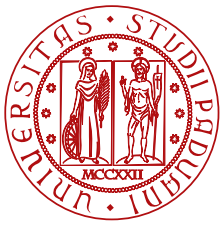


UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

Development of an Underwater Stereo Vision System: Camera Comparison, Hardware Integration, and Robot Control with ROS2

Relatore
Prof. Varagnolo Damiano

Laureando
Toniolo Jacopo

ANNO ACCADEMICO 2024-2025

Data di laurea 23/07/2025

Summary

The aim of this document is to provide a comprehensive overview of the development process of an Underwater Stereo Vision System.

The proposed system is designed to be as versatile as possible, making it suitable for a wide range of applications, including:

- 3D reconstruction of underwater and surface objects
- Identification and classification of marine species
- Seabed reconstruction

This report specifically addresses the following aspects:

- **HARDWARE**: enclosure design, powering methods, electronic boards, camera modules
- **SOFTWARE**: system architecture based on ROS 2 Jazzy
- **COLLABORATIVE DEVELOPMENT**: use of tools such as GitHub, Docker, and GitHub Actions to enable distributed development across team members
- **CONSIDERATIONS ABOUT THE TEST DESIGN**: During the prototype testing session, we identified the strengths and weaknesses of our system.

The focus of my work was primarily on the initial hardware design of the system and the implementation of the ROS 2 software stack.

While the system was developed with underwater applications in mind, the ROS-based architecture is highly modular and reusable. It can also be employed in other robotic projects, such as the autodocking system developed by another research group.

Contents

List of Figures

Chapter 1

Hardware analyses

This chapter focuses on the hardware components used in the system and their respective roles. The initial plan was to use only one board, such as the Nvidia Jetson Nano, but this choice was later reconsidered for the reasons explained in the following chapters.

1.1 Boards

To ensure modularity, energy efficiency, and scalability, the system architecture is based on two separate boards:

- The **main board**, responsible for general operations, sensor interfacing, and overall system coordination.
- The **computing board**, activated only when computationally intensive tasks are required, in order to minimize average power consumption.

This dual-board architecture allows for the separation of control and processing tasks, optimizing resource allocation and power usage. It also improves maintainability and enables independent upgrades of either subsystem.

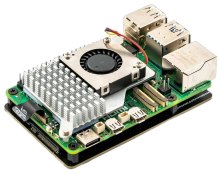
The physical and logical organization of the boards is discussed in the following subsections.

1.1.1 Main System Board

The main system board must meet the following key requirements:

- **Cost-effective:** It should be affordable and widely available on the market.
- **General-purpose:** Capable of performing a broad range of non-specialized tasks reliably.

- **Easily replaceable:** It should avoid unique or proprietary features to ensure modularity and reduce system coupling.



Since the entire system is built on the ROS 2 framework, compatibility with Ubuntu is essential.
A suitable and reliable choice that meets all the criteria is the **Raspberry Pi 5**.

1.1.2 Computing Board

In addition to the main board, the system must perform computationally intensive tasks, particularly related to computer vision, for which the Raspberry Pi alone is insufficient. Two alternative approaches were considered:

1. Integrating a Coral USB Accelerator with the Raspberry Pi.
2. Replacing the Raspberry Pi with an NVIDIA Jetson Nano.

1.1.2.1 Raspberry Pi 4 + Coral Edge TPU

The Coral USB Accelerator includes an Edge TPU, a specialized ASIC (Application-Specific Integrated Circuit) developed by Google. It is designed specifically for fast and efficient execution of machine learning inference tasks, primarily tensor operations.

Table 1.1: Comparison: Raspberry Pi 4 + Coral Edge TPU

Pros	**Cons**
Lower power consumption	Performance benefits limited to ML-based algorithms only
Lower overall cost	Pre/post-processing is still handled by the Raspberry Pi, which may be slow
Efficient execution of TensorFlow Lite models	Incompatible with PyTorch/OpenCV without conversion

1.1.2.2 NVIDIA Jetson Nano

The Jetson Nano is a powerful AI development board from NVIDIA, designed for real-time processing of data streams and parallel execution of neural networks. It supports multiple high-resolution inputs and is suitable for both ML and traditional computer vision tasks.

Table 1.2: Comparison: NVIDIA Jetson Nano

Pros	**Cons**
High computational performance (GPU)	Higher power consumption
Native support for a wide range of libraries (TensorFlow, PyTorch, OpenCV)	Higher cost
Supports both ML and non-ML algorithms	
Large community and strong documentation	

1.1.2.3 Final Choice

The approximate costs of the two configurations are:

- **Raspberry Pi 4 + Coral USB Accelerator:** €120
- **Jetson Nano:** €200

The Coral-based setup is preferred when:

- Budget is highly constrained.
- Only ML-based processing is required.
- TensorFlow Lite is the primary development framework.
- Low power consumption is critical.

However, the Jetson Nano provides a more robust and flexible solution due to:

- Superior processing power for both ML and traditional CV tasks.

- Compatibility with a wide range of frameworks and tools.
- Ability to handle all stages of the inference pipeline (pre-, inference, and post-processing).
- Greater support from the developer community.

For these reasons, we selected the NVIDIA Jetson Nano as the computing board for our system.

This decision is supported by multiple comparisons and recommendations found in online communities and forums [1][2].

1.2 Cameras

The initial constraints for the camera selection were:

- Compatibility with Jetson Nano
- Performance in low-light environments

Through the NVIDIA Marketplace, we consulted a table listing cameras supported by Jetson Camera Partners on the Jetson platform. The marketplace allows filtering based on specific camera requirements, such as:

- Interface type (selected: MIPI)
- Supported boards (selected: Nano — note that Jetson Nano $\neq$ Jetson Orin Nano)

Using these filters, we identified 199 compatible cameras. Among them, 73 were available for purchase, and 15 were suitable for low-light environments. All details are provided in this JSON file, with additional information available here.

To select the most suitable camera, we evaluated several parameters:

- **Shutter type**: a global shutter is preferred, as rolling shutters (more common) introduce distortions (rolling shutter effect).
- **Frame rate**: high frame rates are not necessary for our application and may burden the processing capabilities of the board.
- **Responsivity**: indicates sensitivity to light (low-light cameras typically have an average responsivity $> 1\text{V/lux}\cdot\text{s}$).
- **Field of View (FOV)**, specifically the horizontal FOV (HFOV)

- **Sensor resolution**
- **Pixel size**
- **Hypothetical baseline**, which can be adjusted to meet system requirements

To compare the cameras, we developed a Python script that takes as input:

- **HFOV** [degrees]
- **Sensor resolution** (W×H) [pixels]
- **Pixel size** (W×H) [µm]
- **Disparity step** [pixels]
- **Baseline** [mm]

The script generates a ”depth vs. depth error” graph based on the depth resolution formula presented in [3]:

$$\Delta z = \frac{z^2 \Delta d}{bf + z\Delta d} \tag{1.1}$$

where:

z is the target depth

Δd is the disparity step (step between two successive levels of disparity)

b is the baseline

f is the focal length in pixels

In most cases, the focal length is not explicitly provided, and must be estimated using the following formula [4]:

$$f_{[\text{mm}]} = \frac{\sqrt{(s_W \cdot p_W)^2 + (s_H \cdot p_H)^2}}{2 \cdot \tan\left(\frac{HFOV}{2}\right)} \tag{1.2}$$

Where:

- s_W, s_H: sensor width and height [pixels]
- p_W, p_H: pixel width and height [mm]
- $HFOV$: horizontal field of view [degrees]

We verified this formula using camera datasheets where pixel size, sensor resolution, HFOV, and focal length were reported. The results were consistent. We then computed the focal length in pixels, which is required by the depth accuracy formula:

$$f_{[\text{px}]} = \frac{\sqrt{s_W^2 + s_H^2}}{2 \cdot \tan\left(\frac{HFOV}{2}\right)} \tag{1.3}$$

Before proceeding with the full comparison, we validated our methodology by applying the script to commercial stereo cameras, such as:

- ZED 2i
- Gemini 335/336

We compared our estimated depth errors with those reported in the datasheets. For example, the ZED 2i datasheet reports:

- $<$ 0.8% @ 2m
- $<$ 4.0% @ 12m

Our calculated values were:

- $<$ 0.78% @ 2m
- $<$ 4.5% @ 12m

Another important factor is the minimum detectable distance. Just like humans cannot see their own nose, stereo cameras cannot detect objects that are too close. The closest point detectable corresponds to the intersection of the two FOVs. This minimum distance can be estimated using the baseline b and the $HFOV$ via basic trigonometry (Figure 1.1):

$$d = \frac{b}{2} \cdot \tan\left(\frac{\pi - HFOV}{2}\right) \tag{1.4}$$

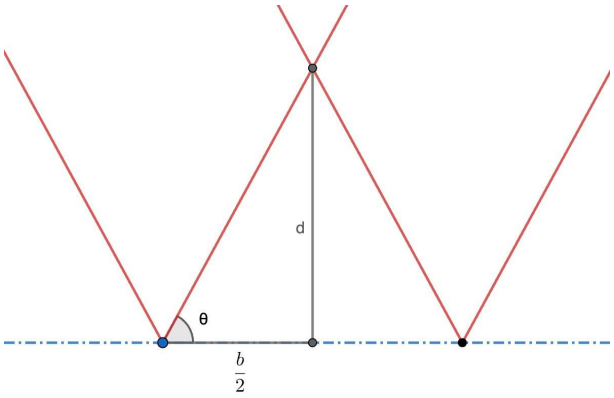


Figure 1.1: Trigonometric interpretation of the minimum detectable depth

The value d represents the closest point visible from both cameras. However, this is likely too close for practical use, as only a very limited area would be observed by both sensors (Figure 1.2).

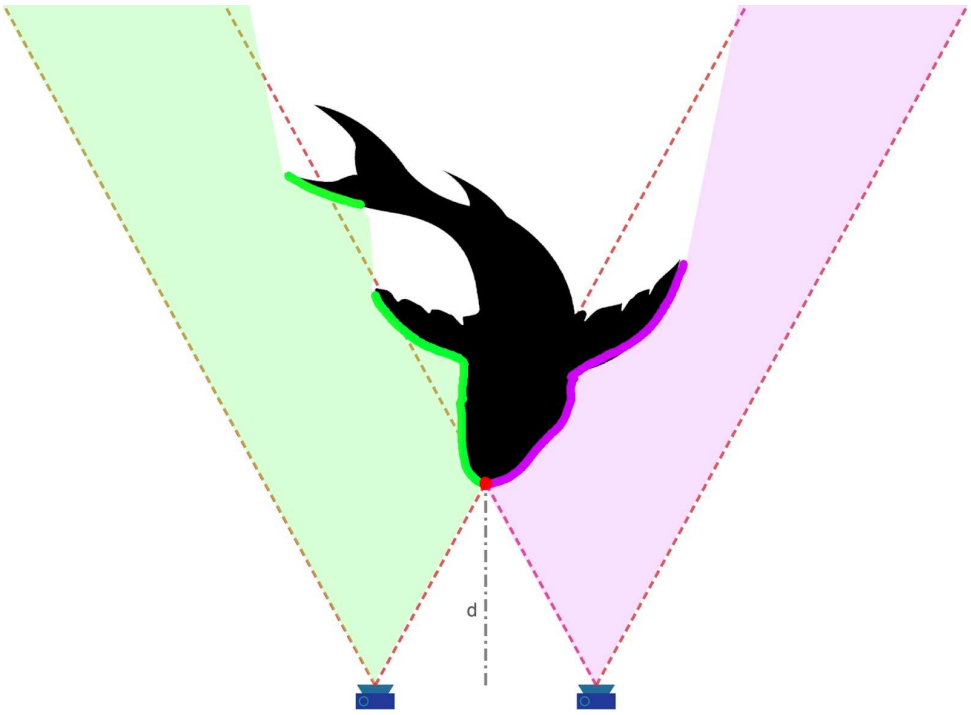
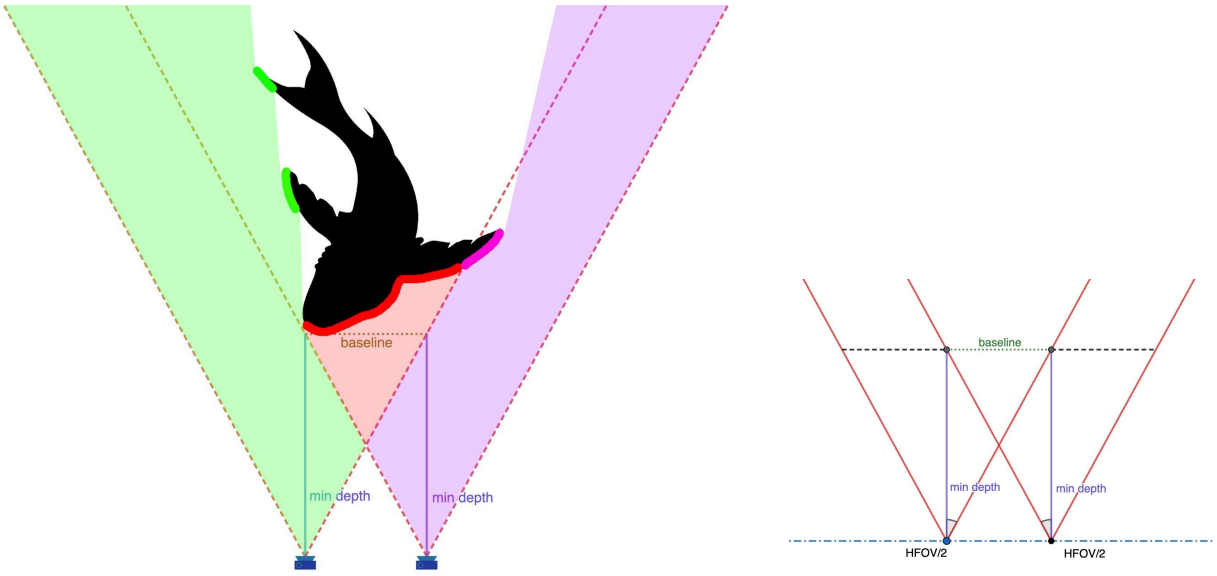


Figure 1.2: Minimum depth based on FOV intersection percentage

Defining the minimum depth as the point at which the two FOVs intersect by at least 50% improves the stereo overlap and the amount of shared visual information (Figure 1.3).



(a) Image A (b) Image B

Figure 1.3: Comparison between two minimum depth strategies

To compute this distance with 50% FOV overlap, the following formula can be used [5]:

$$min\_depth = \frac{baseline}{\tan\left(\frac{HFOV}{2}\right)} \tag{1.5}$$

Optimal camera

Following this study, we identified the e-CAM30_CUNANO as a potential candidate:

- Resolution: 2304 × 1536
- Pixel size: 2.2 µm × 2.2 µm
- HFOV: 96°
- Shutter: rolling
- Price: $79
- Responsivity: 2 V/(lux·s) (average among low-light cameras)
- Frame rate: 38–60 fps

