POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Development and validation of a modular data acquisition architecture for autonomous driving research

Tesi di Laurea Magistrale in
Mechanical Engineering - Ingegneria Meccanica

Author: **Alessandro Dognini**

Student ID: 10607911
Advisor: Prof. Stefano Arrigoni
Co-advisors: Dr. Satyesh Shanker Awasthi
Academic Year: 2024-25

Abstract

An efficient data acquisition architecture is the foundation of every autonomous vehicle (AV). It is the system responsible for gathering all the measurements from all the sensors into a single unified software environment, where automated driving functions can use this data to control the vehicle. For this reason, behind the development of a new AV always stands the major challenge of making a wide variety of advanced sensors and powerful computers work together without errors. Traditional architectures used to chain software to a specific hardware, making it hard to test new ideas or use the same solutions on different vehicles. As cars become more like computers on wheels, there is a growing need for a flexible data acquisition system that is scalable and not tied to specific hardware. This thesis tackles this problem by creating a straightforward workflow for developing and validating a modular architecture for data acquisition. This kind of data acquisition architecture would drastically simplify setting up the vehicle to test any new prototype algorithm, because it is a system made by interchangeable and independent building blocks. In other words, each sensor can be added or removed easily, without influencing the correct functioning of the others. This solution uses Docker, to package the software stack of each sensor into an independent container, and ROS2 (Robot Operating System 2) as its middleware, to allow reliable communication between the containers. The workflow has been validated building a real data acquisition system that was then tested on an AV of the Mechanical Department of Politecnico di Milano. It is an electric car equipped with modern sensors, including a stereo camera, a 128-channel LiDAR and a dual antenna GNSS. The results prove that this container-based approach separates the sensor software stack from the underlying operating system. This simplifies adapting the validated data acquisition architecture to different AVs.

Keywords: autonomous driving; autonomous vehicles; data acquisition; ROS2; Docker.

Abstract in lingua italiana

Un'architettura efficiente di acquisizione dati è la base di ogni veicolo autonomo (AV). È il sistema responsabile di raccogliere tutte le misurazioni provenienti da tutti i sensori in un singolo ambiente software unificato, dove le funzioni di guida autonoma possono utilizzare questi dati per controllare il veicolo. Per questo motivo, dietro lo sviluppo di un nuovo AV si cela sempre la grande sfida di far funzionare insieme e senza errori un'ampia varietà di sensori avanzati e computer potenti. Le architetture tradizionali vincolavano il software ad un hardware specifico, rendendo difficile testare nuove idee o utilizzare le stesse soluzioni su veicoli diversi. Mentre le auto diventano sempre più simili a dei computer su ruote, cresce la necessità di un sistema di acquisizione dati flessibile che sia scalabile e non vincolato ad un hardware specifico. Questa tesi affronta tale problema sviluppando un flusso di lavoro lineare per la realizzazione e la validazione di un sistema di acquisizione dati modulare. Trattandosi di una struttura costituita da blocchi intercambiabili e indipendenti, questo tipo di sistema di acquisizione dati semplificherebbe drasticamente l'attività di configurazione del veicolo, che anticipa qualsiasi sperimentazione di nuovi prototipi di algoritmo. In altre parole, ogni sensore può essere aggiunto o rimosso facilmente, senza influenzare il corretto funzionamento degli altri. Questa soluzione utilizza Docker per impacchettare lo stack software di ciascun sensore in un container indipendente, e ROS2 (Robot Operating System 2) come middleware, per consentire una comunicazione affidabile tra i container. Il flusso di lavoro è stato validato realizzando un sistema di acquisizione dati reale, successivamente testato su un AV del Dipartimento di Meccanica del Politecnico di Milano. Si tratta di un'auto elettrica dotata di sensori moderni, tra cui una stereo camera, un LiDAR a 128 canali e un GNSS a doppia antenna. I risultati dimostrano che questo approccio basato sui container separa lo stack software di ciascun sensore dal sistema operativo sottostante. Ciò semplifica l'adattamento del sistema di acquisizione dati validato a AV diversi.

Parole chiave: guida autonoma; veicoli autonomi; acquisizione dati; ROS2; Docker.

