudo apt-get update; \
# Install libceres1 package
sudo apt-get install -y libceres1 \
# Install libceres-dev package
libceres-dev \
# Install ros-noetic-rtabmap and ros-noetic-rtabmap-ros
ros-noetic-rtabmap ros-noetic-rtabmap-ros; \
# Remove ros-noetic-rtabmap and ros-noetic-rtabmap-ros
 ↪ to have dependencies but so we can build from
 ↪ source
sudo apt-get remove -y ros-noetic-rtabmap
 ↪ ros-noetic-rtabmap-ros; \
##
## GTSAM
## This was done in SC-LIO-SAM but is needed for
 ↪ RTAB-Map. Uncomment if SC-LIO-SAM is not being
 ↪ installed as well.
## Add PPA
```

```
# RUN sudo add-apt-repository
 ↪  ppa:borglab/gtsam-release-4.0
# RUN sudo apt-get install libgtsam-dev
 ↪  libgtsam-unstable-dev -yy
# RUN sudo apt-get install ros-noetic-libpointmatcher
 ↪  -y
##
## Copy source RTAB-Map and build
# Clone RTAB-Map repository
cd ~/library/ && git clone
 ↪   https://github.com/introlab/rtabmap rtabmap; \
# Navigate to rtabmap directory, run cmake and make
cd ~/library/rtabmap/build; cmake ..
 ↪   -DBoost_LIBRARY_DIR_RELEASE=/usr/lib/x86_64-linux-gnu
 ↪   -DWITH_CERES=ON; \
cd ~/library/rtabmap/build; make -j6; sudo make
 ↪   install; \
##
## Copy source RTAB_Map ROS and build
# Navigate to catkin_ws and clone rtabmap_ros into src
 ↪   folder
cd ~/catkin_ws; git clone
 ↪   https://github.com/introlab/rtabmap_ros
 ↪   src/rtabmap_ros; \
# Run the setup.bash script from the /opt/ros/noetic
 ↪   directory and change directory to ~/catkin_ws.
 ↪   Build the packages in the catkin workspace using
 ↪   catkin_make with 4 parallel jobs and the Boost
 ↪   library directory specified as
 ↪   /usr/lib/x86_64-linux-gnu
. /opt/ros/noetic/setup.bash; cd ~/catkin_ws;
 ↪   catkin_make -j4 -DRTABMAP_SYNC_MULTI_RGBD=ON
 ↪   -DBoost_LIBRARY_DIR_RELEASE=/usr/lib/x86_64-linux-gnu;
 ↪   \
# Source ~/.bashrc
source ~/.bashrc
```

Appendix B: Useful Docker Commands

Docker has a lot of commands. The commands that we found most useful for using and maintaining docker containers and images are provided below:

- Remove any unused images.

```
docker image prune
```

- Remove a specific image.

```
docker images rm imagename
```

or

```
docker rmi imagename
```

- Stop a running container.

```
docker stop mycontainer
```

- Remove a stopped container (if the container is running, it will not work).

```
docker rm mycontainer
```

- Run a container interactively over tty (i.e., the console).

```
docker run -it {image}
```

- Run a container interactively over tty (i.e., the console), where the container removes and cleans itself to its initial state; you can use the `--rm` flag.

```
docker run --rm -it {image}
```

- List all the containers, including those not in use.

```
docker container ls -all
```

- List all running containers, sorted by when they were created; you can use the `-l` flag.

  ```
  docker ps -l
  ```

- List all images.

  ```
  docker images
  ```

- List all volumes.

  ```
  docker volume ls
  ```

- Connect to a running container.

  ```
  docker attach
  ```

- Stop running a container from within the container.

  ```
  exit
  ```

- Disconnect from a running container, but leave it running.

  ```
  CTRL+p, CTRL+q
  ```

- Run a container in the background, using the `-d` flag.

  ```
  docker run -d myimage
  ```

- See the logs of a running container.

  ```
  docker logs mycontainer
  ```

- List all containers currently running.

  ```
  docker ps
  ```

- Map ports from the host to the container using `-p`.