Figure 1.4: e-CAM30_CUNANO

Table 1.3:
BASELINE = 60 mm
MIN DEPTH = 45 mm

Depth [mm]	Depth Error [%]
500	0.79
1000	1.58
1500	2.35
2000	3.11

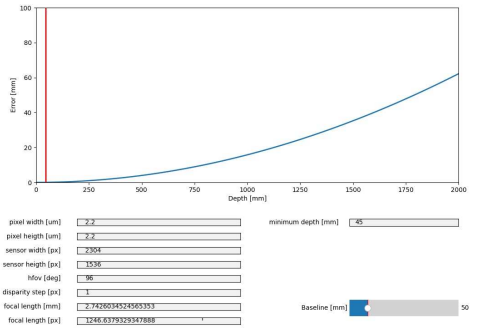


Table 1.4:
BASELINE = 100 mm
MIN DEPTH = 90 mm

Depth [mm]	Depth Error [%]
500	0.40
1000	0.80
1500	1.19
2000	1.58

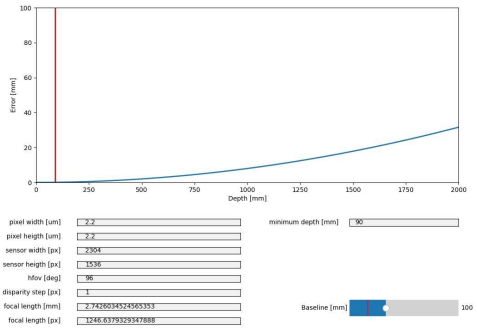


Table 1.5:
BASELINE = 150 mm
MIN DEPTH = 135 mm

Depth [mm]	Depth Error [%]
500	0.27
1000	0.53
1500	0.79
2000	1.06

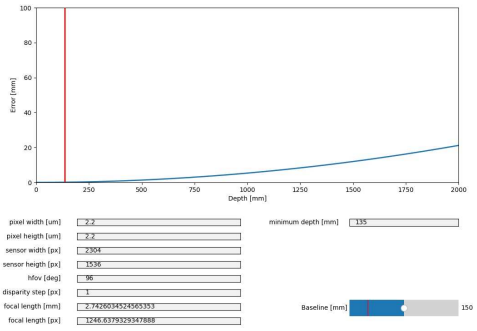
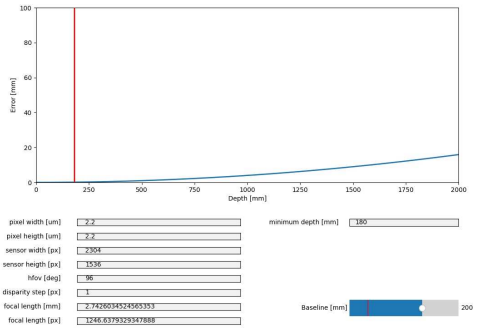


Table 1.6:
BASELINE = 200 mm
MIN DEPTH = 180 mm

Depth [mm]	Depth Error [%]
500	0.20
1000	0.40
1500	0.60
2000	0.80



The camera performs well at short distances, and the error remains relatively low even at longer ranges. The only potential issue is the rolling shutter, which may introduce image distortion. However, considering the BlueROV's maximum speed of 1.5 m/s, such distortions are expected to be minor and can likely be mitigated through software.

The *e-CAM30_CUNANO* would be an optimal choice due to its performance, but it is difficult to source and relatively expensive. For the first iteration of our system, we needed a more accessible and cost-effective solution.

For this reason, we decided to use the **RPi Camera Module 3** during the testing phase. Although this camera is not natively compatible with the Jetson Nano, appropriate drivers can be installed. It is also a general-purpose camera, making it suitable for a variety of early development tasks.

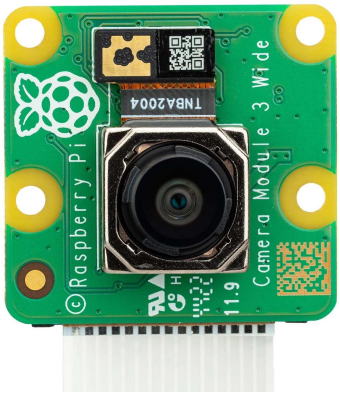


Figure 1.5: RPi Camera V3

1.3 Power

Two different approaches can be used to power the system:

- **Battery powered** — where the key factor is the battery capacity;
- **Power over Ethernet (PoE)** — more suitable for prolonged operation.

The power consumption of each component must be taken into account:

Raspberry Pi 5: up to 9.8 W [6]

Jetson Nano: up to 5 W or 10 W depending on the operating mode [5], [7]

e-CAM30_CUNANO cameras: up to 1.16 W each

RPi Camera Module 3 (used for testing): draws up to 250 mA [8], corresponding to approximately 1.25 W each

In the worst-case scenario, the upper bound for total power consumption is approximately **12.5 W**, assuming a Jetson Nano and two Raspberry Pi Camera Module 3s.

1.3.1 Battery Powered

The first and most straightforward approach to powering the system involves using a battery pack. This can either be a commercial power bank or a custom-built battery pack.
Regardless of the battery type, the main requirement is that it must provide at least 5 V at 3 A, which corresponds to a total power output of 15 W.

Assuming that the single board and two cameras together draw a total of 15 W a 10,000 mAh power bank or battery pack should be able to power the system for approximately 3 hours and 20 minutes.

A battery pack capable of delivering 5 V at 10 A can be assembled using twelve 18650 cells (3.2 V, 1.6 Ah each) arranged in a 2S6P configuration, yielding a total energy capacity of 61.44 Wh. With a step-down voltage regulator operating at approximately 90% efficiency, the usable output would be around 55.3 Wh, equivalent to roughly 5 V at 11 Ah.

It is important to consider the discharge rate as well. For example, a typical power bank can only supply up to 3 A—equivalent to 15 W at 5 V—meaning that one power bank would be required per board.
A custom battery pack, on the other hand, can deliver a higher current and thus power multiple boards simultaneously.

1.3.2 Power over Ethernet (PoE)

Another viable approach consists in using Power over Ethernet (PoE) (Schema on figure 1.6). Neither the Jetson Nano nor the Raspberry Pi 5 supports PoE natively. To enable PoE functionality, a PoE splitter is required for the Jetson, while the Raspberry Pi 5 needs a dedicated PoE HAT.

There are two primary types of PoE standards:

- **PoE** (IEEE 802.3af): delivers up to 15.4 W
- **PoE+** (IEEE 802.3at): delivers up to 30 W

These values do not account for power losses due to cable resistance.

Given that the estimated maximum total consumption is around 12.5 W, only PoE+ (IEEE 802.3at) will be considered in order to provide sufficient power to the system.

It is worth estimating the maximum allowable cable resistance when using PoE+. The following terminology will be used:

PSE – Power Sourcing Equipment (e.g. the PoE injector)

PD – Powered Device (e.g. the PoE splitter)

Using a PoE+ setup, we can power only one board at a time. Regardless of which board is selected, the Ethernet cable used will be the tether cable provided by BlueRobotics [9].

The PoE injector automatically negotiates the power requirement with the PD. The IEEE 802.3at standard defines four supported power classes. A class 4 PD can request up to 25.5 W of power Due to inherent cable losses, the PSE must provide up to 30 W to satisfy the PD's demand. In the worst case, the maximum power the cable can dissipate corresponds to 30 W – 25.5 W = 4.5 W.

According to IEEE 802.3at, the output voltage range of the PSE is 50–57 V. In the worst-case scenario—where the PSE supplies only 50 V and the PD demands the full 25.5 W (hence the PSE must provide 30W) — the current flowing through the cable would be:

$$I = \frac{30\,\mathrm{W}}{50\,\mathrm{V}} = 0.6\,\mathrm{A}$$

The maximum allowable resistance of the cable can then be computed as:

$$R = \frac{P}{I^2} = \frac{4.5\,\mathrm{W}}{(0.6\,\mathrm{A})^2} = 12.5\,\Omega$$

If the PSE outputs 56 V instead, the current drops to:

$$I = \frac{30\,\mathrm{W}}{56\,\mathrm{V}} \approx 0.536\,\mathrm{A}$$

yielding a maximum cable resistance of:

$$R = \frac{4.5\,\mathrm{W}}{(0.536\,\mathrm{A})^2} \approx 15.6\,\Omega$$

Note that these are worst-case values. The BlueRobotics tether cable has a resistance of 0.127 Ω/m [9]. For a 50 m long cable, the total resistance would be:

$$R_{\text{total}} = 50\,\mathrm{m} \times 0.127\,\Omega/\mathrm{m} = 6.35\,\Omega$$

which is well below the critical threshold.

