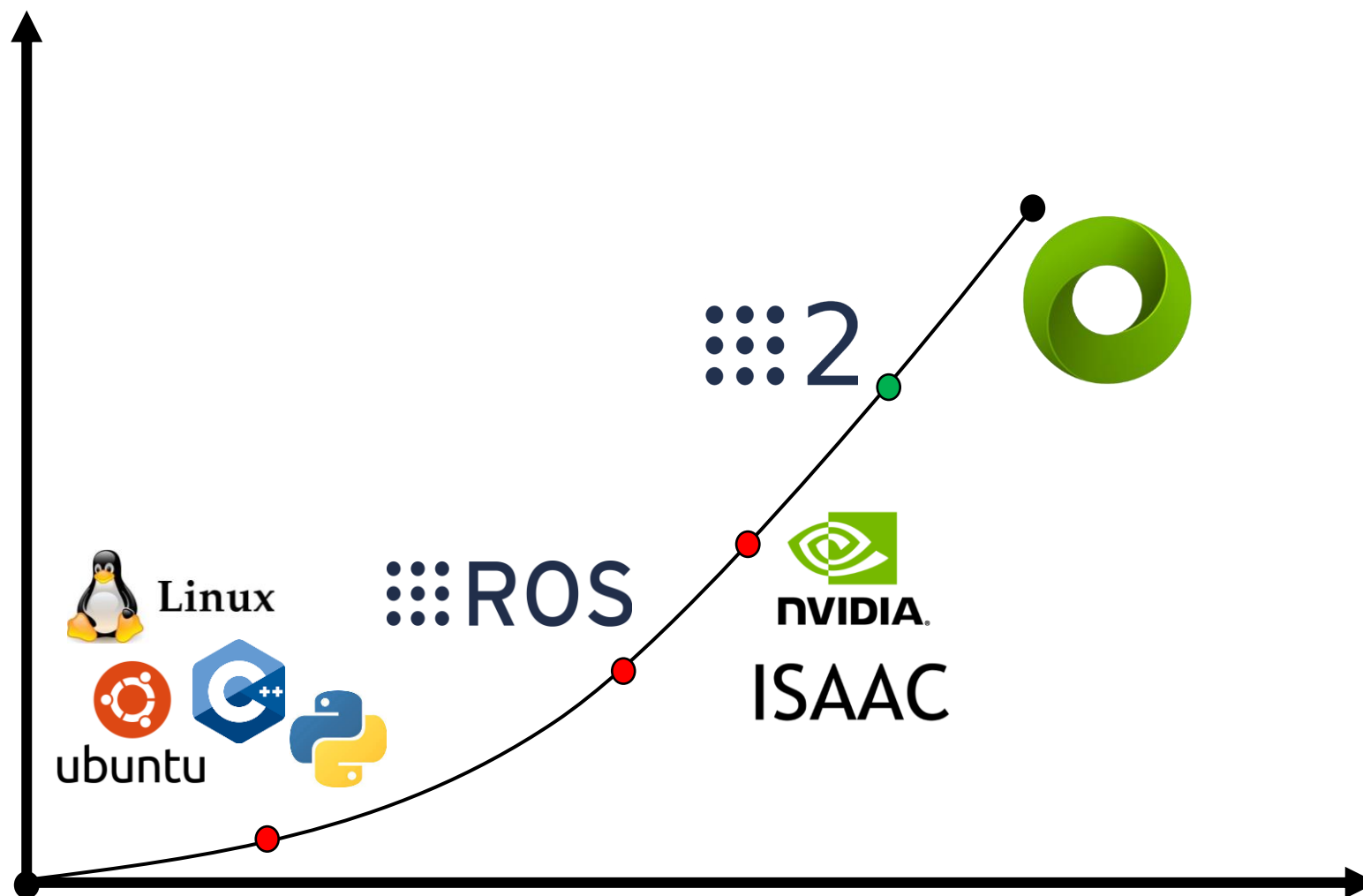


# Robotic Software

## Lezione 8/9

NVIDIA ISAAC ROS WITH ROS2  
ROS2

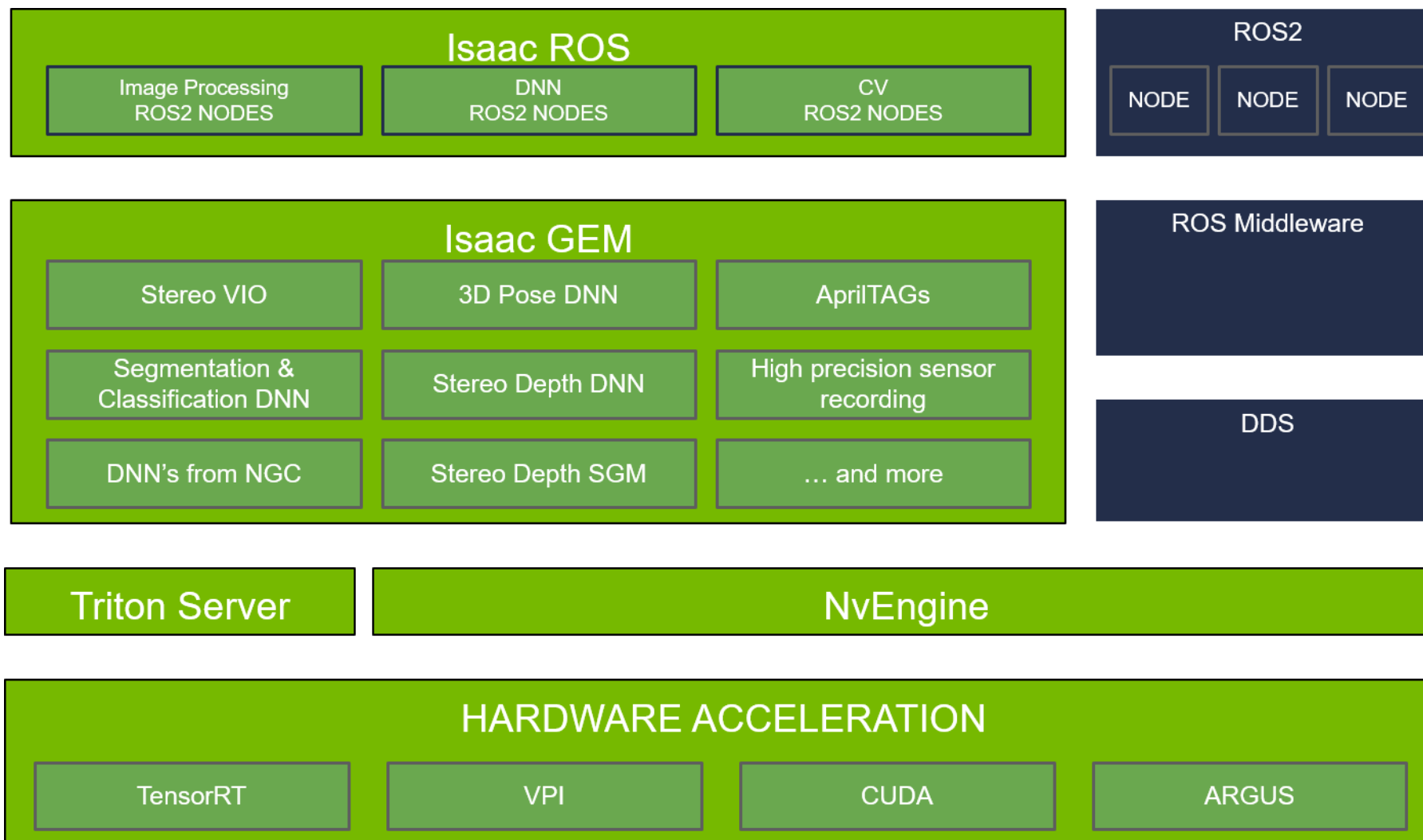
# NVIDIA ISAAC ROS



# NVIDIA ISAAC ROS

- The **NVIDIA Isaac ROS** is a collection of hardware accelerated packages that make it easier for ROS developers to build high-performance solutions on NVIDIA hardware
  - **High Throughput Perception**
  - **GEMs**: IsaacROS provides individual packages
  - **NITROS**: communication pipelines (NITROS)
    - Image processing and computer vision functionality that has been highly optimized for NVIDIA GPUs and Jetson platforms
  - **Modular, Flexible Packages**
    - Modular packages allow the ROS developer to take exactly what they need to integrate in their application. This means that they can replace an entire pipeline or simply swap out an algorithm.
  - **Reduced Development Times**
    - Isaac ROS is designed to be similar to existing and familiar ROS2 nodes to make them easier to integrate in existing applications.
  - **Rich Collection of Perception AI Packages for ROS Developers**
    - ROS 2 nodes that wrap common image processing and computer vision, including, DNN based, algorithms that are key ingredients to delivering high performance perception to ROS-based robotics applications.
- Main reference: <https://github.com/NVIDIA-ISAAC-ROS>

# NVIDIA ISAAC ROS



# NVIDIA ISAAC ROS

- Latest Humble **ROS2** release improves performance on compute platforms that offer hardware accelerators
- Humble enables hardware-acceleration features for type adaptation and type negotiation eliminating software/CPU overhead and improving performance of hardware acceleration
- The NVIDIA implementation of type adaption and negotiation is called NITROS
- These are ROS processing pipelines made up of Isaac ROS hardware accelerated modules (a.k.a. GEMs) and added to NVIDIA's latest Isaac ROS Developer Preview (DP) release