Contents

Introduction

0.1. Motivation and Context

The automotive industry is experiencing a technological revolution centered on the development of AVs, which are expected to fundamentally reshape transportation and society. This transformation is driven by the promise of safer, more efficient, and more accessible mobility. With a projected global market value of $615 billion by 2026, the shift towards autonomy represents one of the most significant advancements in modern engineering [6].

The impact of AVs can be understood across four key domains:

- **Safety**: The most urgent motivation for AV development is the potential to drastically improve road safety. Over 90% of traffic accidents are attributed to human error, arising from factors like distraction, fatigue and poor judgment [6, 12]. By replacing human fallibility with tireless sensors and sophisticated algorithms, autonomous systems offer the potential to significantly reduce the millions of injuries and fatalities that occur on roads worldwide each year [12].
- **Society**: The societal implications of AVs are profound. They offer a new possibility of mobility and independence to underserved populations, including the elderly and individuals with disabilities who may be unable to drive [12]. This technology is also a key enabler for new car sharing services which could lead to a decline in private car ownership in favor of shared, on-demand autonomous fleets. Such a shift would not only reduce the number of vehicles on the road but could also free up urban areas currently dedicated to parking, allowing for the creation of more green and pedestrian spaces. [11, 12].
- **Environment**: The environmental impact of AVs is complex, with studies showing both potential benefits and significant risks [32]. On one hand, AVs can offer environmental gains through optimized energy consumption. Coordinated driving behaviors like "platooning" and "eco-driving" can smooth traffic flow and minimize energy use. On the other hand, these benefits could be reversed by changes in user behavior. As the time spent traveling becomes more productive or relaxing, the

perceived cost of commuting decreases. This could encourage people to live farther from their workplaces, leading to an increase in total kilometers traveled and promoting urban sprawl. Such a trend would not only increase emissions but also have negative consequences for land use and water resources [32]. Furthermore, a portion of the energy efficiency gained from optimized driving is counteracted by the significant power consumption of the onboard computational systems. The array of sensors, cameras, and powerful processors needed for real-time data processing represents a notable energy demand [6, 12].

- **Economics**: The transition to autonomous mobility is creating enormous business opportunities and transforming economic models. Beyond the direct market for AVs, the rise of shared mobility services could generate up to $1 trillion in revenues by 2030 [11]. This shift will revolutionize industries from logistics and delivery to public transport and personal taxi services, creating new economic growth [12]. Real-world examples are already emerging, with companies like Waymo, operating fully autonomous car services in some US cities [3].

At the center of this revolution is the car itself. It is becoming a powerful computer on wheels. A modern self-driving car is a complex system of hardware. It includes a variety of high-tech sensors like LiDARs, radars and cameras to see the world around it. It also has powerful onboard computers to process information and complex software to control the vehicle in real time [12].

The system that collects data from all these sensors is the foundation for all of the car's self-driving abilities. An autonomous driving system works in a few basic steps: first, it collects data from its sensors; second, it processes that data to understand its location and surroundings; third, it plans its next moves; and finally, it controls the car's steering, acceleration and braking [6]. The safety and reliability of the entire car depend on getting accurate data from the sensors. Modern sensors produce huge amounts of data, which can create bottlenecks and slow the system down. A poorly designed data collection system can lead to serious problems, such as lost data or delays. These failures can cause the car to make bad decisions, making it unsafe to operate. For this reason, building a strong and efficient data acquisition system is a critical challenge.

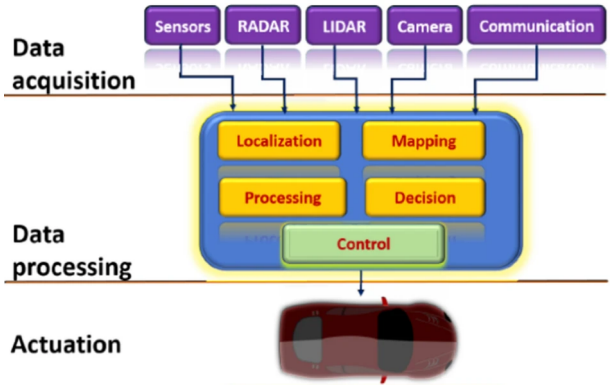


Figure 1: A representation of the main functional blocks of an AV [6].