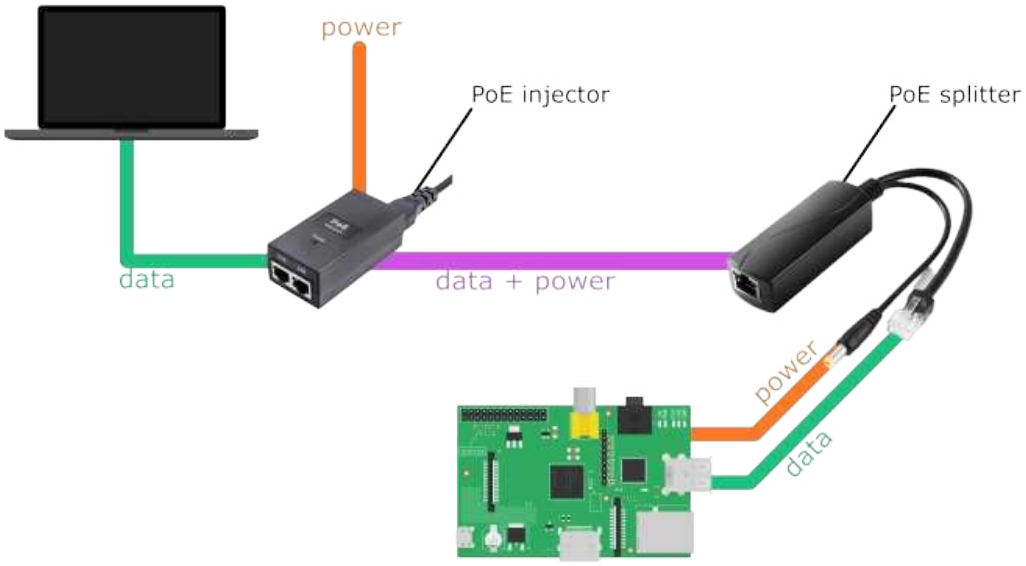


Figure 1.6: PoE+ schema

1.4 Structure

Considering that the system must operate on the seafloor at a depth of approximately 20 m, the most suitable solution for the external structure is to use the enclosures developed by BlueRobotics.

These enclosures consist of a transparent cast-acrylic tube with specific end caps that ensure watertightness. The transparency of the tube is advantageous, as it allows the cameras to be oriented directly toward the seafloor. The Jetson Nano (100×80 mm) is larger than the Raspberry Pi 5 (85×56 mm); therefore, the former will be used as the reference for this structural analysis.

BlueRobotics provides two types of enclosures:

Locking series – newer, better water resistance

Non-locking series – older, lower depth rating

Locking Series

The largest transparent enclosure in the locking series has the following specifications:

Inner Diameter (ID): 101.6 ± 2.1 mm

Outer Diameter (OD): 114.3 ± 0.8 mm

Maximum Length: 400 mm

Depth Rating: 60 m

This tube has an ID that is only slightly larger than the Jetson Nano.

Option 1 – Short side perpendicular to the tube axis: Inserting the board with its short side perpendicular to the tube axis does not leave sufficient space for the cables (Figure 1.7).

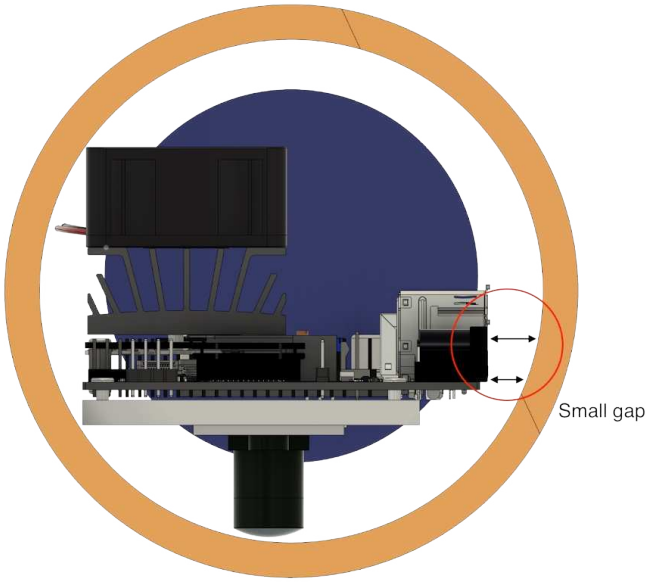


Figure 1.7: Jetson with short side perpendicular

Option 2 – Short side along the tube axis: Alternatively, inserting the board lengthwise increases the risk of it becoming stuck (Figure 1.8).

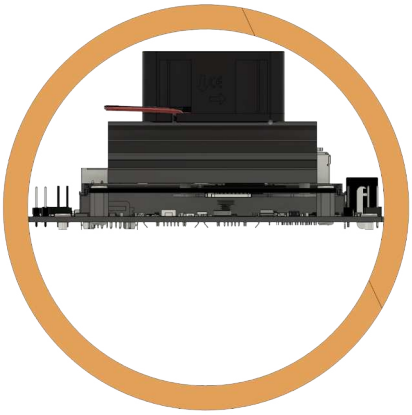


Figure 1.8: Jetson with short side parallel

Non-Locking Series

The non-locking series offers a wider range of transparent tube diameters.
Choosing a non-locking enclosure would solve the space constraints; however, it would also result in a lower depth rating and increased complexity in opening and closing the enclosure.
On the other hand, BlueRobotics has made the non-locking series completely open-source. They have released the 3D models to the public, allowing us to create custom enclosures based on their design. This grants us greater flexibility in terms of tube dimensions (e.g., length).

For these reasons, a custom non-locking series enclosure was selected for the final setup (Figure 1.11)

Final System Design

The final system will require four main components:

- Jetson Nano
- Raspberry Pi 5
- Two MIPI cameras
- Battery

This setup will enable communication with the system while it remains above water.
To achieve underwater communication, an acoustic modem will be necessary. In this case, the enclosure must also house an acoustic transceiver and an Ethernet switch, as both the Jetson Nano and the Raspberry Pi 5 have only one Ethernet port.
If there is not enough room for all the components, the enclosure can be extended by using a longer tube, while keeping the same end caps and flanges.

Test Design

Given the cost of the final system components, every detail must be carefully validated.
One of our goals was to conduct preliminary tests prior to finalizing the design. For this reason, a 3D-printable test box was developed (Figure 1.9 and 1.10).
The box is made of PLA and subsequently coated with epoxy resin. The lid is secured with 14 bolts, and a silicone gasket ensures water-tightness by sealing the gaps.

This setup is not intended for deep-water deployment. Its main purpose is to allow for rapid system testing and data collection.
Each box can accommodate only one board and a battery pack. However, thanks to the modular design, one box can be stacked on top of another, effectively doubling the available space.

Figure 1.9: Assembled test box.

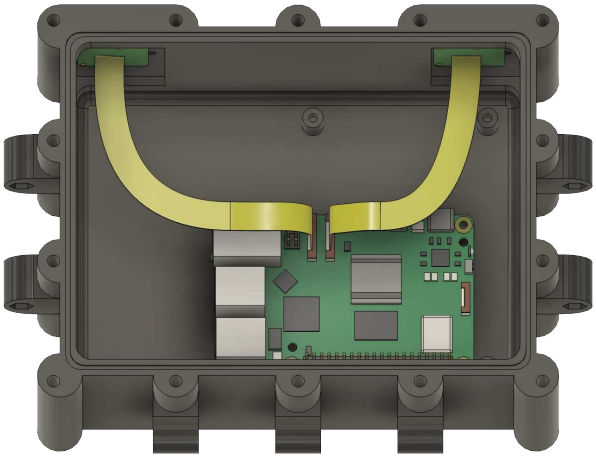


Figure 1.10: Top view of the test box.

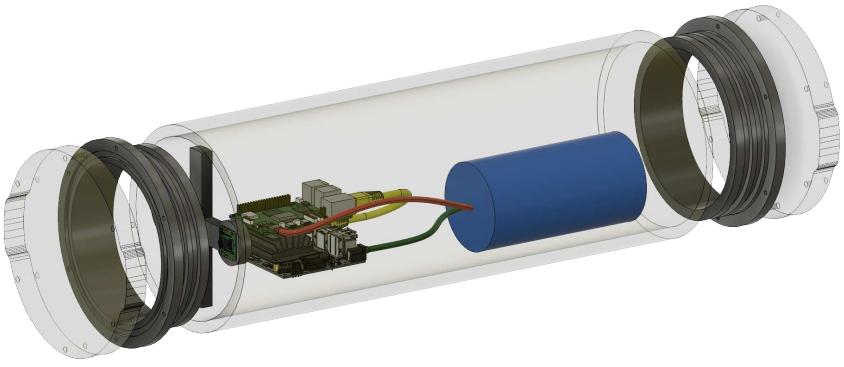


Figure 1.11: Exploded (disassembled) view of the enclosure.