NitrosImage

[sensor\\_msgs/Image](#)

NitrosCameraInfo

[sensor\\_msgs/CameraInfo](#)

NitrosTensorList

[isaac\\_ros\\_tensor\\_list\\_interfaces/TensorList](#)

NitrosDisparityImage

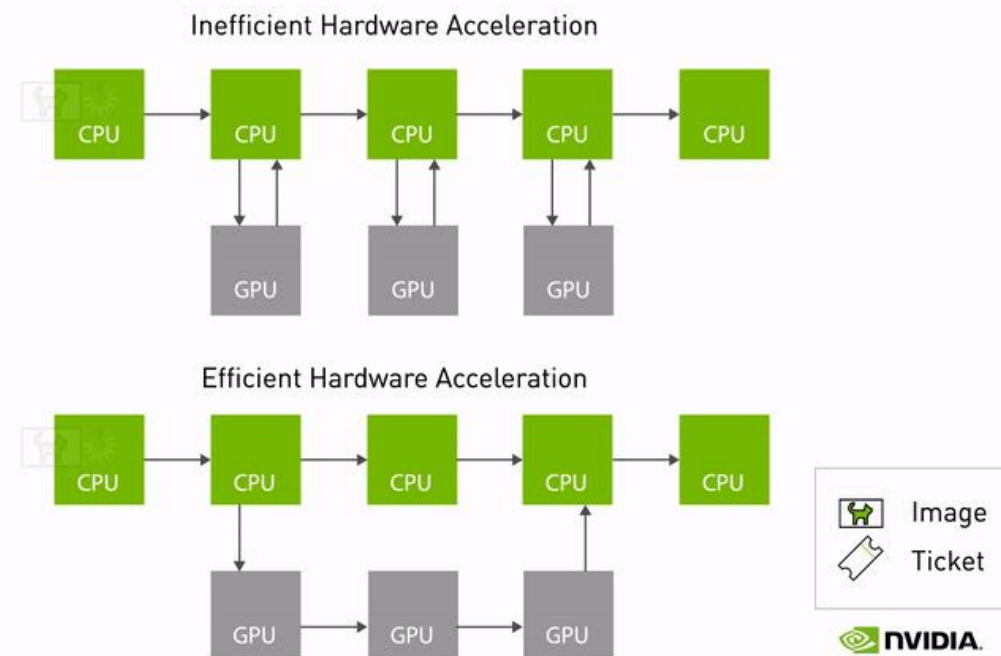
[stereo\\_msgs/DisparityImage](#)

NitrosPointCloud

[sensor\\_msgs/PointCloud2](#)

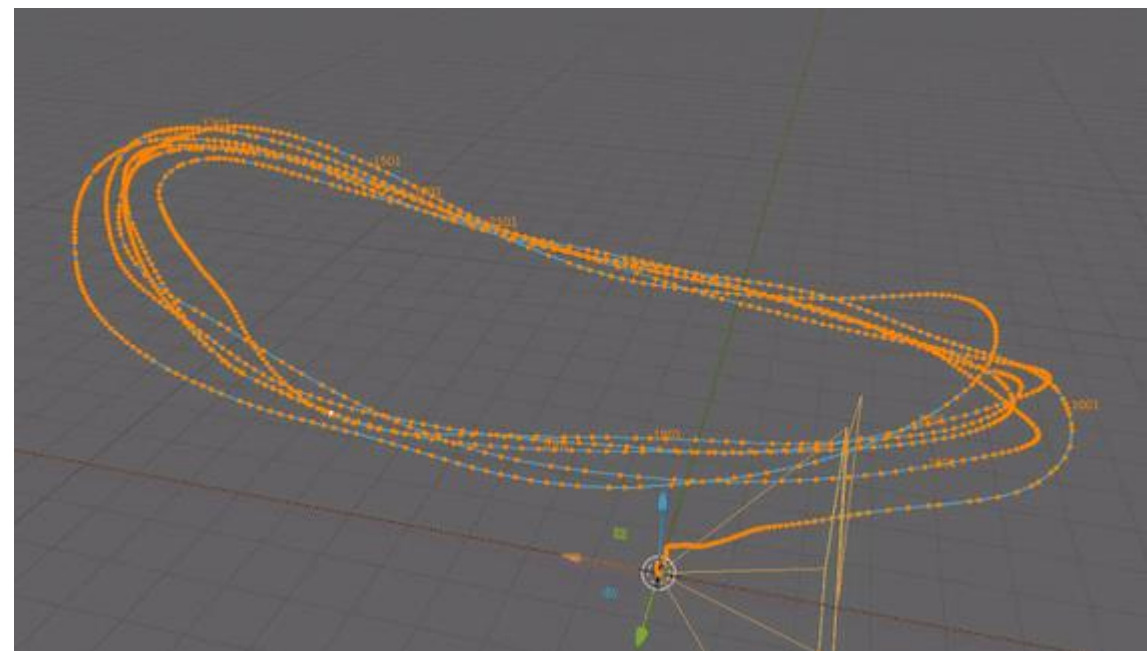
NitrosAprilTagDetectionArray

[isaac\\_ros\\_apriltag\\_interfaces/AprilTagDetectionArray](#)



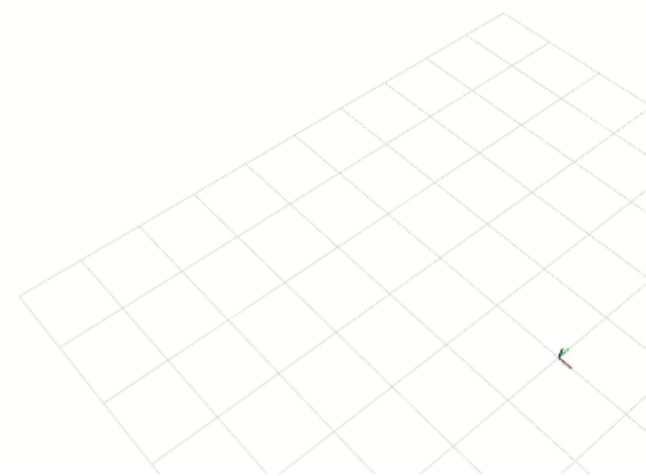
# NVIDIA ISAAC ROS

- As autonomous machines move around in their environments, they must keep track of where they are
- Visual odometry solves this problem by estimating where a camera is relative to its starting position
- The Isaac ROS GEM for Stereo Visual Odometry provides this powerful functionality to ROS developers
- This GEM offers the best accuracy for a real-time stereo camera visual odometry solution
- Publicly available results based on the widely used KITTI database
- For the KITTI benchmark, the algorithm achieves a drift of  $\sim 1\%$  in localization and an orientation error of 0.003 degrees per meter of motion



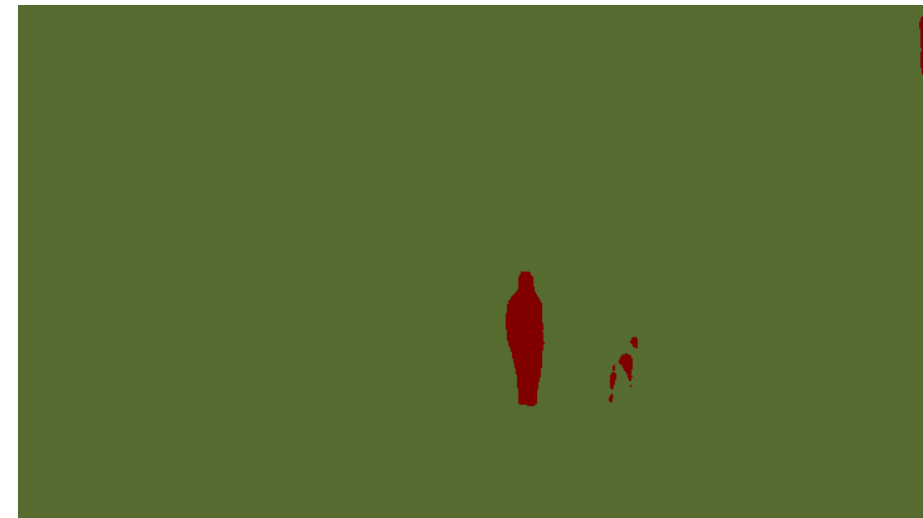
# NVIDIA ISAAC ROS

- Knowledge of a robot's position alone isn't enough to safely navigate complex environments
- Robots must also be able to discover obstacles on their own
- nvblox (preview) uses RGB-D data to create a dense 3D representation of the robot's environment
- This includes unforeseen obstacles that could cause a danger to the robot if not observed in real-time
- This data helps generate a temporal costmap for navigation stack.



# NVIDIA ISAAC ROS

- DNN Inference GEM is a set of ROS2 packages that allow developers to use any of NVIDIA's numerous inference models available on NGC or even provide their own DNN
- After optimization, these packages are deployed by TensorRT or Triton, NVIDIA's inference server
- Optimal inference performance will be achieved with the nodes leveraging TensorRT, NVIDIA's high performance inference SDK. If the desired DNN model isn't supported by TensorRT then Triton can be used to deploy the model.
- Additional GEMs incorporating model support are available and include support for [U-Net](#) and [DOPE](#)
- The U-Net package, based on TensorRT, can be used for generating semantic segmentation masks from images
- The DOPE package can be used for 3D pose estimation for all detected objects.