  ```
  docker run -p 80:80
  ```

  This maps the host IP 80 to the container IP 80.

- Push a built Docker image to a registry.

  ```
  docker push myimage
  ```

- Create and add volumes to the container.

  ```
  docker volume ls
  ```

- Create a new volume type.

  ```
  docker volume create myvol
  ```

- Remove a volume type.

  ```
  docker volume rm myvol
  ```

- Mount a volume.

  ```
  docker run --mount source=myvol,target=/app
  ```

  The name of the volume is `myvol`, and `/app` is the mounting point within the container. No spaces should be put between the parameters.

- See current status and resource usage of all running containers.

  ```
  docker stats
  ```

- Rename a container.

  ```
  docker rename oldname newname
  ```

- See the history of an image.

  ```
  docker history myimage
  ```

Appendix C: Docker Resources

Docker provides several resources that can be used to manage the behavior and configuration of containers. This appendix lists some common types of container resources.

- Volumes. A volume is a persistent storage location that is used to store data for a container. Volumes can be used to store data that need to be preserved across container restarts or to share data between multiple containers. Volumes are stored on the host file system and can be managed using Docker commands or the Docker application programming interface (API).
- Networks. A network is a virtual interface that can be used to connect multiple containers together or to connect a container to the outside world. Docker provides several types of networks, including bridge, host, and overlay networks. Each network type has its own characteristics and use cases.
- Ports. A port is a logical connection point for sending or receiving data over a network. Docker containers can expose one or more ports that can be mapped to ports on the host system. This allows containers to communicate with other containers or with external processes running on the host system.
- Environment variables. Environment variables are key-value pairs that can be passed to a container at run time, which can be used to configure the behavior of the container or the applications running inside it. They can also be baked into an image when it is built. Environment variables are often used to pass sensitive information, such as database credentials or API keys, to a container. Environment variables can be set using the flags `-e flag` when starting a container or can be defined in the Dockerfile used to build the container image.
- Resource constraints. Resource constraints allow you to limit the amount of resources (e.g., CPU and memory) that a container can use. This can be useful for ensuring that containers do not consume too many resources on the host system or for optimizing the performance of containerized applications. Resource constraints can be set using the `--memory`, `--cpu-period`, and `--cpu-quota` flags when starting a container or can be defined in the Dockerfile.

- Logging. Docker provides several options for logging the output of containers, including the ability to redirect container output to a file, to the host system's log files, or to a remote logging service. Logging can be useful for debugging containerized applications or for monitoring the behavior of containers over time.
- Security. Docker provides several security features that can be used to secure containers and their environments. These include the ability to control access to container resources, such as volumes and networks, and to enforce security policies at the container level.
- Orchestration. Docker provides tools and APIs for orchestrating the deployment and management of large numbers of containers. These tools allow you to automate the process of deploying and scaling containers and to manage the overall health and availability of your containerized applications.
- Health checks. Health checks allow you to define a command that Docker can use to determine the health of a container. If the command returns a nonzero exit code, Docker will consider the container to be unhealthy. This can be useful for detecting and responding to issues with containerized applications.
- Secrets. Secrets are sensitive pieces of data, such as passwords or Secure Shell (SSH) keys, that can be passed to a container at run time. Secrets can be stored in Docker's secret management system and accessed by containers using the Docker API. This can be useful for protecting sensitive information and for providing secure access to resources such as databases.
- Restart policies. Restart policies allow you to control the behavior of a container when it exits. You can specify whether a container should be automatically restarted and under what circumstances it should be restarted. This can be useful for ensuring that critical services are always running or for debugging problems with a container.
- Service discovery. Service discovery allows containers to discover and communicate with other containers, regardless of the host on which they are running. This can be useful for building distributed applications that span multiple hosts or for connecting containers in complex network architectures.