The biggest obstacle to making reliable self-driving cars is managing their complexity. It is very difficult to get all the different hardware parts, often from different companies, to work together smoothly. Traditionally, this has led to systems where the software is tightly linked to specific hardware. These systems are fragile and hard to work with. A small update to a single sensor's software can require major changes throughout the system, making it difficult to maintain the car, upgrade it, or use the same software on a different vehicle. A modular data acquisition system offers a clear solution to these problems. By designing the system as a set of independent parts, it becomes much easier to update, replace or add new components without affecting the rest of the system. This approach makes the entire platform more flexible, easier to maintain, and adaptable to different vehicles and new technologies, which is crucial for accelerating research and development.

While various software platforms for autonomous driving exist, many research teams still spend a lot of time and effort on system integration for each new project. Integrating a new set of sensors or moving to a different vehicle platform often requires building a new data acquisition system from scratch. This repeated effort slows down the pace of innovation and creates a barrier for new research in the field. This thesis aims to address this challenge by designing, building and testing a complete solution that is modular, scalable and that can be easily adapted to any AV. The goal is to create a reusable framework that simplifies the integration process and allows researchers to focus more on higher-level software, developing and testing new autonomous driving capabilities.

0.2. Project Roadmap

The work for this thesis was structured into these main phases:

1. **Literature Review:** a comprehensive study of the state of the art in AV hardware,

software platforms, and data integration challenges. This initial phase established the technical context and identified the need for a modular and portable data acquisition solution.

2. **System Architecture Design:** definition of the core architectural principles for the data acquisition system to be developed. This involved selecting the foundational technologies. Docker was chosen for containerization and ROS2 as the communication middleware, being compatible for real-time applications [9].

3. **Workflow Development:** creation of a standardized open-source workflow for building modular data acquisition systems. This workflow was designed to be repeatable and adaptable for different vehicles and sensor configurations.

4. **Practical Implementation:** integration of the physical hardware onto the test vehicle, including the computational core, made of a NVIDIA Jetson and an Intel NUC, and the full sensor suite, formed by a ZED X stereo camera, an Ouster 128-channels LiDAR, a dual-antenna GNSS, and an IMU. This phase also included the development of all the specific software modules, packaging each sensor's drivers and ROS2 nodes into its own Docker container.

5. **System Testing and Validation:** a series of tests were conducted using the vehicle to empirically validate the architecture's performance against its design goals: modularity, portability, reliability, data integrity, real-time performance, and system resource utilization.

6. **Analysis and Documentation:** analysis of the collected data to draw conclusions on the workflow's effectiveness and on the performance of the implemented system. The final phase involved summarizing the project's contributions and documenting the system, including a user guide for future researchers of the Politecnico di Milano.

0.3. Thesis Structure

The thesis is organized as follows:

- **Chapter 1: State of the Art** provides a complete review of the state of the art in autonomous driving technology. It starts detailing the hardware foundation, including the sensor suite and computational platforms. It then analyzes the software ecosystem, covering key standards, open-source projects, and commercial solutions. The chapter connects these technologies to the escalating requirements of driving automation and concludes with an overview of current challenges and future trends.

- **Chapter 2: Software Design and Implementation** details the core technical work of this thesis. This chapter first introduces the foundational software technologies used, Docker and ROS2, explaining their relevance to the project. It then describes the actual open-source workflow that was developed specifically for this thesis. Finally, it presents the specific design and implementation of the real data acquisition architecture that was built to validate the workflow.
- **Chapter 3: Tests and Results** focuses on the practical validation of the real architecture. It starts by describing the test vehicle and its complete hardware and sensor configuration. The chapter then explains the methodology used for conducting the tests. It presents the results of these tests, focusing on performance metrics like data speed and reliability. This chapter provides the data to show that the system works well and is robust.
- **Chapter 4: Conclusions and Future Developments** summarizes the main results of the thesis and confirms that the project goals were met. It also points out the limitations of the current work and suggests ideas for future improvements.
- **Appendix: User Guide** contains a practical user manual that provides step-by-step instructions on how to operate and use the real developed system. This part is thought for all researchers that will use the system in the future.

1 State of the Art

1.1. The Data Integration Challenge

The development of AVs represents one of the most significant engineering challenges of the modern era. At the heart of this transformation is the vehicle itself, which has evolved from a primarily mechanical system into a powerful "computer on wheels" [16]. This evolution is characterized by an immense and growing complexity in integrating a diverse ecosystem of sensors and high-performance computational hardware into a single reliable system. A modern AV can generate between 20 and 40 terabytes of data per day, a huge amount of information that must be ingested and processed in real-time to ensure safe and effective operation [16].