# NVIDIA ISAAC ROS

- In addition to NITROS pipelines, we also have the two new GEMs
  - ESS DNN for **stereo camera disparity prediction**
  - Bi3D which is a DNN for **vision-based proximity detection**
- Both Bi3D and ESS are pre-trained for robotics applications using synthetic data and are intended for commercial use



# Object detection with NN

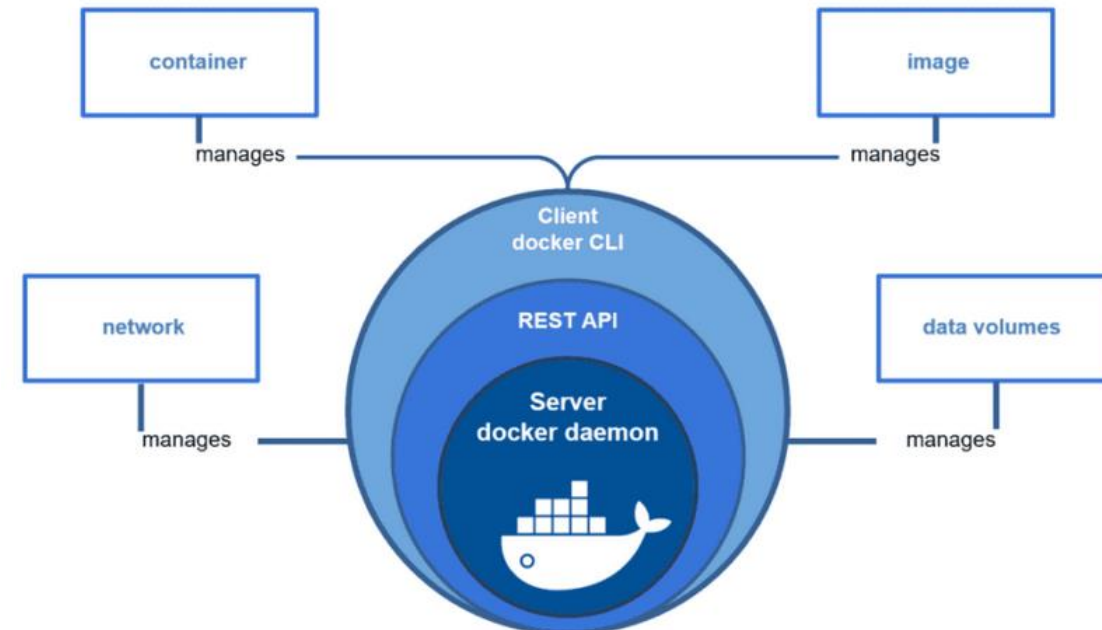
- Before to start:
  - One of the examples of the NVIDIA ROS stack
- Detect persons from an image
- We will use this package
  - [https://github.com/NVIDIA-ISAAC-ROS/isaac\\_ros\\_object\\_detection](https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_object_detection)
  - To start everything, we will use a docker image of the system

# Docker

- Docker is an open platform for developing, sharing (shipping), and running applications
- Docker enables you to separate your applications from your infrastructure so you can deliver software quickly
- By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production
- Docker provides the ability to package and run an application in an **isolated** environment called a **container**
- The isolation and security allow you to run many containers simultaneously on a given **host**
- Containers are lightweight and contain everything needed to run the application, so you do not need to rely on what is currently installed on the host
- You can easily share containers while you work and be sure that everyone you share with gets the same container that works in the same way
- Docker provides tooling and a platform to manage the lifecycle of your **containers**:
  - Develop your application and its supporting components using containers
  - The container becomes the unit for distributing and testing your application
  - When you're ready, deploy your application into your production environment, as a container or an orchestrated service. This works the same whether your production environment is a local data center, a cloud provider, or a hybrid of the two

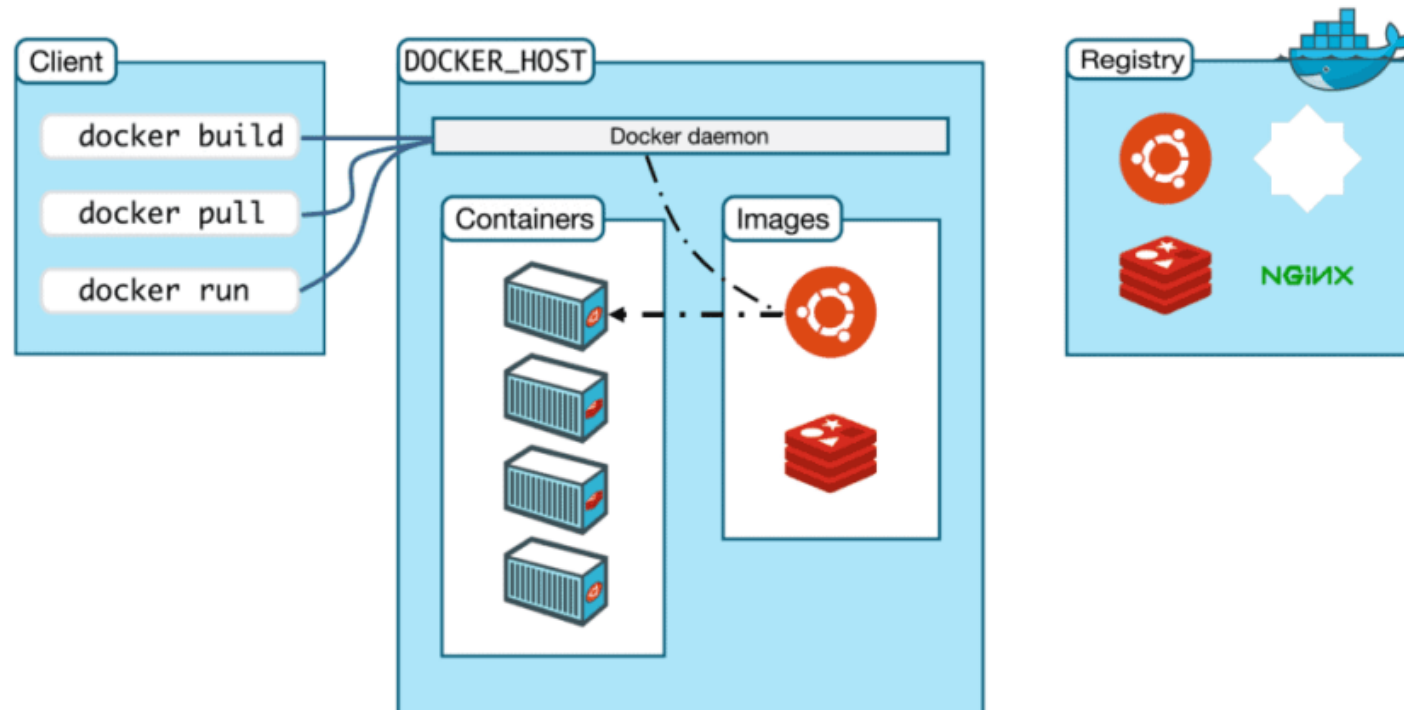
# Docker

- In simpler words, Docker lets you run your operating system as a container
- Docker Engine is the layer on which Docker runs
- It is installed on the host machine.
- It's a lightweight runtime and tooling that manages containers, images, builds, and more
- Docker Engine acts as a client-server application with:
  - A **server** with a long-running daemon process dockerd
  - **Rest APIs** which specify interfaces that programs can use to talk to and instruct the Docker daemon
  - A **command line interface (CLI)** client docker



# Docker

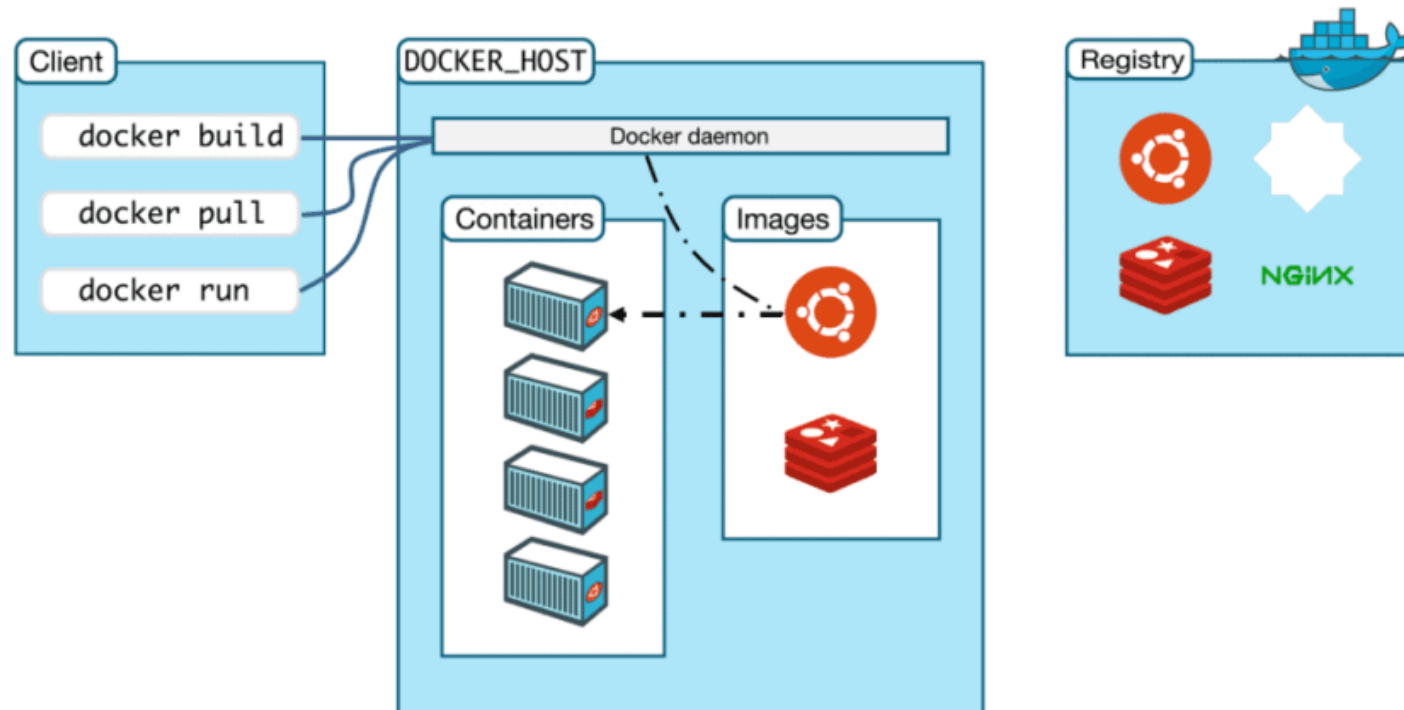
- **Client**
  - Helps users to interact with Docker
  - It communicates with the Docker Daemon using the commands and rest APIs
  - Docker client provides a command-line interface (CLI) that allows users to run and stop application commands to a Docker daemon



