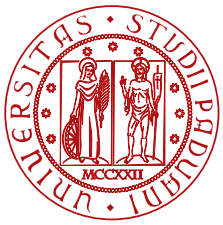

UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

BACHELOR'S DEGREE IN COMPUTER ENGINEERING

# Design and Implementation of a ROS2-based Vision Pipeline for Autonomous Docking: Integration and Performance Evaluation using Raspberry Pi 5 and Jetson Nano


**Supervisor**
Varagnolo Damiano

**Candidate**
Pavanetto Marco



ACADEMIC YEAR 2024-2025

Graduation date 23/07/2025

*A special thanks to everyone who believed in me.*

# Abstract

This thesis presents the design and implementation of a ROS2-based visual data pipeline to support autonomous docking functionalities in small-scale robotic boats. The proposed system focuses on real-time image acquisition using the Raspberry Pi Camera Module 3, integrated via a custom Python ROS2 node running on a Raspberry Pi 5. The captured images are transmitted to the "Open Water" segmentation algorithm, developed with PyTorch by a member of the Autodocking team, which classifies image regions into water, sky and urban elements. Due to the computational limitations of the Raspberry Pi 5, the architecture was progressively evolved to delegate processing tasks to an NVIDIA Jetson Nano, introducing a modular and distributed design. To address system compatibility, Docker containers were used to harmonize dependencies across heterogeneous hardware and operating systems. Field testing in Padua and Chioggia validated system performance in real-world aquatic environments, highlighting key challenges such as thermal management and segmentation inaccuracies caused by environmental reflections. Although full GPU acceleration on the Jetson Nano was not achieved, the groundwork has been laid for future improvements. The system's modularity and documentation aim to facilitate further development toward autonomous navigation and docking.

# Contents

# List of Figures

# Chapter 1

# Introduction

In recent years, the field of autonomous systems has made significant strides, driven by advancements in embedded computing, computer vision and robotic middleware frameworks. Among these, Robot Operating System 2 (ROS2) has emerged as one of the most widely adopted platforms for developing robotic applications. Its modular architecture, support for real-time communication and open-source ecosystem make it an excellent choice for building scalable and flexible robotic systems.

This thesis is part of the Autodocking project focused on enabling autonomous docking capabilities for a small blue boat. While the boat's structure and basic control systems were already in place, it lacked a vision-based system to support autonomous navigation. My contribution to the project was the development of the image acquisition and communication pipeline using ROS2, a foundational component for realizing vision-based autonomy.

To achieve this, I needed a ROS2 node in Python to interface with the *Raspberry Pi Camera Module 3*. The node needs to be designed to continuously capture images and publish them into a topic as ROS2 messages, allowing other nodes to access and process the data. Initially, both image acquisition and processing were performed on a single Raspberry Pi 5. However, due to performance limitations, the architecture was restructured to offload image processing to a Jetson Nano, a more powerful device capable of handling computationally intensive tasks. In the revised setup, the Raspberry Pi 5 remained responsible for image capture and publication, while the Jetson Nano subscribed to the image stream and executed the computer vision algorithm.

The processing component, developed by another team member, was an "Open Water" algorithm built using the PyTorch machine learning library; this module analyzed the incoming images to divide them in three parts: water (of a river or the sea), buildings and the sky. This division of labor between the Raspberry Pi 5 and Jetson Nano introduced a modular, distributed architecture that enhanced maintainability and scalability.

This thesis focuses on the design, implementation and evaluation of the ROS2-based image

acquisition node, along with the integration efforts required to ensure reliable communication between the Raspberry Pi and Jetson Nano. By providing a robust visual data pipeline, this work lays the groundwork for further development of the boat's autonomous navigation capabilities.

# Chapter 2

# Initial work

At the very beginning of the project, we had planned to use an *Asus Tinker Board S* for the initial development work. The idea was to get started with ROS2 Humble Hawksbill on that board while waiting for the *Raspberry Pi 5*, which was meant to be our actual target hardware. Since the Tinker Board S is also a single-board computer, we thought it could be a good placeholder to start building the image acquisition system.

However, this approach didn't last long. We quickly found out that the Tinker Board S is a 32-bit ARM architecture and this turned out to be a major issue. Not only was it hard to find an operating system image that would work well with it, but most of the recent ROS2 distributions, including Humble, are designed with 64-bit systems in mind. Compatibility problems started piling up, and it became clear that continuing with the Tinker Board would have cost us more time than it was worth. So we decided to drop that path and wait for the Raspberry Pi 5 to arrive.

As soon as the *Raspberry Pi 5* was in our hands, we switched gears and focused on setting it up. Thanks to its much more powerful hardware compared to previous models, it was the right platform for our project. After some research we came to the conclusion that the best choice will be to install *Ubuntu 24.04*, which at the time was the latest LTS version available. To flash the OS on the SD card we used Raspberry Pi Imager, which is a trivial and error prone way to install an operating system that will then be used on a Raspberry Pi. Based on that choice, we also decided to use ROS2 Jazzy Jalisco, since it's the most recent Long Term Support version of ROS2 which is compatible with Ubuntu 24.04.