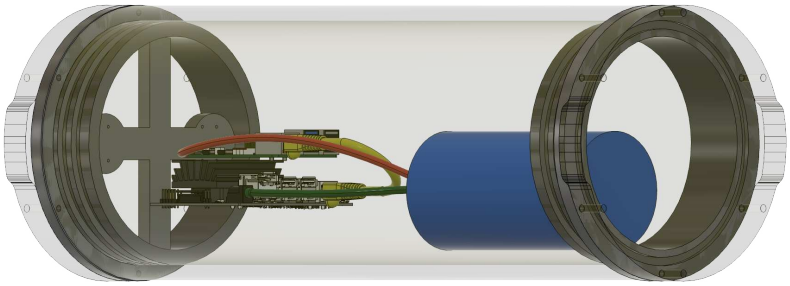


Figure 1.12: Side view of the enclosure

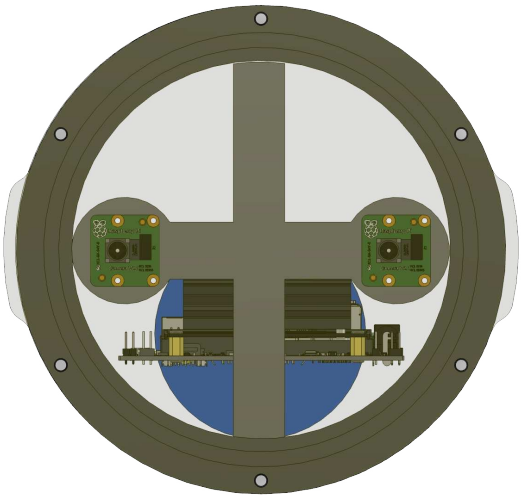


Figure 1.13: Front view of the enclosure

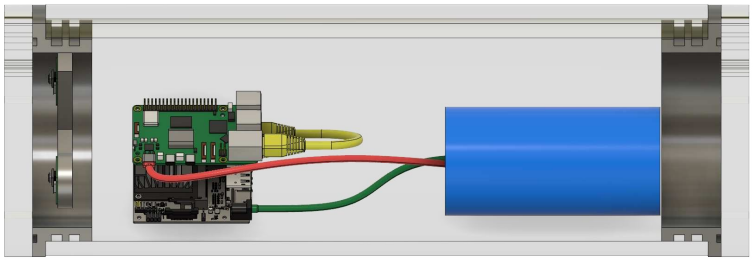


Figure 1.14: Larger internal section of the enclosure

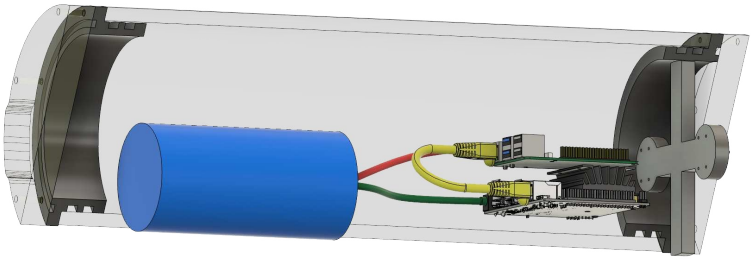


Figure 1.15: Smaller internal section of the enclosure

Chapter 2

Software System

ROS 2 (Robot Operating System 2) is an open-source framework for building robot software. It supports the development, deployment, and operation of robotic systems across a wide range of platforms and applications. It provides a publish/subscribe communication model, services, actions, and parameter servers for modular and reusable code. Under the hood, all communication relies on the *Data Distribution Service (DDS)*, which is independent of ROS 2 and enables real-time data exchange.

2.1 Component Overview

Our system relies primarily on publishers and subscribers. For critical nodes that require high fault tolerance, we use managed lifecycle nodes. We also selected a specific DDS implementation to meet our real-time and reliability requirements.

2.1.1 Nodes

A *node* in ROS 2 is a single, modular process responsible for a specific function—such as controlling wheel motors or publishing laser scanner data [10]. Nodes exchange information via topics, services, actions, or parameters. In our design, we communicate exclusively over topics.

2.1.2 Topics

Topics are the primary mechanism for data transfer between nodes. Each node can publish to multiple topics and subscribe to multiple topics simultaneously.

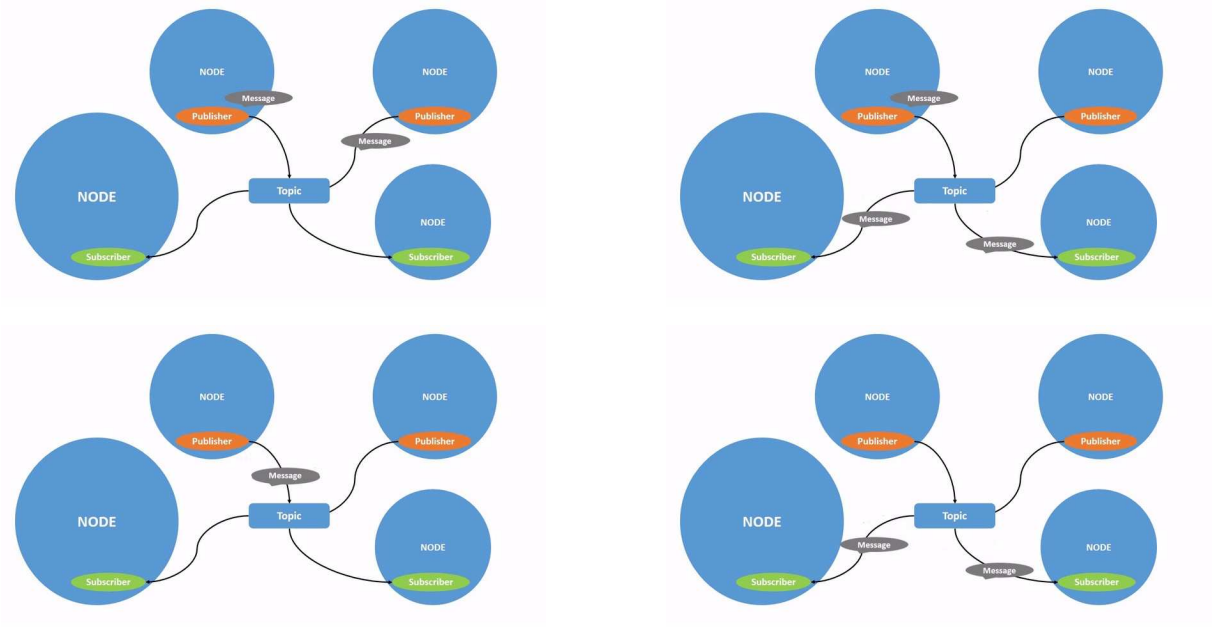


Figure 2.1: Visualization of ROS 2 nodes and their topic connections.

2.1.3 Lifecycle Nodes

Managed (lifecycle) nodes provide explicit control over the state of each component, ensuring that all nodes are properly initialized before they execute their behavior and allowing on-line restart or replacement [11]. Figure 2.2 illustrates the four standard lifecycle states:

- **Unconfigured**: Default state after node creation.
- **Inactive**: Node has been configured but is not processing data.
- **Active**: Node is running and handling its tasks.
- **Finalized**: Node has been shut down and cleaned up.

The lifecycle state transitions are triggered by service calls in accordance with the schema.