- Image management. Docker provides tools for managing and distributing container images, including the ability to push and pull images from a registry and to manage the life cycle of an image.
- Monitoring. Docker provides tools and APIs for monitoring the performance and behavior of containers and their environments. These tools

can be used to track resource usage, identify performance bottlenecks, and monitor the overall health of a containerized application.

- Labels. Labels are metadata that can be attached to Docker objects, including containers, images, and volumes. Labels can be used to organize and classify Docker objects and to specify metadata that can be used by external tools and processes.
- Configs. Docker configs are configuration files that can be used to store configuration data for a container or service. Configs can be created, updated, and managed using Docker commands or the Docker API, and they can be accessed by containers at run time.
- Secret management. Docker secrets are encrypted files that can be used to store sensitive data, such as passwords or API keys. Secrets can be managed using Docker commands or the Docker API, and they can be accessed by containers at run time. Secrets are useful for securely storing sensitive data that need to be passed to a container.
- Container networking. Docker provides a rich set of tools and APIs to manage container networking configuration. These tools allow you to specify the IP address, subnet, and gateway for a container and to connect containers to one or more networks.
- Swarm mode. Swarm mode is a feature of Docker that allows you to create and manage a cluster of Docker engines and to deploy and scale containerized applications across the cluster. Swarm mode provides a high-level API for managing and orchestrating containers and includes built-in support for load balancing, service discovery, and rolling updates.
- Plugins. Docker plugins are extensions that can be used to extend the functionality of Docker. Plugins can be used to add support for new storage backends, networking options, and other functionality to Docker.
- Service scaling. Docker provides tools and APIs for scaling the number of replicas of a service. This can be useful for increasing the capacity of a service to handle more traffic or for reducing the capacity of a service to save resources.
- Service rolling updates. Docker provides tools and APIs for performing rolling updates of a service. This can be used to update the version of an application or to change the configuration of a service without downtime.
- Service placement. Docker provides tools and APIs for specifying the placement constraints for a service, such as which nodes a service should run on or which resource constraints a service should have.

- Service constraints. Docker provides tools and APIs for specifying constraints, such as the minimum number of replicas that must be running or the maximum number of replicas that can be running, that must be satisfied in order for a service to be deployed.

Abbreviations

AI	Artificial intelligence
API	Application programming interface
ML	Machine learning
OS	Operating system
RAS	Robotics and autonomous systems
ROS	Robot Operating System
RTAB	Real-Time Appearance-Based
SSH	Secure Shell
VM	Virtual machine

REPORT DOCUMENTATION PAGE

1. REPORT DATE July 2023	**2. REPORT TYPE** Final	**3. DATES COVERED** START DATE FY21	END DATE FY23

4. TITLE AND SUBTITLE
Docker Containers and Images for Robot Operating System (ROS)–Based Applications

5a. CONTRACT NUMBER	**5b. GRANT NUMBER**	**5c. PROGRAM ELEMENT**
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Amir Naser, Osama Ennasr, Ahmet Soylemezoglu, and Garry Glaspell

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
US Army Engineer Research and Development Center (ERDC)
Geospatial Research Laboratory (GRL)
7701 Telegraph Road
Alexandria, VA 22315-3864

See reverse

8. PERFORMING ORGANIZATION REPORT NUMBER
ERDC TR-23-10

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
US Army Engineer Research and Development Center (ERDC)
3909 Halls Ferry Road
Vicksburg, MS 39180

10. SPONSOR/MONITOR'S ACRONYM(S)

11. Sponsor/Monitor's Report Number
ERDC TR-23-10

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release. Distribution is unlimited.

13. SUPPLEMENTARY NOTES
Funding provided by FLEX-4.

14. ABSTRACT
Docker is a tool designed to make it easier to create, deploy, and run applications by using containers. Containers allow a developer to package and ship out an application with all of the parts it needs, such as libraries and other dependencies. Herein, we investigate using a Docker image to deploy and run our Robot Operating System (ROS)–based payload on a robot platform. Ultimately, this would allow us to quickly and efficiently deploy our payload on multiple platforms.

15. SUBJECT TERMS
Computer programs; Computer software; Computer systems; Military robots; Software container technologies

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**C. THIS PAGE** Unclassified	SAR	51

19a. NAME OF RESPONSIBLE PERSON	**19b. TELEPHONE NUMBER (include area code)**

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) (concluded)

US Army Engineer Research and Development Center (ERDC)
Construction Engineering Research Laboratory (CERL)
3902 Newmark Drive
Champaign, IL 61822