Once the operating system and ROS2 were up and running, I started looking into how to make the *Raspberry Pi Camera Module 3* work properly with the new board. After some online searching, I learned that I needed two key components: the official Raspberry pi `libcamera` fork and `rpicam-apps`. These are essential for accessing the camera hardware and capturing images on the Pi 5.

So to get everything working, I followed the official guide from the Raspberry Pi website,

which explains how to build and install both `libcamera` and `rpicam-apps` from source. While the process wasn't overly complicated, it did require adding a few packages to resolve some dependency issues and making sure everything was configured properly for the version of Ubuntu we decided to use. Then I connected the ribbon cable of the camera to the Raspberry to check that I didn't made any mistake and thus it could correctly see the image stream.

As I went through the installation steps, I kept detailed notes of everything I did, including the specific commands, potential pitfalls and how to troubleshoot some received warnings. I later turned those notes into a short and easy to follow guide for the rest of the group, so that they could reproduce the setup without running into the obstacles I encountered.

This initial setup phase was important because it established the foundation for everything that came afterwards. With a now working camera and ROS2 running smoothly, we had a solid starting point for integrating the image processing part, which was being developed by another teammate.

# Chapter 3

# Sending images to the ”Open Water” algorithm

At this stage of the project, the primary objective was to develop the necessary ROS2 code to facilitate the integration between the images captured by the *Raspberry Pi Camera Module 3* and the *Open Water* algorithm. The overarching goal was to ensure a seamless, reliable and efficient transfer of image data from the acquisition hardware to the subsequent processing by the advanced computer vision algorithm, thereby forming a robust foundation for the project's broader goals in autonomous navigation.

Given the significance of establishing a stable and robust communication channel between the ROS2 environment and the external image processing module, it was crucial to dedicate sufficient time, careful planning and resources towards achieving an optimal implementation. To begin, I systematically explored various available solutions to assess whether existing ROS2-compatible camera acquisition packages could be leveraged effectively, significantly streamlining the development process and avoiding redundant coding efforts.

Through extensive online research and community-driven resources, I successfully identified a highly suitable pre-existing solution, specifically this well-maintained ROS2 node. This particular node offered extensive compatibility and comprehensive support for numerous camera models, notably including Raspberry Pi camera modules. Its functionality was based on the previously installed `libcamera` package, a versatile and widely adopted camera control library. By leveraging an existing, thoroughly tested solution, I effectively reduced the overall development time and mitigated the risk of encountering significant implementation errors related to camera interfacing.

Following the identification and evaluation of this robust solution, I proceeded with the installation of the pre-built ROS2 node onto the Raspberry Pi 5 platform. The installation process was straightforward and well-documented, enabling rapid deployment. As a result, I quickly

transitioned from configuration and setup activities to the immediate development of integration components tailored specifically for the computer vision algorithm.

## 3.1 First naive implementation

To effectively initiate the development of my ROS2 node, I dedicated considerable effort towards studying the official ROS2 documentation. This involved gaining familiarity with ROS2's core architecture, essential functionalities, established best practices and recommended design patterns. Having acquired this foundational knowledge, I felt adequately prepared to begin writing my node using Python, chosen primarily due to its robust integration with ROS2, extensive community support, ease of learning and suitability for rapid prototyping.

My initial implementation adopted a deliberately straightforward approach to provide a solid and manageable starting point. To initiate the execution of the main Python script associated with the segmentation algorithm directly from within my ROS2 node, I utilized the Python *subprocess* module. This enabled efficient and reliable execution of terminal commands inside the Python code. Specifically, by invoking the external script as a subprocess, the ROS2 node effectively managed its execution lifecycle, simplifying initial debugging and performance tuning tasks.

Following the development of this initial setup, I conducted comprehensive performance evaluations to precisely quantify the processing time required by the Raspberry Pi 5 to analyze a single image using the *Open Water* algorithm. Through systematic empirical testing, I established that the typical processing duration for each individual image averaged between 50 and 55 seconds. Armed with these concrete performance metrics, I subsequently implemented a simplistic, yet effective timing mechanism within my ROS2 node. This involved hardcoding a fixed interval value based on the average processing duration, ensuring the subprocess execution started only after the completion of the preceding image processing task.

While operationally effective at this preliminary stage, this initial implementation approach was consciously simplistic. It served primarily as a functional baseline to evaluate basic integration success and identify critical bottlenecks. My intention moving forward was to iteratively refine and progressively enhance the implementation, significantly improving efficiency, reliability, scalability and maintainability of the entire image processing integration pipeline.