# Docker

- **Daemon**

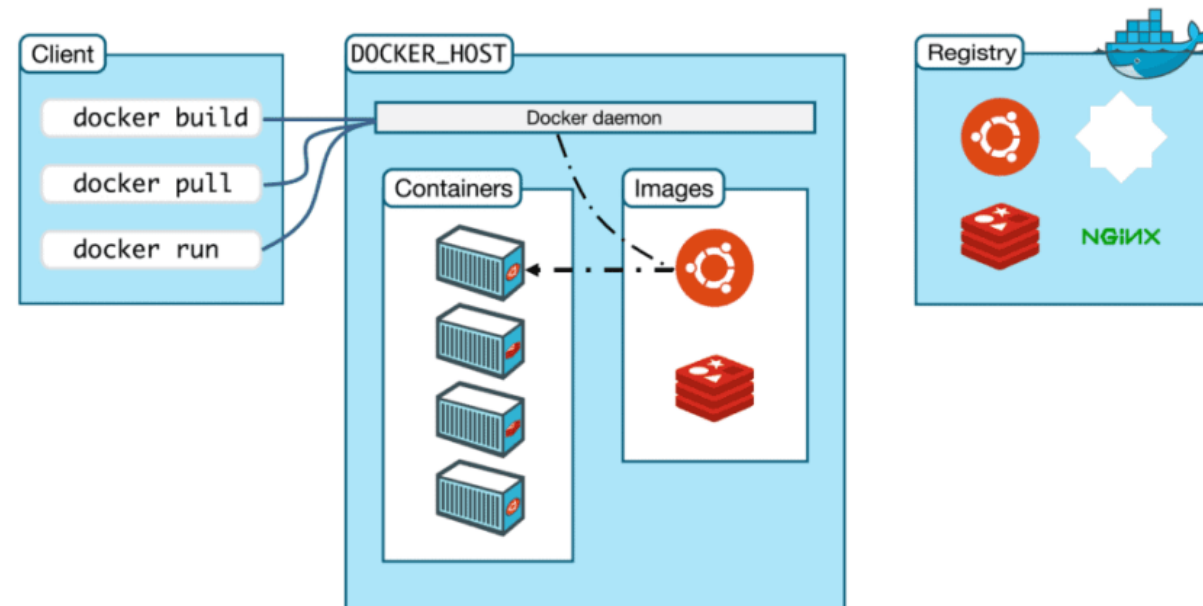
- It runs on the operating system of the host
- It handles Docker objects like images, containers, network and volumes by listening for Docker API requests
- It is used to run containers and also manage services of the Docker
- A Daemon can communicate with other Daemons to manage services
- It is responsible for handling the construction, execution and distribution of Docker containers



# Docker

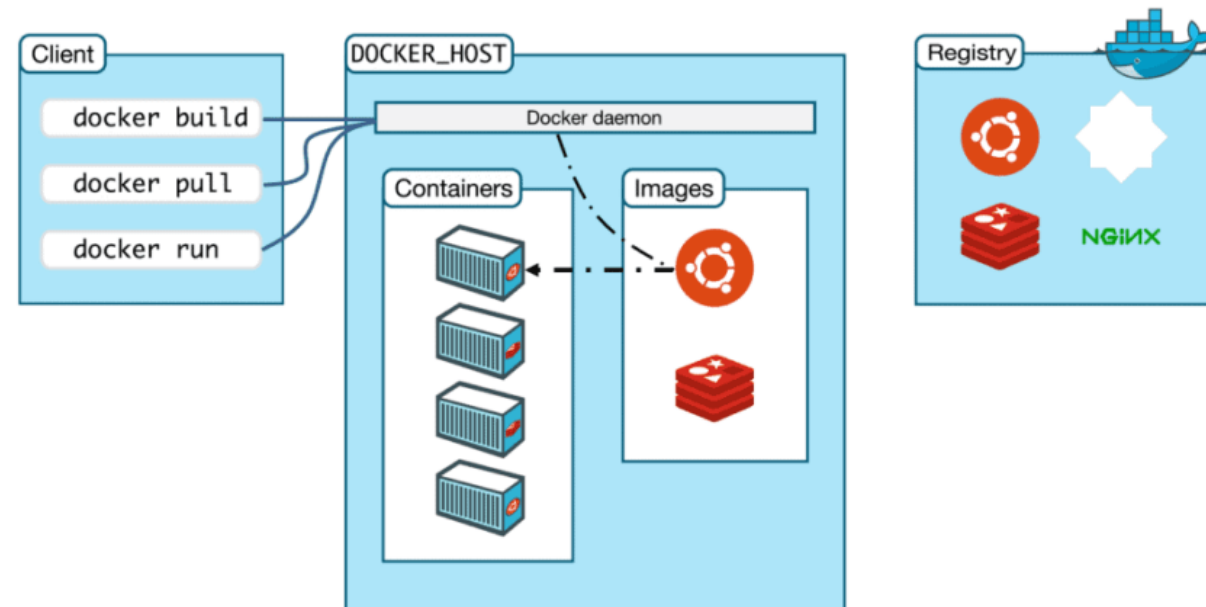
- Host

- It is very important because without it, users cannot make use of Docker
- It is an environment in which applications can be executed and run
- Commands issued by the client are sent over to the docker host which is received by the Docker daemon



# Docker

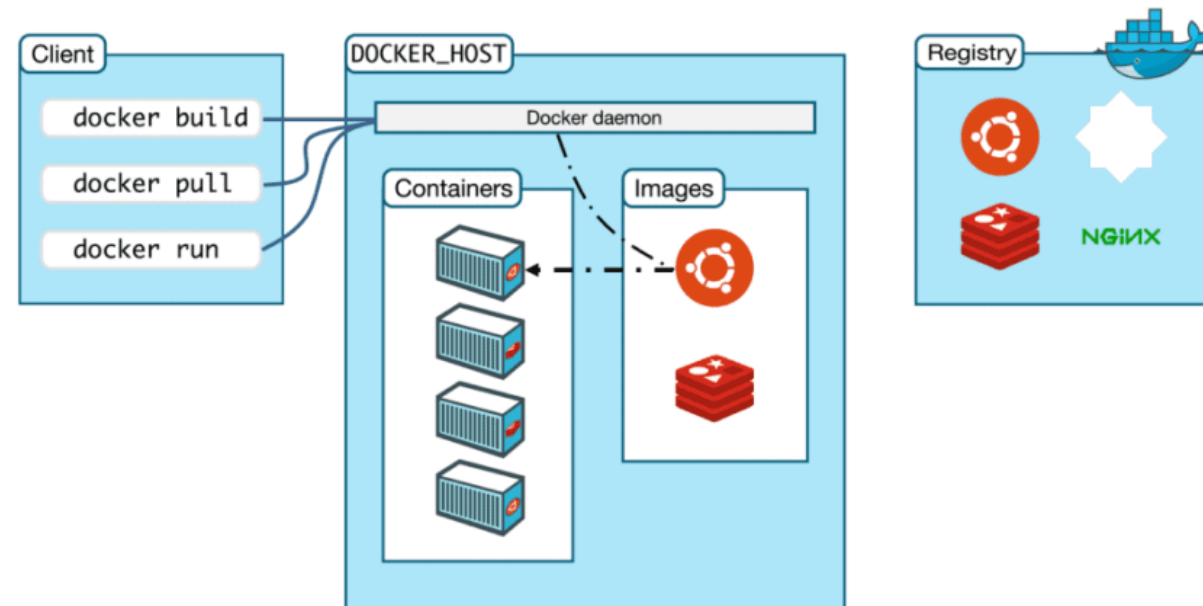
- **Registry**
  - Docker Registry also known as Docker Hub is a repository of Docker images for almost all technology stack where you can install or pull images