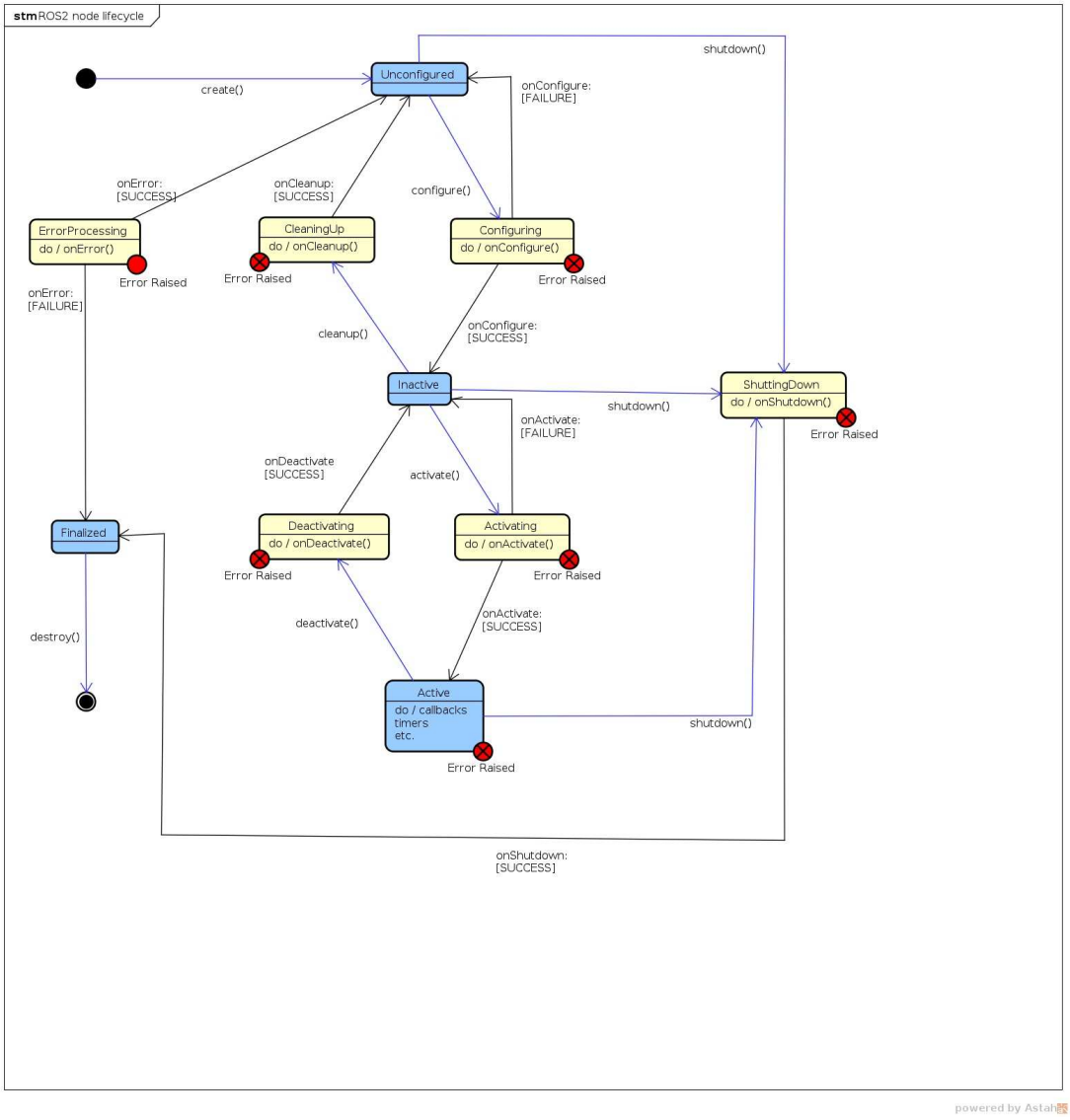


Figure 2.2: State machine for a ROS 2 lifecycle node.

2.2 Software System Design

Figure 2.3 shows the high-level architecture of our software system. Red rectangles denote nodes, grey rectangles denote message types, and yellow trapezoids denote topics. "Processing" nodes (shown in dashed outline) run on the Jetson Nano to leverage its CUDA cores, while all other nodes run on the Raspberry Pi 5.

The data flow is as follows:

1. **cam_left_node** and **cam_right_node** capture frames from the left and right cameras and publish them as `sensor_msgs/CompressedImage` on the `/cam_left/frame` and `/cam_right/frame` topics, respectively.

2. **preprocessing_node** subscribes to both camera topics, synchronizes the two frames, and publishes a `stereo_msgs/ImagePair` message (custom: it contains left frame, right frame, and timestamp) on `/stereo/image_pair`.

3. On the Jetson Nano, three processing nodes subscribe to `/stereo/image_pair`:

    - **MonoIRNode**: Performs mono-vision inference on one frame and publishes JSON results to `/output/cv_mono_ir`.
    - **StereoIRNode**: Runs stereo-vision inference on both frames and publishes JSON results to `/output/cv_stereo_ir`.
    - **StereoVONode**: Executes stereo visual odometry on both frames and publishes JSON results to `/output/cv_stereo_vo`.

4. **normalizator_node** subscribes to all three output topics, reformats each JSON payload into the message structure required by the acoustic modem, and publishes to `/to/modem`.

5. **modem_driver_node** subscribes to `/to/modem` and transmits each message to the ground station via the acoustic modem.

QoS settings:

- `/cam_left/frame`, `/cam_right/frame`, `/stereo/image_pair`:
  `BEST_EFFORT`, depth = 1 (minimizes latency by discarding old frames).
- All other topics:
  `RELIABLE`, depth = 10 (guarantees that each processed frame is delivered).

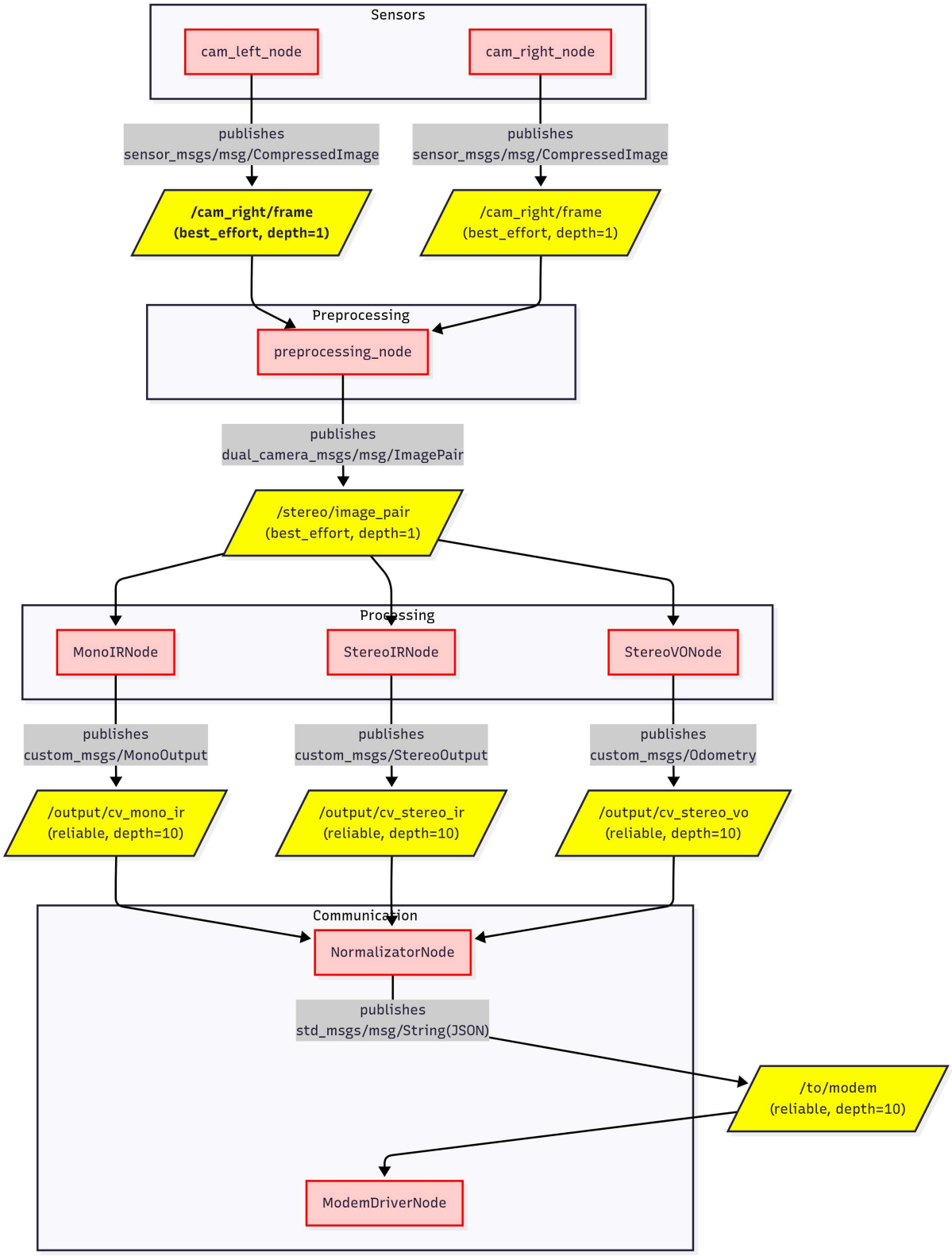


Figure 2.3: High-level ROS 2 node and topic architecture.