## 3.2 Advanced Refinements and Optimization

### 3.2.1 Direct Integration of the Algorithm

Following the successful establishment of the initial implementation, I began refining and optimizing my ROS2 node. My first major improvement was eliminating the subprocess approach, which, although functional, was suboptimal in terms of efficiency and maintainability. Instead, I directly imported the Python script containing the *Open Water* algorithm and attempted to invoke its main function directly. However, this initial integration encountered difficulties, primarily because the original computer vision algorithm was designed to run from the terminal using specific command-line flags and arguments.

To resolve this, I coordinated with the team member responsible for writing its code, requesting modifications to allow the algorithm to run effectively as an imported Python module, accepting parameters directly from function calls rather than command-line arguments. After receiving the updated version, I immediately integrated the revised code into my ROS2 node. Initial tests suggested successful integration, but closer inspection revealed inaccuracies in the processed images since they were all very similar even when the input images were different.

### 3.2.2 Resolving Image Encoding Issues

Through a collaborative debugging effort, it became clear that the underlying issue stemmed from differences in image encoding formats. The images obtained from ROS2 topics were initially converted into OpenCV (cv2) images encoded in the BGR color space, whereas the segmentation algorithm required images encoded in RGB and formatted as PIL images. To address this discrepancy, I developed a dedicated image conversion function, efficiently transforming cv2 images into PIL format and adjusting the color encoding from BGR to RGB. After integrating this conversion step, comprehensive tests confirmed that the processed images were now correctly interpreted by the algorithm.

### 3.2.3 Enhanced Configuration and Logging

To further enhance usability and monitoring capabilities, I introduced a configurable ROS2 boolean parameter named `save_image`. This feature provided flexibility by allowing the conditional saving of input images to a predefined directory when the parameter was set to `True`. Independently, the processed output images were consistently saved locally, facilitating subsequent analysis and debugging.

Moreover, I refined the `listener_callback` function within the ROS2 node to dynamically trigger image processing as soon as the previous processing task completed, effectively replac-

ing the previously hardcoded timing intervals. Additionally, I implemented detailed logging functionality to provide real-time terminal output, clearly indicating the time taken to process each image. These enhancements proved invaluable in monitoring and optimizing the performance of the segmentation algorithm throughout ongoing development and testing activities.

# Chapter 4

# Experimental Field Tests and Analysis

Having successfully implemented and integrated the ROS2-based image acquisition and processing system, the next critical step was conducting real-world tests to evaluate the practical performance of our solution. To achieve this, we planned a series of field tests in Padua along the Piovego river. This environment was selected specifically for its varied features, including reflective water surfaces, distinct architectural elements and diverse lighting conditions, all of which would thoroughly challenge the robustness and accuracy of our image-processing pipeline.

## 4.1 Setup and Initial Field Conditions

The testing apparatus consisted of the Raspberry Pi 5 equipped with two Raspberry Pi Camera Modules, securely housed within a 3D printed box. The system was placed near the river bank, ensuring consistent image acquisition during testing. The ROS2 node responsible for the image capture and initial handling was executed without modifications, precisely mirroring the operational conditions anticipated during actual deployment.

## 4.2 Performance Evaluation using the Lightweight Segmentation Model

Initially, tests were performed using the ”light” version of the trained weights, prioritizing execution speed and computational efficiency over accuracy. Several representative images were captured, each subjected to analysis by the Open Water algorithm.

### 4.2.1 Test Results

The following images illustrate some of the results obtained:

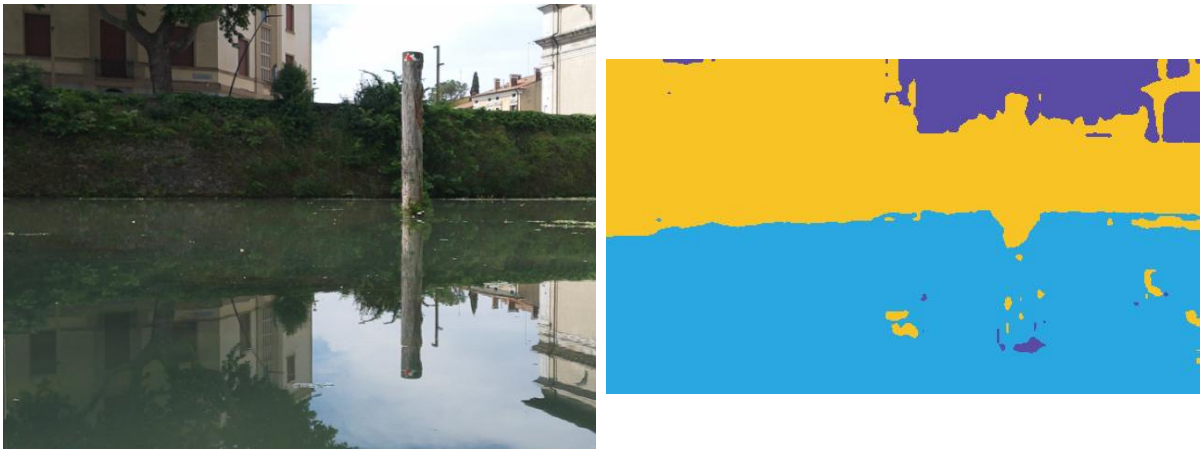

Figure 4.1: Raw input image (left) and corresponding segmentation output (right) using the lightweight model – first example.