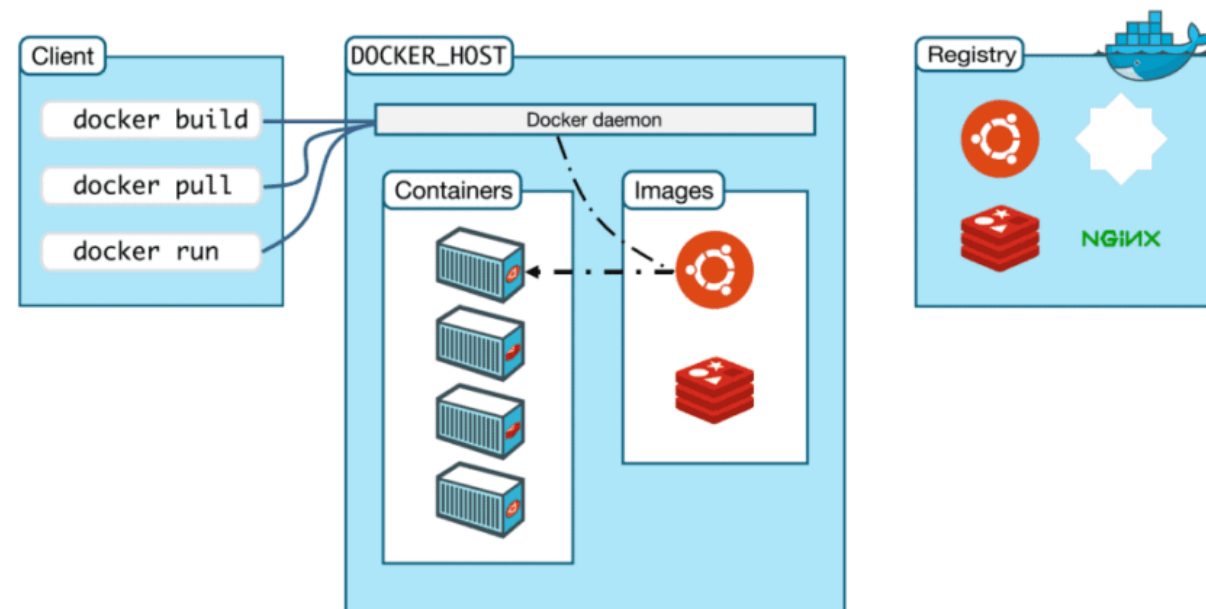
# Docker

- Docker Images
  - Docker images are read-only templates with instructions to create a docker container
  - Docker image can be pulled from a Docker hub and used as it is, or you can add additional instructions to the base image and create a new and modified docker image
  - You can create your own docker images also using a dockerfile



# Docker

- Docker **Containers**
  - Docker containers are isolated, packaged, and secured application environments that contain all the packages, libraries, and dependencies required to run an application
  - After you run a docker image, it creates a docker container
  - All the applications and their environment run inside this container
  - You can use Docker API or CLI to start, stop, delete a docker container
- For example, if you create a container associated with the Ubuntu image, you will have access to an isolated Ubuntu environment.
- You can also access the bash of this Ubuntu environment and execute commands.



# Docker

- *docker build <path to docker file>*
  - This command is used to build an image from a specified docker file
- *docker --version*
  - This command is used to get the currently installed version of docker
- *docker run -it -d <image name>*
  - This command is used to create a container from an image
- *docker ps*
  - This command is used to list the running containers
- *docker ps -a*
  - This command is used to show all the running and exited containers

# Docker

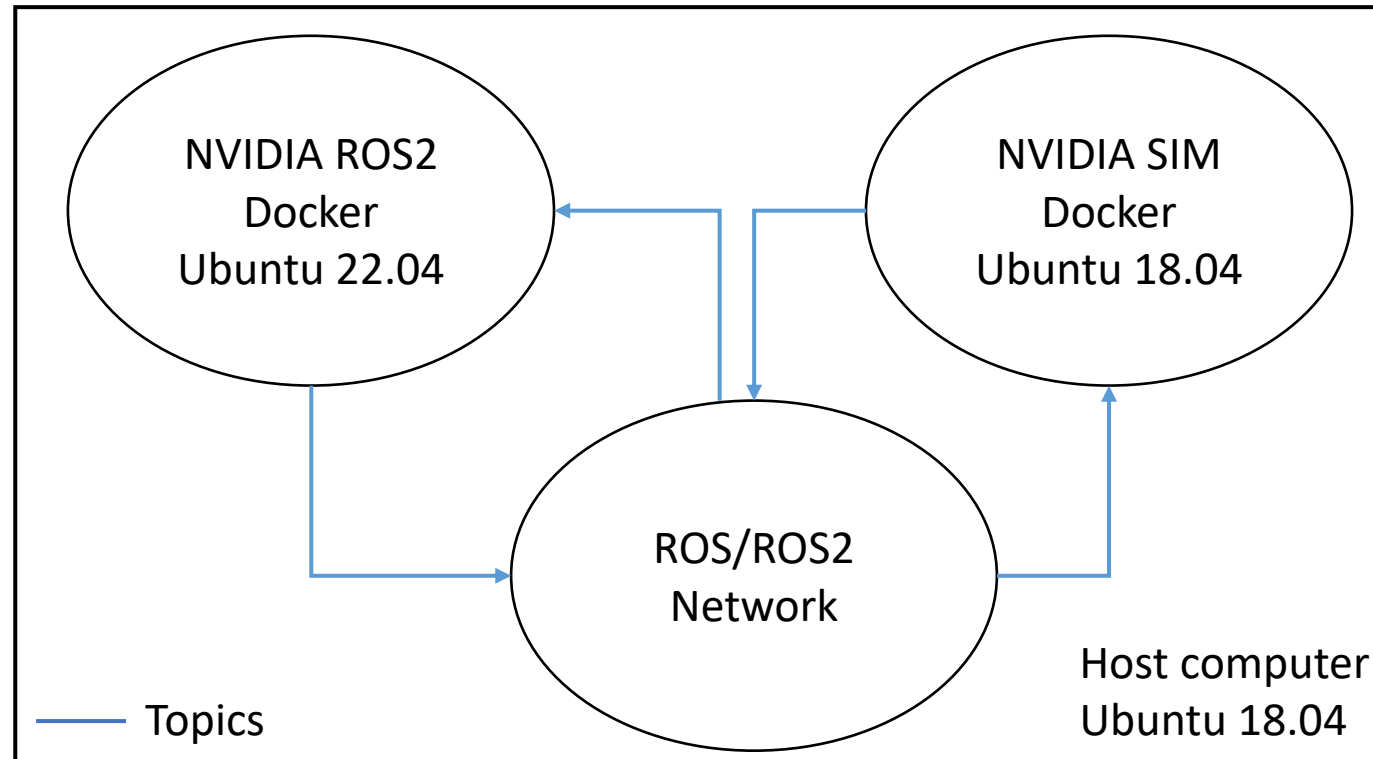
- *docker stop <container id>*
  - This command stops a running container
- *docker kill <container id>*
  - This command kills the container by stopping its execution immediately
- *docker pull*
  - This command is used to pull images from the docker repository
- *docker push <username/image name>*
  - This command is used to push an image to the docker hub repository
- *Docker rmi <image-id>*
  - This command is used to delete an image from local storage
- *docker rm <container id>*
  - This command is used to delete a stopped container

# Docker

- We will use dockers in practice in the next lessons
- The goal is to connect NVIDIA dockers with our host computer

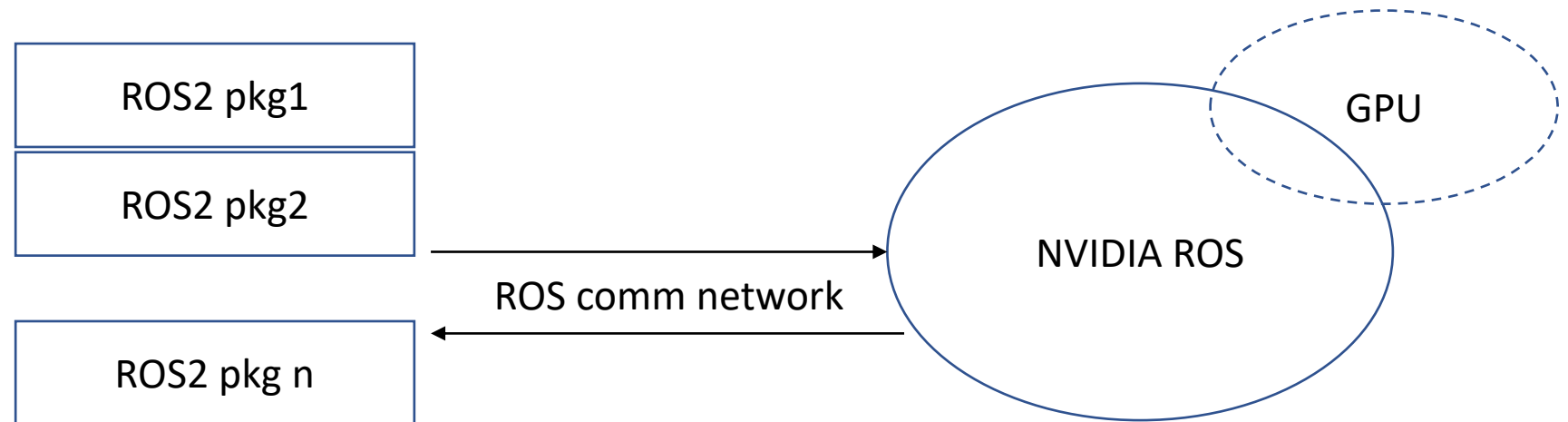
# Docker network

- Different dockers containers can share the host network
- Different dockers conotainers can communicate using the host network
- A docker image and the host can communicate using the host network