2.3 DDS and Data Throughput

We compared two DDS implementations—Fast DDS and Cyclone DDS—and chose Fast DDS for its superior performance under high data rates [12].
It's important to choose wisely the frame rate and frame size, as well as the compression percentage. Sending uncompressed 2304×1296 RGB frames at 60 fps would require:

$$\frac{2304 \times 1296 \times 3 \text{ bytes}}{\text{frame}} \times 60 \frac{\text{frames}}{\text{s}} \approx 537{,}500{,}000 \frac{\text{bytes}}{\text{s}} \approx 512 \text{ MB/s}.$$

By reducing resolution to 640x480 (≈900 kB raw) at 15 fps, the raw throughput drops to ≈13 MB/s. With JPEG compression at quality 80 (empirical compression factor ≈ 29× [13]) each frame becomes ≈31 kB, yielding ≈0.5 MB/s (30 MB/min). This matches our measurements.

2.4 Frame Synchronization

The **preprocessing_node** uses ROS 2's `message_filters::ApproximateTimeSynchronizer` to align left and right frames by their timestamps [14]. Frames whose timestamp difference exceeds a configurable threshold are dropped. Empirical tests confirm reliable synchronization.

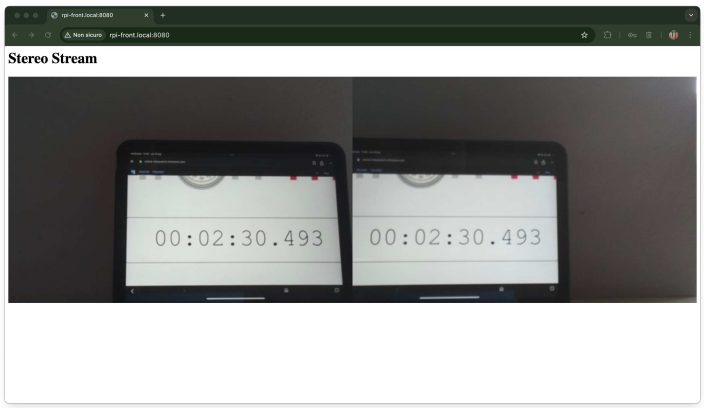


Figure 2.4: Timestamp alignment of left and right frames using the ApproximateTimeSynchronizer.

2.5 Lifecycle-Based Acquisition

Figure 2.5 illustrates the lifecycle design for `cam_left_node` (`cam_right_node` is equivalent). A central **lifecycle_manager** launch file instantiates both the camera node and the LifecycleManagerNode. The manager transitions the camera node from *unconfigured* → *inactive* → *active*,

retrying any failed transition indefinitely. If the camera node crashes (e.g. freezes), the manager detects the *finalized* state and restarts it automatically, greatly improving fault tolerance.

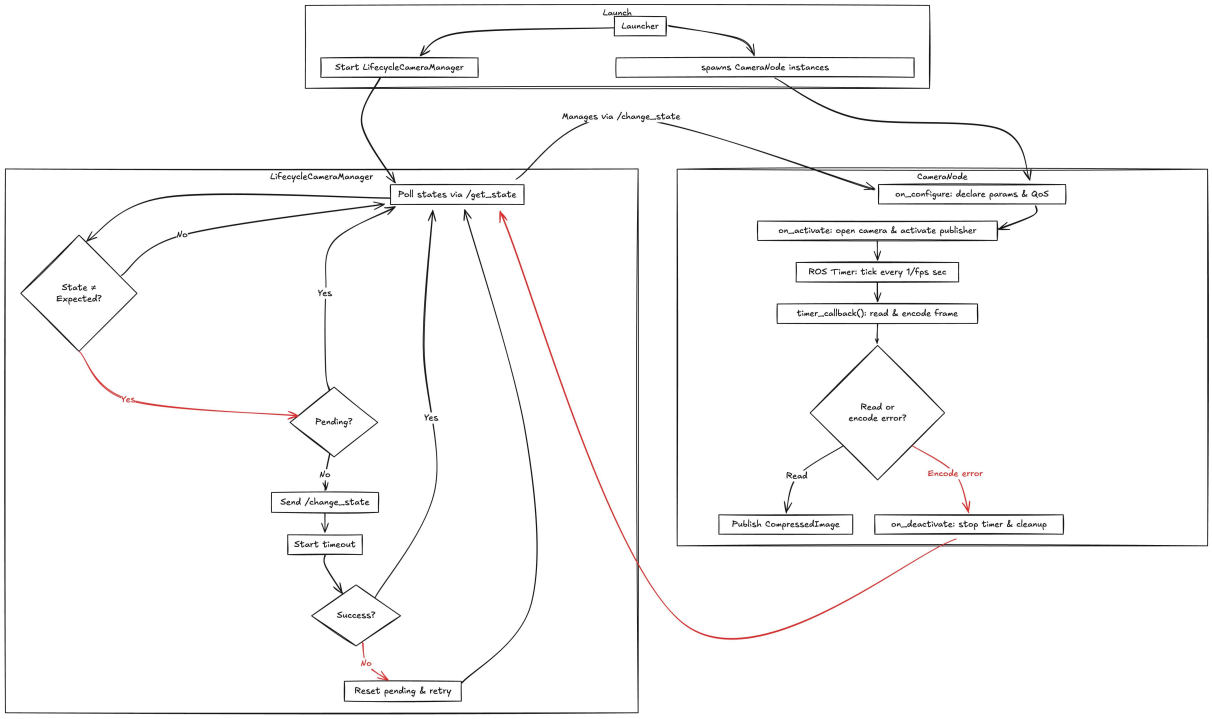


Figure 2.5: Lifecycle management for the camera acquisition node.

2.6 Development Environment and Automation

To ensure consistent and collaborative development across heterogeneous embedded platforms, a modular infrastructure based on **Docker** and **GitHub Actions** was implemented. Dedicated containerized environments were created for both the *Raspberry Pi* and the *Jetson Nano*, with automated CI pipelines for building and publishing images.

2.6.1 Repository Structure

The system is organized into three main repositories:

- `raspberry-setup`: contains the Dockerfile and configuration for the Raspberry Pi container.
- `jetson-nano-setup`: contains the Dockerfile and configuration for the Jetson Nano container.

- `sensing-rigs-ros2`: hosts the main ROS 2 workspace and the `docker-compose.yml` file used to launch the appropriate containers.

This separation allows each image to be customized and built independently according to the hardware-specific requirements of each platform.

2.6.2 Containerization for Raspberry Pi

The `raspberry-setup` repository defines a Docker image based on `ros:jazzy`, enriched with all the necessary dependencies for ROS 2, OpenCV, GStreamer, and camera support. It includes a multi-stage build process to compile and install:

- `libcamera`, built from source with GStreamer and V4L2 support.
- `rpicam-apps`, configured with OpenCV support.

A custom `entrypoint.sh` script is used to automatically build the ROS 2 workspace on first container startup, if needed.

2.6.3 Containerization for Jetson Nano

The `jetson-nano-setup` repository defines a similar Docker image, also based on `ros:jazzy`, but with a lighter setup that excludes components unnecessary for the Jetson platform.
Note that this container requires further development to ensure compatibility with OpenCV built with CUDA support. View more details on Section 3, which talks about issues with the Jetson Nano.

2.6.4 Automated Build with GitHub Actions

Both repositories include a GitHub Actions workflow that builds and publishes the Docker image to the GitHub Container Registry. The workflow is triggered on every push to the Dockerfile or the workflow configuration itself.

- It runs on an ARM-compatible Ubuntu runner (`ubuntu-24.04-arm`).
- It builds and pushes the image for the `linux/arm64` platform.

This automation ensures that the latest changes are always reflected in the published container images.

2.6.5 Central ROS 2 Workspace and Orchestration

The `sensing-rigs-ros2` repository hosts the shared ROS 2 codebase inside a common `ros2_ws` workspace. The deployment of the system is managed via a `docker-compose.yml` file, which orchestrates the containers for both the Raspberry Pi and the Jetson Nano platforms.