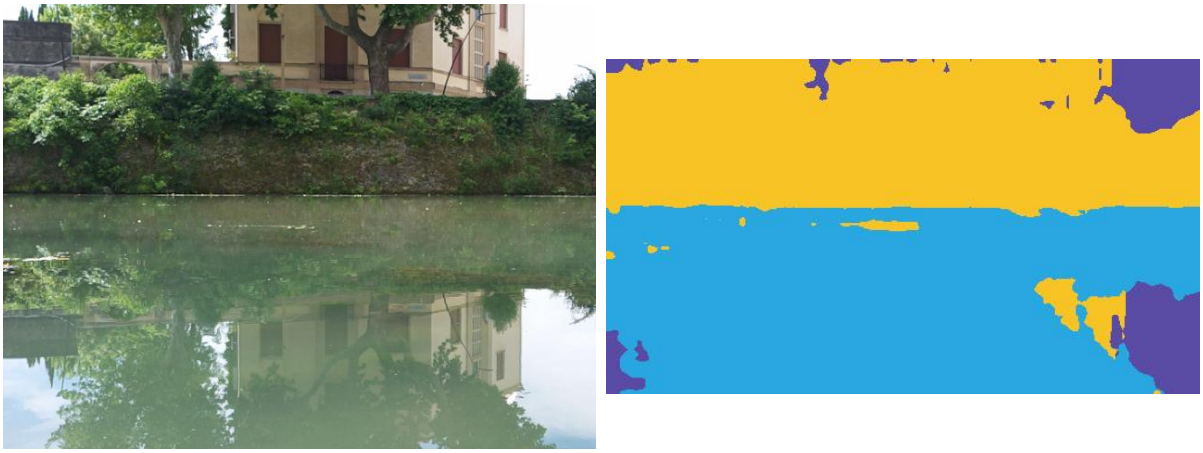

Figure 4.2: Raw input image (left) and corresponding segmentation output (right) using the lightweight model – second example.

Upon visual inspection, several notable issues became apparent. The segmentation boundaries identified by the algorithm exhibited rough, imprecise edges. Additionally, reflective water surfaces frequently introduced errors, causing misclassifications particularly evident where reflections from the water were mistaken for sky or structural elements. This confusion underscores a significant limitation of the lightweight model in handling complex reflective environments.

## 4.3 Performance Evaluation using the Heavyweight Segmentation Model

To improve accuracy, the tests were repeated using a more computationally demanding version of the trained model, by using the ”heavy” weights version. This variant prioritizes accuracy and robustness at the expense of increased processing time and computational load.

### 4.3.1 Test Results

Again, we obtained multiple sets of representative images:

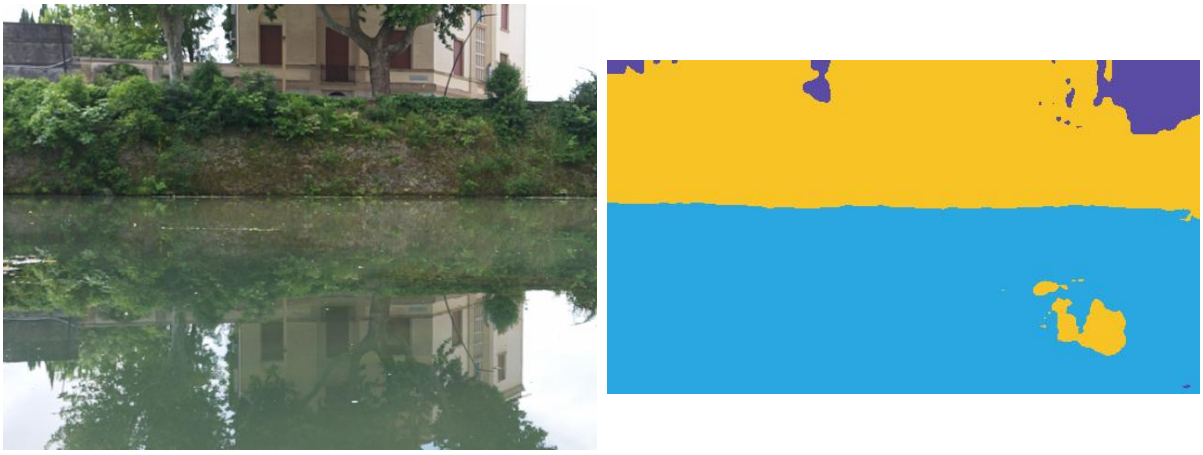

Figure 4.3: Raw input image (left) and corresponding segmentation output (right) using the heavyweight model – first example.