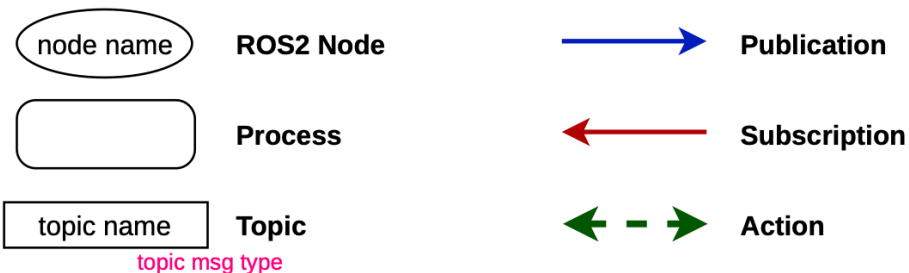
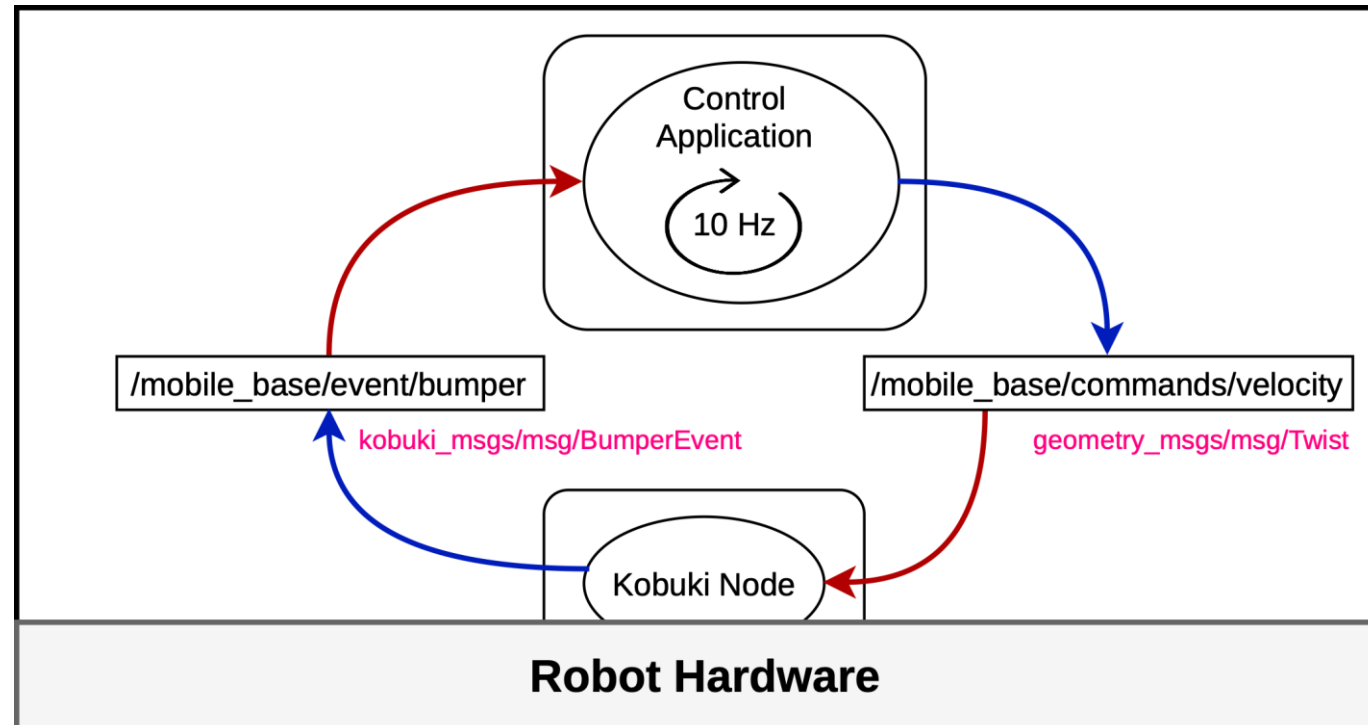
# Object detection with NN

- NVIDIA ROS uses ROS2 as main middleware
- Learning ROS2 is useful to interface our code with NVIDIA ROS applications
- NVIDIA will release new packages to support AI Robotics



# ROS2

- A robot's software looks like during its execution
- A Computation Graph contains ROS2 nodes that communicate with each other so that the robot can carry out some tasks
- The logic of the application is in the nodes, as the primary elements of execution in ROS2
- Communication mechanisms:
  - **Publication/Subscription**: Asynchronous N:M
  - **Services**: Synchronous 1:1
  - **Actions**: Asynchronous 1:1
- Execution model
  - Iterative
  - Event-Oriented





# ROS2







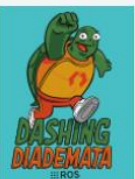
- ROS2 is the natural successor of ROS
- No QoS in communication
  - Communication delay is not predictable
  - Message priority can not be managed
- Security: the ROS master will respond to requests from any device on the network
  - ROS cannot be used in industry as is
  - You can easily control ROS based device from an external network
- The core of ROS is out-of-date, and its support will end in 2025
- After 2025 ROS project officially ends
- Are we wasting our time?



- ROS was born in 2007
  - A lot has changed in the robotics and ROS community.
- The goal of the ROS 2 project is to adapt to these changes, leveraging what is great about ROS 1 and improving what isn't.
- ROS and ROS2 shares the programming philosophy
- ROS2 is not mature enough and more difficult to learn
  - Let's start learning ROS (1)

# ROS2 Distributions

- Several distribution
  - For Ubuntu 18.04 use Dashing Diademata
  - Last Ubuntu LTS 22.04 ROS Humble
    - Ubuntu 22.04 doesn't support ROS1 anymore
    - Changes in the ROS2 core between the versions is minimal
      - Biggest changes regard the supported packages
    - NITROS (transport layer to connect ROS and NVIDIA ROS)
    - We can use any kind of distro, communicating with Humble on the docker version to communicate with NITROS

Humble Hawksbill	May 23rd, 2022		May 2027
Galactic Geochelone	May 23rd, 2021		November 2022
Foxy Fitzroy	June 5th, 2020		May 2023
Eloquent Elusor	November 22nd, 2019		November 2020
Dashing Diademata	May 31st, 2019		May 2021

# Install ROS2

- Set language settings
  - `$ locale # check for UTF-8`
  - `$ sudo apt update && sudo apt install locales`
  - `$ sudo locale-gen en_US en_US.UTF-8`
  - `$ sudo update-locale LC_ALL=en_US.UTF-8 LANG=en_US.UTF-8`
  - `$ export LANG=en_US.UTF-8`
- Setup sources
  - `$ sudo apt update && sudo apt install curl gnupg2 lsb-release`
  - `$ sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.key -o /usr/share/keyrings/ros-archive-keyring.gpg`
- Add repositories
  - `$ echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/ros-archive-keyring.gpg] http://packages.ros.org/ros2/ubuntu $(lsb_release -cs) main" | sudo tee /etc/apt/sources.list.d/ros2.list > /dev/null`
- Install packages
  - `$ sudo apt update`
  - `$ sudo apt install ros-dashing-desktop`
- Install argcomplete to autocompletion
  - `$ sudo apt install -y python3-pip`
  - `$ pip3 install -U argcomplete`

# Configure the environment

- Source the environment
  - `$ source /opt/ros/dashing/setup.bash`
- First test
  - Terminal 1: `$ ros2 run demo_nodes_cpp talker`
  - Terminal 2: `$ ros2 run demo_nodes_py listener`
- Install colcon extensions to handle ROS2 packages
  - `$ sudo apt install python3-colcon-common-extensions`

# Configure the environment

- ROS 2 relies on the notion of combining workspaces using the shell environment
  - Workspace is a ROS term for the location on your system where you're developing with ROS 2
  - The core ROS 2 workspace is called the underlay
  - Subsequent local workspaces are called overlays
  - When developing with ROS 2, you will typically have several workspaces active concurrently.
- Combining workspaces makes developing against different versions of ROS 2, or against different sets of packages, easier
- It also allows the installation of several ROS 2 distributions (or “distros”, e.g. Dashing and Eloquent) on the same computer and switching between them.
- This is accomplished by sourcing setup files every time you open a new shell, or by adding the source command to your shell startup script once
- Without sourcing the setup files, you won't be able to access ROS 2 commands, or find or use ROS 2 packages

# Configure the environment

- Setup the main workspace and the colcon installation directory
  - `$ echo "source /opt/ros/dashing/setup.bash" >> ~/.bashrc`
  - `$ echo "export _colcon_cd_root=/opt/ros/dashing/" >> ~/.bashrc`
- A workspace is a directory containing ROS 2 packages
- Before using ROS 2, it's necessary to source your ROS 2 installation workspace in the terminal you plan to work in
- This makes ROS 2's packages available for you to use in that terminal.
- You also have the option of sourcing an “overlay” – a secondary workspace where you can add new packages without interfering with the existing ROS 2 workspace that you're extending, or “underlay”
- Your underlay must contain the dependencies of all the packages in your overlay
- Packages in your overlay will override packages in the underlay
- It's also possible to have several layers of underlays and overlays, with each successive overlay using the packages of its parent underlays