Each container is configured independently to match the hardware and software requirements of its respective platform. In particular:

- The container on the Raspberry Pi is launched with settings tailored for that platform, enabling direct access to MIPI cameras through device volume mapping.
- The container on the Jetson Nano is configured accordingly and runs only on that board.

This separation allows each container to operate independently on its specific hardware, while maintaining a shared codebase and architecture.

2.6.6 Benefits

This setup provides several advantages:

- Consistent development environments across machines and developers.
- Simplified onboarding for new contributors.
- Automated, reproducible builds with minimal manual intervention.
- Separation of concerns between platform-specific dependencies and shared application code.

Chapter 3

Prototype Testing Session

In order to collect realistic datasets, we organized a Prototype Testing Session in Chioggia.
The outing helped us better understand the strengths and weaknesses of our system.

Strengths

Ease of Use

We automated most of the system, making it extremely simple to set up and run.
After connecting the Raspberry Pi to the same network of the computer (via Ethernet or WiFi), the user only needs to:

- Connect via SSH
- Launch the container with a single command (`./start_system.sh`)
- Once inside, launch the system with another command (`./launch.sh`)

The latter command starts all necessary ROS2 nodes and opens an HTTP server that displays stereo images in real-time.
All stereo images can be recorded using the standard ROS2 command `ros2 bag`, which stores all messages published to a given topic. This allows us to replay the recorded data as if it were being received from the cameras in real-time.

Modularity

The system worked well for both the **Sensing-Rigs** and **Autodocking** projects.
Both teams were able to use the same setup with only minor adjustments.

Ease of Setup

Since all Docker images were already built and stored on GitHub, setting up a new Raspberry Pi took between 10 and 20 minutes.
The most time-consuming part was installing the Raspberry Pi OS on the SD card and downloading the Docker image for the first time.

Weaknesses

Jetson Nano

The Jetson Nano was not ready for the test day.
The main issue is that, to run ROS2 Jazzy, we are using a Docker container based on Debian Bookworm. While the container runs on the Jetson, it doesn't have access to the CUDA cores. Our code uses OpenCV for computer vision tasks, but to utilize the Jetson's CUDA cores, a specific CUDA-compatible version of OpenCV must be used. Such a version is not easily installable via `pip` or `apt`, as no precompiled packages are available. It must be built from source. Additionally, the required OpenCV version depends on Python 3.6, while ROS2 Jazzy requires Python 3.11.
A possible workaround would be to install an older ROS2 version on the Jetson that is compatible with Python 3.6. The communication with the Raspberry Pi would still work, since ROS2 nodes interact via DDS, which is independent of the ROS2 version.

Due to time constraints, we decided to focus on recording data and address this issue later.

Power Issues

The test box was too small to accommodate the power bank intended to power the Raspberry Pi.
To resolve this, we placed the power bank outside the box, as shown in Figure 3.1 (the battery is inside the transparent box).
Initially, a cap with pass-through cables (Ethernet and power) sealed with epoxy resin was designed, but it became useless due to the next issue.

Water Tightness

The box was designed with a 1 mm diameter silicone gasket to seal it when closed.
A groove of less than 0.5 mm depth was added both on the body and on the cap.
Unfortunately, the 3D printer couldn't maintain the required precision, and the groove ended up too deep on both parts.

This caused the gasket to sink completely into the groove, rendering it ineffective.
Potential solutions would be to purchase a larger gasket or redesign the box. However, neither option was feasible within the available time.
As a temporary workaround, we left the box open and operated with extreme caution near water.

Heat

In direct sunlight, the Raspberry Pi heats up significantly. A heatsink is essential; without it, the Raspberry Pi 5 shuts down after a few minutes due to overheating.
The dark color of the box also contributed to heat retention, an aspect that should be reconsidered in future designs.

Results

We were able to record a large number of images and test the computer vision algorithm (though only on the Raspberry Pi 5, not the Jetson).
Figure 3.2 shows the crab detection algorithm identifying a real crab.
Figures 3.3 and 3.4 show two moments during real-world data recording.
Figures 3.5 and 3.6 are example frames from the recorded footage.
A closer look at our setup is shown in Figure 3.7.
The setup used by the Autodocking group is shown in Figure 3.8, and some of their results are presented in Figures 3.9 and 3.10.

Figure 3.1: Close-up of the Autodocking setup.

Figure 3.2: Crab detected by the CV algorithm.

Figure 3.3: Recording session – moment 1.

Figure 3.4: Recording session – moment 2.

Figure 3.5: Recorded stereo image with scallop

Figure 3.6: Recorded stereo image with two crabs

Figure 3.7: Close-up of the main recording setup.

Figure 3.8: Setup used by the Autodocking group.

Figure 3.9: Autodocking experiment – back.

Figure 3.10: Autodocking experiment – front.

Bibliography

[1] u/supersuperpotato, *Nvidia jetson nano or raspberry pi 4 + google coral?* Reddit post, r/JetsonNano, Accessed: 2025-07-09, 2020. [Online]. Available: https://www.reddit.com/r/JetsonNano/comments/jouz00/nvidia_jetson_nano_or_raspberry_pi_4_google_coral/?show=original.

[2] u/cheesemover69, *Should i get a google coral usb accelerator for yolo?* Reddit post, r/computervision, Accessed: 2025-07-09, 2023. [Online]. Available: https://www.reddit.com/r/computervision/comments/1161mab/should_i_get_a_google_coral_usb_accelerator_for/.

[3] T.-M. Wang and Z.-C. Shih, "Measurement and analysis of depth resolution using active stereo cameras," *IEEE Sensors Journal*, vol. 21, no. 7, pp. 9218–9230, 2021. DOI: 10.1109/JSEN.2021.3054820.

[4] Edmund Optics, *Understanding focal length and field of view*, Accessed: 2025-07-09, n.d. [Online]. Available: https://www.edmundoptics.com/knowledge-center/application-notes/imaging/understanding-focal-length-and-field-of-view/.

[5] S. O. user Stphn, *Answer to "can i run yolov5 on raspberry pi 4 with coral usb accelerator?"* Answer on Stack Overflow, Accessed: 2025-07-09, 2023. [Online]. Available: https://stackoverflow.com/a/75745742.

[6] J. Geerling, *New 2gb pi 5 has 3% smaller die, 30% idle power savings*, Accessed: 2025-07-11, 2024. [Online]. Available: https://www.jeffgeerling.com/blog/2024/new-2gb-pi-5-has-33-smaller-die-30-idle-power-savings#:~:text=8.9W-,9.8W,-0.9W%20(%2B10%25).

[7] OpenDataCam Documentation, *Jetson nano power modes*, Accessed: 2025-07-11. [Online]. Available: https://opendatacam.github.io/opendatacam/documentation/jetson/JETSON_NANO.html#:~:text=Jetson%20Nano%20has%20two%20power%20mode%2C%205W%20and%2010W..

[8] Raspberry Pi Documentation, *Camera module power requirements*, https://www.raspberrypi.com/documentation/computers/camera_software.html, Accessed: 2025-07-11. The Camera Module adds about 200–250 mA to the power requirements of your Raspberry Pi.

[9] Blue Robotics, *Fathom rov tether (rov ready)*, Accessed: 2025-07-11. [Online]. Available: https://bluerobotics.com/store/cables-connectors/cables/fathom-rov-tether-rov-ready/.

[10] Open Robotics, *Understanding ros 2 nodes*, https://docs.ros.org/en/jazzy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html, Accessed: 2025-07-12, 2023.

[11] Open Robotics, *Ros 2 node lifecycle*, https://design.ros2.org/articles/node_lifecycle.html, Accessed: 2025-07-12, 2023.

[12] D. Cabezas, *New fast dds performance testing*, ROS Discourse forum post, Available at: https://discourse.ros.org/t/new-fast-dds-performance-testing/29539, accessed Jul. 2025, Jan. 2023.

[13] Aurigma Inc., *Compression ratio for different jpeg quality values*, Accessed: 2025-07-12, Nov. 2014. [Online]. Available: https://www.graphicsmill.com/blog/2014/11/06/Compression-ratio-for-different-JPEG-quality-values.

[14] Open Source Robotics Foundation, *Message_filters (python api) — ros lunar documentation*, Accessed: 2025-07-14, 2017. [Online]. Available: https://docs.ros.org/en/lunar/api/message_filters/html/python/index.html.