Figure 4.4: Raw input image (left) and corresponding segmentation output (right) using the heavyweight model – second example.

The processed images using the heavier weights demonstrated some improvements. Edges were notably more refined and accurate, indicating enhanced segmentation performance. Moreover, the incidence of misclassification due to reflections, while still present, was substantially reduced. Overall, fewer imperfections and clearer distinctions between water, sky and buildings were observed, highlighting the superiority of the heavier model in environments with challenging reflective conditions.

## 4.4 Comparative Analysis and Discussion

Comparing the two test conditions clearly reveals a trade-off between computational efficiency and segmentation accuracy. While the lighter weights provide faster processing, their accuracy deteriorates notably in complex environments. Conversely, the heavier weights, despite their increased computational demand, offer greater precision and robustness, essential for reliable autonomous navigation.

This analysis confirms the necessity of strategically selecting the appropriate model weights according to the specific requirements and constraints of the deployment environment. For future deployments, adaptive or hybrid approaches might be explored, balancing performance and computational constraints dynamically based on environmental conditions and system state.

# Chapter 5

# Comprehensive Integration of Jetson Nano and ROS2 for Enhanced Image Processing

## 5.1 Motivation for using Jetson Nano

In the initial phases of the project, it became evident that the image processing demands exceeded the capabilities of our existing hardware setup. While the Raspberry Pi 5 was effective for image acquisition tasks, its computational limitations presented a significant bottleneck when handling more complex processing operations. Consequently, the decision was made to integrate a *Jetson Nano*, a device specifically designed to perform computationally intensive tasks efficiently. Our primary objective was clearly defined: the Raspberry Pi 5, equipped with ROS2 Jazzy, would acquire images, while the Jetson Nano would handle intensive processing tasks using the sophisticated *Open Water* algorithm.

### 5.1.1 Benefits of the Jetson Nano Platform

The Jetson Nano platform, developed by NVIDIA, offers notable advantages for applications involving image processing and machine learning algorithms. Its GPU-oriented architecture allows for accelerated computing, significantly improving performance over traditional CPU-based devices. This capability aligns well with the computational demands of the segmentation algorithm, making the Jetson Nano an ideal choice.

## 5.2 Operating System Considerations for Jetson Nano

Selecting an appropriate Operating System (OS) for the Jetson Nano represented the first practical challenge in the integration process. Consistency across hardware platforms was our primary

goal, thus prompting us to initially aim for Ubuntu 24.04, the same OS version utilized by the Raspberry Pi 5.

### 5.2.1 Challenges with OS Compatibility

Despite our efforts, we encountered significant challenges in sourcing an appropriate version of Ubuntu 24.04 compatible with the Jetson Nano. Various community-driven distributions initially appeared promising; however, subsequent attempts to flash these onto the Jetson Nano consistently resulted in boot failures. These obstacles significantly delayed our progress, forcing us to reconsider our approach.

### 5.2.2 Exploring Official OS Solutions

In response to these setbacks, we shifted focus towards official solutions provided by NVIDIA. Detailed research eventually identified an officially supported distribution: Ubuntu 18.04. After carefully following the official installation guide, we downloaded and flashed Ubuntu 18.04 onto an SD card. Subsequently, the OS was successfully installed in the Jetson Nano, marking a crucial step forward in our integration efforts.

## 5.3 Resolving Cross-Platform Compatibility Issues

With the Raspberry Pi 5 running Ubuntu 24.04 and the Jetson Nano utilizing Ubuntu 18.04, compatibility issues soon became apparent. Differences between OS versions posed potential risks of discrepancies within the ROS2 Jazzy environment, which could compromise the integrity and consistency of data processing.

### 5.3.1 Evaluating Docker as a Solution

Recognizing these challenges, Docker emerged as an ideal cross-platform solution. Docker's containerization technology allows applications to operate within a consistent software environment, abstracting away discrepancies caused by differing OS versions.

### 5.3.2 Docker Container Implementation

A proactive initiative from a member of the ROS2 subgroup within the Sensig-Rigs team successfully addressed this compatibility issue by developing custom Docker images based on Ubuntu 24.04. Two images were expertly crafted to support the Raspberry Pi 5 and Jetson

Nano, integrating critical dependencies such as `libcamera` and `rpicam-apps`, based on a comprehensive installation guide that I had previously created. This ensured automatic installation and simplified the deployment process for end users.

Implementing this Docker solution allowed us to replace the heavier Ubuntu 24.04 with the more lightweight RaspberryPiOS on the Raspberry Pi 5, enhancing system performance without sacrificing functionality. Following the provided deployment instructions, the Docker images were successfully installed on both hardware platforms, thereby maintaining uniformity and streamlining operations.