# Configure the environment

- Create a new workspace directory
  - Best practice is to create a new directory for every new workspace
  - The name doesn't matter, but it is helpful to have it indicate the purpose of the workspace
  - Let's choose the directory name `ros2_ws`
    - `$ mkdir -p ~/ros2_ws/src`
    - `$ cd ~/ros2_ws/src`
      - Here you can put the source code
  - Compile the workspace
    - `$ cd ~/ros2_ws`
    - `$ colcon build`
  - Other useful arguments for `colcon build`:
    - `--packages-up-to` builds the package you want, plus all its dependencies, but not the whole workspace (saves time)
    - `--symlink-install` saves you from having to rebuild every time you tweak python scripts

# Configure the environment

- Source the overlay
  - Configure colcon to open this workspace by default
    - `$ export _colcon_cd_root=/home/jcacace/ros2_ws`
  - To add your packages (from the local workspace), just source the setup.bash file from the workspace
    - `$ colcon_cd`
    - `$ source install/setup.bash`



# ROS2 package

- A **package** can be considered a container for your ROS2 code
- If you want to be able to install your code or share it with others, then you'll need it organized in a package
- With packages, you can release your ROS 2 work and allow others to build and use it easily
- Package creation in ROS2 uses ament as its build system and colcon as its build tool
- You can create a package using either CMake or Python, which are officially supported, though other build types do exist
- ROS 2 Python and CMake packages each have their own minimum required contents:
  - C++
    - **package.xml** file containing meta information about the package
    - **CMakeLists.txt** file that describes how to build the code within the package
  - Python
    - **package.xml** file containing meta information about the package
    - **setup.py** containing instructions for how to install the package
    - **setup.cfg** is required when a package has executables, so ros2 run can find them
    - **/<package\_name>** - a directory with the same name as your package, used by ROS 2 tools to find your package, contains **\_\_init\_\_.py**

# ROS2 package

- Create a ROS2 Package
  - C++
    - `$ ros2 pkg create --build-type ament_cmake <package_name>`
  - Python
    - `$ ros2 pkg create --build-type ament_python <package_name>`
  - To create a basic sample node along with the package
    - `$ ros2 pkg create --build-type ament_cmake --node-name my_node my_package`
    - `$ ros2 pkg create --build-type ament_python --node-name my_node_py my_package_py`
  - To compile the packages
    - `$ cd ~/ros2_ws/`
    - `$ colcon build`
    - `$ source install/setup.bash`
  - Use the package
    - `$ ros2 run my_package my_node`
    - `$ ros2 run my_package_py my_node_py`

# Package elements

- Package files
  - C++
    - CMakeLists.txt
    - include
    - package.xml
    - Src
      - Check source codes
  - Python
    - my\_node\_py.py
    - package.xml
    - resource
    - setup.cfg
    - setup.py
    - test

# ROS2 nodes

- Each node in ROS should be responsible for a single, module purpose (e.g., one node for controlling wheel motors, one node for controlling a laser range-finder, etc.)
- Each node can send and receive data to other nodes via **topics**, **services**, **actions**, or **parameters**
- A full robotic system is comprised of many nodes working in concert
- In ROS2, a single executable (C++ program, Python program, etc.) can contain one or more nodes

## Example 1.9

- Writing simple Publisher/Subscriber in C++
- Create a node able to send/get data via topics
  - `$ ros2 pkg create --build-type ament_cmake cpp_pubsub`
  - Add the source code
  - Modify the dependencies' list from the package.xml file
  - Modify the CMakeLists.txt to allow the compilation of the package

## Example 2.9

- Writing simple Publisher/Subscriber in python
- Create a node able to send/get data via topics
  - `$ ros2 pkg create --build-type ament_python py_pubsub`
  - Add the source code
    - Must be put in the `py_pubsub/py_pubsub`
  - Modify the dependencies' list from the `package.xml` file
    - `<exec_depend>rcpp</exec_depend>`
    - `<exec_depend>std_msgs</exec_depend>`
  - Modify the `setup.py` file:
    - `entry_points={`
    - `'console_scripts': [`
    - `'talker = py_pubsub.publisher_node:main',`
    - `],`
    - `}`
  - Modify the `CMakeLists.txt` to allow the compilation of the package

# ROS2 Commands

- All the commands used in ROS to handle the topics work similarly for ROS2
  - `$ ros2 topic list`
  - `$ ros2 topic publisher`
  - ...
- Nodes can be inspected and handled via ROS2 node commands
  - `$ ros2 node list`
  - `$ ros2 node info <node_name>`
    - `ros2 node info` returns a list of subscribers, publishers, services, and actions (the ROS graph connections) that interact with that node

## Example 3.9

- Create a new package
  - `$ ros2 pkg create --build-type ament_cmake custom_msgs`
- Modify the CMakeLists.txt
  - `find_package(rosidl_default_generators REQUIRED)`
  - `rosidl_generate_interfaces(${PROJECT_NAME}`
    - `"srv/AddTwoInts.srv"`
  - `)`
- Modify the package.xml
  - `<export>`
    - `<build_type>ament_cmake</build_type>`
    - `<build_depend>rosidl_default_generators</build_depend>`
    - `<exec_depend>rosidl_default_runtime</exec_depend>`
  - `</export>`
  - `<member_of_group>rosidl_interface_packages</member_of_group>`



## Example 3.9

- Compile the custom services
  - `$ cd ros2_ws`
  - `$ colcon build`
  - `$ source install/setup.bash`
- Check the compiled service
  - `$ ros2 srv show custom_msgs/srv/AddTwoInts`
  - The relative header file will be named as the message file name without the Camel notation
  - Example: `AddTwoInts.srv` -> `add_two_ints.h`

## Example 4.9

- Create a new service client/server using cpp
- We can add the dependencies in the package creation
  - `$ ros2 pkg create --build-type ament_cmake cpp_srvcli --dependencies rclcpp custom_msgs`
- After written the code we need to add the dependencies
  - `add_executable(service_server src/srv_server.cpp)`
  - `ament_target_dependencies(service_server rclcpp custom_msgs)`
  - `add_executable(service_client src/srv_client.cpp)`
  - `ament_target_dependencies(service_client rclcpp custom_msgs)`
  - `install(TARGETS service_server service_client DESTINATION lib/${PROJECT_NAME})`

## Example 5.9

- Use the ROS2 parameter server
- Create a package
  - `ros2 pkg create --build-type ament_cmake cpp_parameters --dependencies rclcpp`
- Add the source code
- Run the source code
- Change the param
  - `ros2 param set /parameter_node my_parameter earth`

## Example 5.9

- Use the ROS2 parameter server
- Create a package
  - `ros2 pkg create --build-type ament_cmake cpp_parameters --dependencies rclcpp`
- Add the source code
- Run the source code
- Change the param
  - `ros2 param set /parameter_node my_parameter earth`

# ROS2 Launch file

- The launch system in ROS 2 is responsible for helping the user describe the configuration of their system and then execute it as described
  - programs to run
  - where to run them
  - what arguments to pass
- It is also responsible for monitoring the state of the processes launched, and reporting and/or reacting to changes in the state of those processes.
- Launch files written are in Python and they can start and stop different nodes as well as trigger and act on various events
- The package providing this framework is **launch\_ros**, which uses the non-ROS-specific launch framework underneath.
- One way to create launch files in ROS 2 is using a Python file
  - `$ ros2 launch` is the command responsible for starting launch files
- To use launch files, launch must be a dependency of our package
- the appropriate state.

# ROS2 Launch file

- The typical usage in ROS2 is to have launch files invoked by ROS2 tools
- We need to compile the package after edited a proper launch file (very differently from ROS)
- After running colcon build and sourcing your workspace, you should be able to launch the launch file as follows:
  - `$ ros2 launch <PACKAGE_NAME> <LAUNCH_FILE_NAME>`
- In the launch file a Node object is created
  - `Node ( package, executable, output, parameters, ... )`

## Example 6.9

- Create launch file for the Cpp parameter package
- Import the necessary python libraries
- Use the «generate\_launch\_description» function to generate the launch file structure
- Define a new node from:
  - Package: cpp\_parameters
  - Node\_executable: parameter\_node
  - Output: screen
  - Prefix: stdbuf -o L (this is used to force the flush of command line buffer)
  - Parameters: a new parameter value for: «my\_parameter»

# ROS2 Components

- In ROS1 you can write your code as a ROS node
- ROS1 nodes are compiled into executables
- In ROS 2 the recommended way of writing your code is defining shared libraries called **Components**
- Components are very similar to the ROS2 nodes written so far
- This makes it easy to add common concepts to existing code, like a life cycle
  - Load/unload component
  - Share the component
  - ...
- By making the process layout a deploy-time decision the user can choose between:
  - running multiple nodes in separate processes with the benefits of process/fault isolation as well as easier debugging of individual nodes
  - running multiple nodes in a single process with the lower overhead and optionally more efficient communication
  - ros2 launch tool can be used to automate these actions through specialized launch actions



# ROS2 Components

- Since a component is only built into a shared library it doesn't have a main function
- A component is commonly a subclass of `rclcpp::Node`
  - Since it is not in control of the thread it shouldn't perform any long running or blocking tasks in its constructor
  - It can use timers to get periodic notification
  - It can create publishers, subscribers, servers, and clients.
- An important aspect of making such a class a component is that the class registers itself using macros from the package `rclcpp_components`
- This makes the component discoverable when its library is being loaded into a running process - it acts as kind of an entry point.
- Once a component is created, it must be registered with the index to be discoverable by the tooling (the name of the component)

# ROS2 Components

- To use the components, we must rely on the composition package.
- The composition package contains a couple of different approaches on how to use components
- The three most common ones are:
  1. Start a (generic container process) and call the ROS2 service `load_node` offered by the container
    - The ROS2 service will then load the component specified by the passed package name and library name and start executing it within the running process
    - Instead of calling the ROS2 service programmatically you can also use a command line tool to invoke the ROS2 service with the passed command line arguments
  2. Create a custom executable containing multiple nodes which are known at compile time
    - This approach requires that each component has a header file (which is not strictly needed for the first case).
  3. Create a launch file and use ROS2 launch to create a container process with multiple components loaded.