## 5.4 Integration and Harmonization of ROS2 Nodes

To further enhance compatibility and foster collaborative efforts within the broader Nautilus project, it was essential to standardize the ROS2 codebase across different project groups.

### 5.4.1 ROS2 Codebase Standardization

A member of the Sensig-Rigs group, responsible for developing the Docker solution, also established a comprehensive GitHub repository containing a standardized collection of ROS2 nodes. This repository encompassed robust implementations for image acquisition as well as advanced computer vision algorithms tailored specifically to his team's requirements. The importance of standardization became clear, prompting us to adopt this unified ROS2 repository for our Autodocking team as well.

### 5.4.2 Adapting Existing ROS2 Nodes

To effectively transition to this standardized ROS2 repository, significant efforts were dedicated to reviewing, understanding, and adapting the existing code. My previously developed ROS2 node was subsequently modified and the image acquisition task removed due to redundancy since the new codebase already managed similar functionalities, thus simplifying maintenance and future collaboration.

This adaptation involved detailed discussions and code reviews, ensuring alignment with established best practices. While modifications to my code were required, these changes were relatively minor and primarily focused on ensuring seamless integration with existing stereo camera configurations managed by the repository.

### 5.4.3 Mono-Camera Configuration Integration

Considering our specific use-case involved a mono-camera algorithm, adaptations were strategically made to subscribe to only the ”left” camera stream within the existing stereo camera ROS2 setup. This selective approach enabled smooth and efficient integration, avoiding unnecessary complexity. Ultimately, these thoughtful modifications maintained consistency within the broader codebase, significantly facilitating ongoing and future development efforts within the Nautilus project ecosystem.

## 5.5 Optimizing Jetson Nano Performance with PyTorch and CUDA

Upon integrating my ROS2 node within the newly standardized codebase and deploying it on the Raspberry Pi and Jetson Nano setup, initial tests revealed an unexpected performance issue: the Jetson Nano was processing images more slowly than the Raspberry Pi. This anomaly prompted an in-depth analysis to pinpoint the underlying cause.

### 5.5.1 Identifying the Bottleneck

Careful investigation indicated that the primary cause for the reduced performance on the Jetson Nano stemmed from PyTorch operating without CUDA support. The absence of GPU acceleration led the Jetson Nano to rely solely on its CPU, thus significantly diminishing processing efficiency and negating its inherent computational advantages. Recognizing the critical importance of GPU support for PyTorch to fully leverage the Jetson Nano’s capabilities, I initiated extensive research into methods for enabling CUDA integration.

### 5.5.2 Exploring Solutions for PyTorch with CUDA Support

Addressing this issue required navigating complex software compatibility constraints. The Docker container on the Jetson Nano was based on Ubuntu 24.04, whereas the host operating system was Ubuntu 18.04. This disparity in Python and OS versions complicated the straightforward installation of PyTorch with CUDA support.

After thorough exploration of various resources, I identified that the ideal approach involved creating a Python virtual environment aligned with the host OS’s Python version. This virtual environment would allow compatibility with the specific CUDA version available for Ubuntu 18.04, thus ensuring GPU acceleration for PyTorch.

### 5.5.3 Implementation Attempt Using Pyenv

To achieve compatibility, I decided to use pyenv inside the Docker container of the Jetson Nano. Pyenv facilitates the management of multiple Python versions and virtual environments, making it a suitable choice for resolving our version mismatch issue. The strategy involved the following detailed steps:

1. Install pyenv within the Jetson Nano Docker container to manage Python versions independently from the Docker image's default configuration.
2. Create a Python virtual environment matching the Python version utilized by the host OS (Ubuntu 18.04).
3. Within this virtual environment, install a CUDA-compatible version of PyTorch to leverage GPU acceleration.

### 5.5.4 Encountered Challenges and Troubleshooting

Following this strategy, I discovered an extensive guide on NVIDIA's developer forums, specifically addressing PyTorch installation with CUDA support for Jetson Nano. The instructions provided were comprehensive, but during execution, I encountered multiple issues. Despite these setbacks, each error provided valuable insights, guiding iterative corrections and adjustments to the implementation procedure. Progressively resolving these issues required meticulous examination of the error logs, documentation and forum discussions, thereby expanding my understanding of the underlying complexities.

### 5.5.5 Documentation and Future Work

Ultimately, my efforts culminated in an unresolved critical error that persisted despite numerous attempts at troubleshooting. Recognizing the need to move forward with the overall project timeline, I concluded that further attempts at this stage were impractical.

However, understanding the significance of documenting this challenging process, I meticulously recorded each step, including the commands executed. This comprehensive documentation ensures that future contributors can easily resume this critical integration task without repeating previous troubleshooting steps, significantly facilitating subsequent problem-solving efforts.

The detailed instructions list I provided aimed not only to capture the current state clearly but also serves as a foundational reference to enhance future development and optimization of Jetson Nano performance within the broader Nautilus project.

# Chapter 6

# Field Testing in Chioggia: Environmental Validation and Thermal Evaluation

After the successful implementation and preliminary validation of the ROS2-based image acquisition and processing system in controlled environments, it became imperative to evaluate its performance in real-world aquatic scenarios. To this end, we conducted a two-day field testing campaign in the town of Chioggia, known for its network of canals and proximity to the sea. The objective of this field test was to collect visual data under realistic operational conditions, assess system robustness and identify practical challenges such as environmental interference and hardware limitations.

## 6.1 System Preparation and Pre-Deployment Enhancements

### 6.1.1 Code Improvements

In preparation for the field test, I executed a final round of verification and made a small but important enhancement to the ROS2 image acquisition node. Specifically, I extended the functionality of the `save_image` parameter. While its original purpose was to control the saving of unprocessed input images, the updated version also governs the saving of images processed by the segmentation algorithm. This improvement enabled better flexibility and traceability during testing, allowing us to compare raw and segmented images efficiently.

### 6.1.2 Hardware Validation

A comprehensive system check was also performed prior to departure. This involved verifying the image streams from all camera modules, ensuring correct topic publication within the ROS2

ecosystem and confirming the integrity of the data pipeline connecting the Raspberry Pi devices and the Jetson Nano.

Although the initial plan was to use the Jetson Nano for image processing, due to its hardware-accelerated inference capabilities, this approach was temporarily abandoned since at the time the Jetson was taking longer than the Raspberry to process the incoming images, caused by the lack of GPU acceleration.

## 6.2 Day 1: Data Acquisition in Canal Environment

### 6.2.1 Deployment Strategy

On the first day, the morning was dedicated to setting up the hardware. Two Raspberry Pi 5 units were used, each connected to two Raspberry Pi Camera Module 3 sensors. Both systems were powered by independent power banks and enclosed in custom-made waterproof boxes to ensure operational safety in humid or splash-prone environments.

### 6.2.2 Field Testing Procedure

In the afternoon, the entire assembly was mounted on a stand-up paddleboard (SUP), providing us with a stable and maneuverable platform for data collection in the narrow canal system of Chioggia. Team members, including myself, took turns paddling through the area to acquire a diverse dataset under varying lighting and environmental conditions.

We followed a semi-structured acquisition route and captured multiple image sets. Below are a few representative examples that include both the raw images and their corresponding outputs from the segmentation algorithm:

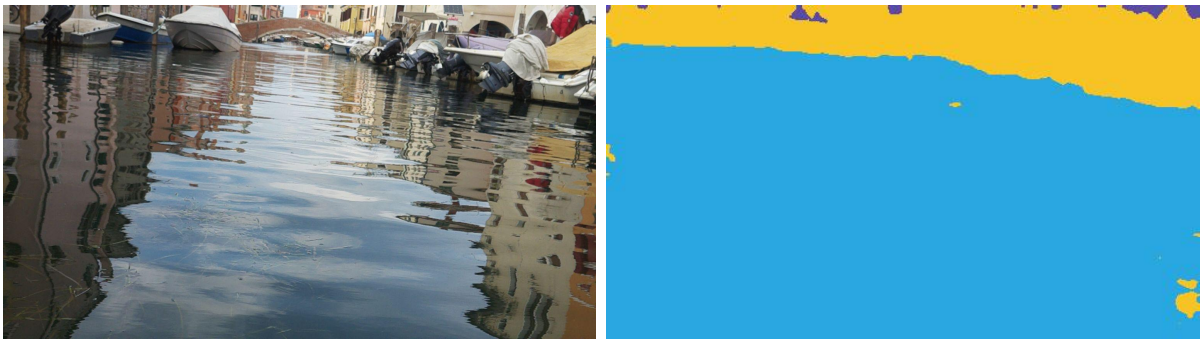

Figure 6.1: Comparison of raw image (left) and segmentation result (right) captured during canal environment testing – Case 1.

Figure 6.2: Comparison of raw image (left) and segmentation result (right) captured during canal environment testing – Case 2.

Figure 6.3: Comparison of raw image (left) and segmentation result (right) captured during canal environment testing – Case 3.

### 6.2.3 Thermal Management Issues

During this first testing session, a thermal issue emerged. One of the Raspberry Pi units, which lacked an active cooling system, shut down unexpectedly likely due to overheating. Interestingly, this occurred despite relatively moderate weather conditions (cloudy and not particularly hot). The second Raspberry Pi, equipped with an active cooling solution consisting of a heatsink and integrated fan, reached elevated temperatures but continued functioning throughout the test. This difference clearly illustrated the critical role of thermal management in maintaining system stability.