# ROS2 Actions

- To use the components, we must rely on the composition package.
- The composition package contains a couple of different approaches on how to use components
- The three most common ones are:
  1. Start a (generic container process) and call the ROS2 service `load_node` offered by the container
    - The ROS2 service will then load the component specified by the passed package name and library name and start executing it within the running process
    - Instead of calling the ROS2 service programmatically you can also use a command line tool to invoke the ROS2 service with the passed command line arguments
  2. Create a custom executable containing multiple nodes which are known at compile time
    - This approach requires that each component has a header file (which is not strictly needed for the first case).
  3. Create a launch file and use ROS2 launch to create a container process with multiple components loaded.

# ROS1 <-> ROS2 bridge

- Create launch file

# NVIDIA ISAAC ROS

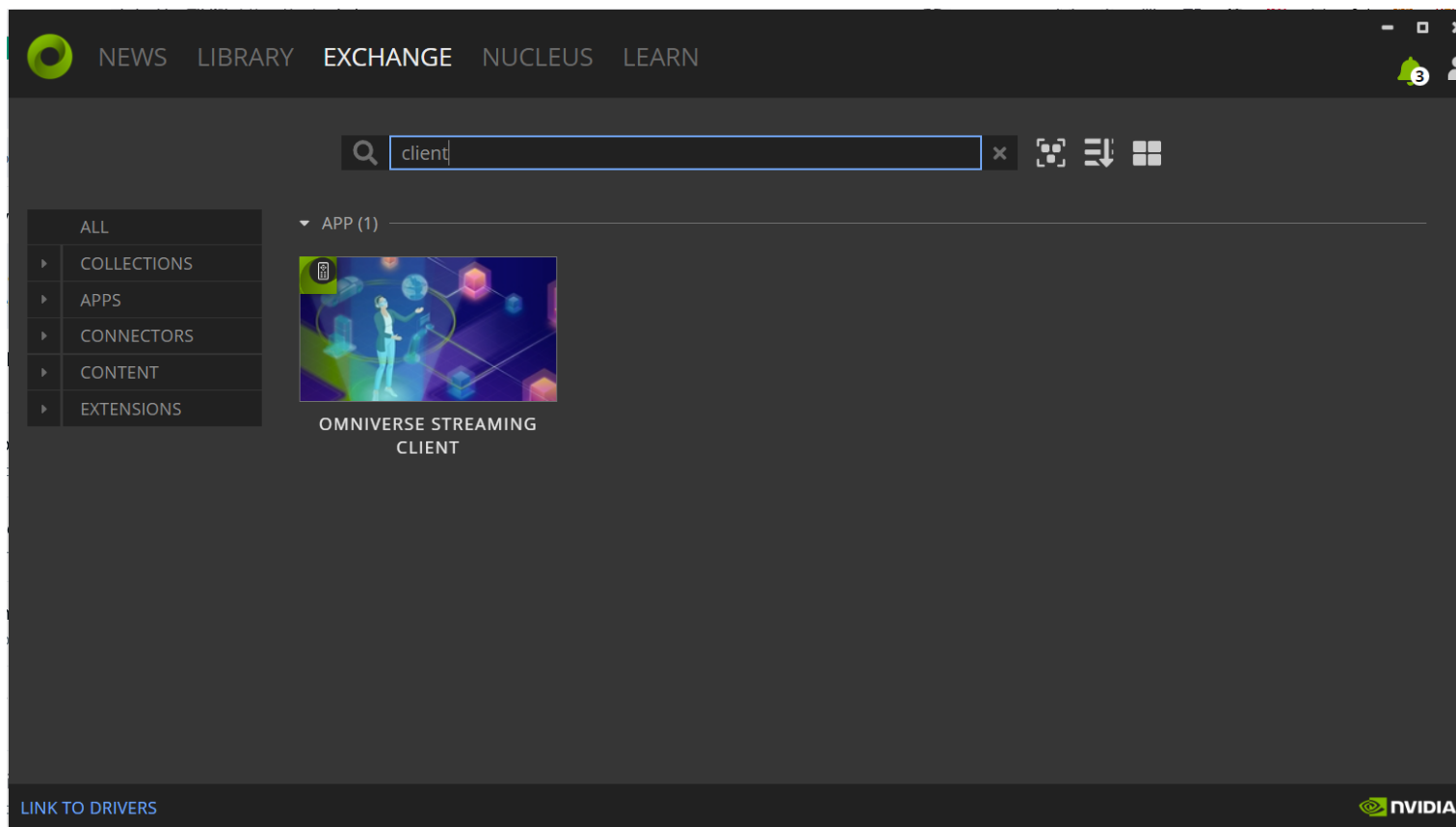
- Before starting to learn ROS, a simple demonstration
- Docker: create docker daemon.json
  - [https://github.com/NVIDIA-ISAAC-ROS/isaac\\_ros\\_common/blob/main/docs/dev-env-setup.md](https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_common/blob/main/docs/dev-env-setup.md)
  - `sudo usermod -aG docker $USER && newgrp docker`
  - `docker exec -it CONTAINER_id bash`
- Modify the `setup_model.sh`


- `$ sudo apt-get update`
- `sudo apt-get install nvidia-driver-515`
- `$ sudo apt-get install nvidia-headless-515`
- `$ sudo apt install nvidia-utils-515`
- `$ nvidia-smi`
- `$ sudo apt install docker.io`
- `$ sudo docker pull public.ecr.aws/l1t2k0k4/isaac-sim:2022.1.1`
- <https://www.nvidia.com/en-us/omniverse/download/>
  - Fill the form
  - Select the platform
  - <https://drive.google.com/file/d/1dAqigr2kl89fF2dN9JYvhQqC-eidIWOf/view?usp=sharing>



- `$ sudo apt-get update`
- `sudo apt-get install nvidia-driver-515`
- `$ sudo apt-get install nvidia-headless-515`
- `$ sudo apt install nvidia-utils-515`
- `$ nvidia-smi`
- `$ sudo apt install docker.io`
- `$ sudo docker pull public.ecr.aws/l1t2k0k4/isaac-sim:2022.1.1`
- `sudo docker run --name isaac-sim --entrypoint bash -it --gpus all -e "ACCEPT_EULA=Y" --network=host -v ~/docker/isaac-sim/cache/ov:/root/.cache/ov:rw -v ~/docker/isaac-sim/cache/pip:/root/.cache/pip:rw -v ~/docker/isaac-sim/cache/gldownload:/root/.cache/nvidia/GLDownload:rw -v ~/docker/isaac-sim/cache/computecache:/root/.nv/ComputeCache:rw -v ~/docker/isaac-sim/logs:/root/.nvidia-omniverse/logs:rw -v ~/docker/isaac-sim/config:/root/.nvidia-omniverse/config:rw -v ~/docker/isaac-sim/data:/root/.local/share/ov/data:rw -v ~/docker/isaac-sim/documents:/root/Documents:rw public.ecr.aws/l1t2k0k4/isaac-sim:2022.1.1`
- `$ git clone https://github.com/robotic-software/isaac-sim`
- <https://www.nvidia.com/en-us/omniverse/download/>
  - Fill the form
  - Select the platform
  - <https://drive.google.com/file/d/1dAqigr2kl89fF2dN9JYvhQqC-eidIWOf/view?usp=sharing>





- List docker
- `$ sudo docker ps -a`
- Get CONTAINER ID
- Start the contained
- `sudo docker start CONTAINER ID`
- Attach the docker
- `sudo docker attach 33c661ed46b2`
- `./runheadless.native.sh`
  - Wait for Isaac Sim Headless Native App is loaded.






[NEWS](#)
[LIBRARY](#)
[EXCHANGE](#)
[NUCLEUS](#)
[LEARN](#)

ALL


▶ COLLECTIONS

▶ APPS

▶ CONNECTORS

▶ CONTENT

▶ EXTENSIONS




OMNIVERSE | STREAMING CLIENT

103.1.1

INSTALL

×




### About Omniverse Streaming Client

Omniverse Streaming Client is a lightweight application that gives users the ability to stream Omniverse Apps to their favorite thin client device. Omniverse Streaming Client builds the

Publisher

NVIDIA

[LINK TO DRIVERS](#)


- INSTALL ROS2 (eloquent)

# Tutorial ROS2

- INSTALL ROS2 (eloquent)