## 6.3 Day 2: Open Water Evaluation and Thermal Stress Testing

### 6.3.1 Adjustments and Setup

Drawing on the experience of the first day we replaced, for the second session, the Raspberry Pi that lacked a cooling system with another one that was equipped with an heatsink and a fan. The hardware configuration remained otherwise unchanged. This time, we chose a more exposed location, a beach area, aiming to collect data in an open water scenario, where environmental variables such as sunlight reflection, wave movement and horizon line interference could impact the performance of the computer vision algorithm.

### 6.3.2 Observed Behavior under Thermal Stress

Unfortunately, despite the addition of active cooling, the elevated ambient temperatures and direct sunlight on the second day posed severe challenges. Both Raspberry Pi units experienced noticeable performance degradation. Symptoms included increased image processing latency and multiple instances of system crashes, most likely induced by sustained thermal stress. These findings emphasized that even with cooling solutions, Raspberry Pi devices can struggle under sustained processing loads and adverse thermal conditions, especially in outdoor scenarios.

## 6.4 Insights and Lessons Learned

The field tests in Chioggia proved to be highly instructive. They offered valuable empirical insights into the operational limitations of our current hardware configuration. Key takeaways include:

- **Thermal Sensitivity:** Even moderate workloads can result in critical overheating, especially in direct sunlight. Future iterations must consider more robust cooling systems or offloading processing to more capable devices like the Jetson Nano (with GPU acceleration enabled).
- **Environmental Interference:** The presence of water reflections and dynamic lighting conditions posed challenges for the segmentation algorithm. Preliminary observations suggested that reflections were sometimes misclassified as sky or architectural structures.
- **Mobility and Platform Stability:** The SUP platform proved to be a viable and effective method for mobile data collection in shallow water environments.
- **ROS2 Robustness:** The ROS2-based image pipeline remained stable and functional during most of the trials, validating its suitability for embedded robotic applications in field environments.

These field experiments not only helped us validate the functioning of the system outside the laboratory, but they also laid the groundwork for future optimization, particularly in terms of thermal design, hardware selection and real-time performance tuning.

# Chapter 7

# Conclusion

This thesis presented the design, implementation and evaluation of a ROS2-based visual data acquisition pipeline tailored for autonomous docking applications. Developed within the broader framework of the Nautilus project, the system aimed to provide a reliable, modular and efficient solution for capturing and transmitting image data to a segmentation algorithm designed to divide an image in 3 different areas: water, sky and urban features.

The project was structured in progressive stages: from the early attempts using an Asus Tinker Board S, to the stable deployment of a ROS2 node on the Raspberry Pi 5; from the integration of the Open Water segmentation algorithm to the offloading of computationally intensive tasks to the Jetson Nano. Each phase introduced specific technical challenges, ranging from hardware-software compatibility and real-time image conversion, to containerization and field performance under thermal stress.

A significant part of the work involved overcoming cross-platform compatibility through Docker and attempting GPU acceleration using PyTorch with CUDA on the Jetson Nano. While full GPU support on the Jetson Nano was not achieved due to persistent configuration issues, all efforts were carefully documented to support future continuation of this objective. Additionally, integration into a standardized ROS2 codebase and the use of Docker images ensured long-term maintainability and ease of collaboration.

Field testing, conducted in Chioggia under real-world environmental conditions, validated both the functionality and limitations of the current system. The ROS2 image pipeline demonstrated robustness and adaptability in both canal and open water scenarios. However, hardware limitations, particularly related to thermal dissipation on the Raspberry Pi 5, emerged as a significant concern. These findings underscore the importance of proper cooling solutions and the potential advantages of offloading computation to more capable hardware such as the Jetson Nano once full CUDA acceleration is available.

**Future Work**

- **Enable GPU Acceleration with CUDA on Jetson Nano:** Future efforts should focus on finalizing the integration of PyTorch with CUDA on the Jetson Nano. Leveraging the device's GPU is crucial to achieving real-time segmentation performance, which is fundamental for autonomous navigation.
- **Thermal Management and Hardware Redesign:** Given the overheating issues encountered during field tests, especially under direct sunlight, improved thermal solutions should be designed. This could involve active cooling for all units and a review of enclosure materials to ensure better heat dissipation.
- **Integration of Fiducial Markers (AprilTags):** AprilTags represent a promising addition for future localization and docking accuracy. Their integration into the visual pipeline could provide complementary positioning data, enhancing the reliability of the entire guidance system.
- **Dynamic Configuration and Smart Logging:** More advanced logging and configuration systems should be introduced to allow for runtime tuning of image saving, model selection (light vs. heavy) and logging verbosity, depending on environmental conditions or available computational resources.
- **Extending to Autonomous Control:** Although this work focused on visual perception, it provides a solid foundation for subsequent teams to integrate perception data into navigation and control subsystems. This will be essential to close the perception–action loop necessary for full autonomy.

In conclusion, this thesis contributes a critical building block to the broader goal of enabling autonomous docking capabilities. While the system is not yet fully complete, the modular architecture, clear documentation and field-tested design serve as a robust platform upon which future development can confidently proceed.