The traditional approach to automotive design, where software is tightly coupled to specific hardware components, has proven inadequate for managing this complexity. Such systems make it difficult to update a single sensor's software without requiring substantial changes throughout the entire system. This outdated model creates a significant bottleneck, slowing the pace of innovation and obstructing the ability to incorporate new technologies. Consequently, the industry is undergoing a fundamental shift away from this model toward a "software-defined vehicle" (SDV) paradigm [15]. In an SDV, the sensor suite and the computational core serve as a generic foundation. In fact, the vehicle's capabilities are defined primarily through software. This paradigm demands a new approach to system architecture, one that prioritizes flexibility and decouples software functions from the underlying hardware.

This chapter provides a comprehensive review of the state of the art in AV hardware and software, establishing the technical context. The various approaches and technologies discussed will be evaluated against four key criteria that are essential for any modern research and development platform: modularity, scalability, portability, and reliability. Modularity refers to the degree to which a system is decomposed into independent, interchangeable components, each with a distinct responsibility. A modular architecture emphasizes the separation of functions into self-contained units that hide their internal

complexity behind well-defined interfaces, allowing them to be developed, tested, and maintained in isolation with minimal impact on other parts of the system. Scalability is the capacity to handle an increasing number of sensors, higher data throughput, and more demanding computational loads as technology evolves. Portability is the ease with which the software architecture can be deployed on different vehicle platforms and operating systems. Reliability is the ability of the system to perform its required functions under stated conditions for a specified period of time without failure.

To build this foundation, the following sections will first present the system requirements as defined by the standard levels of automation. Next, the chapter will detail the physical hardware layer of AVs, including the sensor suite and the computational core. It will then provide a complete analysis of the dominant software platforms, standards, and tools that constitute the modern AV software stack. Finally, the chapter will conclude by summarizing the main remaining challenges and future trends that drive the need for the kind of modular, portable, and scalable data acquisition architecture proposed in this thesis.

1.2. The SAE J3016 Standard: A Framework for Escalating Complexity

The Society of Automotive Engineers (SAE) J3016 standard provides a universally accepted taxonomy for classifying the different levels of driving automation [21]. This classification, which ranges from Level 0 to Level 5, is more than just a set of definitions. It establishes a clear framework for understanding the technical demands placed on the vehicle's data acquisition, processing, and control systems. Each step up the automation levels corresponds to a significant increase in system complexity and responsibility, directly impacting the requirements for the underlying hardware and software.

The six levels of automation are defined as follows:

- **Level 0 (No Automation):** the human driver is responsible for all aspects of driving at all times. The vehicle has no automated driving features [21].
- **Level 1 (Driver Assistance):** the system can assist the driver with either steering or acceleration/braking, but not both simultaneously. For example, adaptive cruise control or basic lane-keeping assistance are functions of this level of automation. The human driver must perform all other driving tasks and continuously monitor the environment [21].

- **Level 2 (Partial Automation):** the system can control both steering and acceleration/braking under specific conditions. This level includes many modern Advanced Driver-Assistance Systems (ADAS). However, the human driver must remain fully engaged, supervise the system, and be prepared to take immediate control at any moment [21].
- **Level 3 (Conditional Automation):** the system can perform all driving tasks under certain, well-defined conditions (e.g., a traffic jam pilot on a highway). The human driver is not required to supervise the system within this domain but must be available to take back control when the system issues a request [21].
- **Level 4 (High Automation):** the system can perform all driving tasks and monitor the driving environment without any human intervention, but only within a specific, predefined Operational Design Domain (ODD). An ODD might be a geofenced urban area, a specific highway route, or a campus environment. Waymo's robotaxi service is a prominent example of a Level 4 system [27]. If the vehicle travels outside its ODD, it must be able to achieve a safe state (e.g., pull over) without requiring a human to take over [21].
- **Level 5 (Full Automation):** this is the ultimate goal of autonomous driving. The system can perform all driving tasks under all roadway and environmental conditions that a human driver could manage. A Level 5 vehicle would have no geographical or conditional restrictions and would not require any human intervention [21]. Achieving this level of capability necessitates a huge improvement of current computing systems and sensor technologies.

The transition from driver assistance (Levels 1-2) to true autonomy (Levels 3-5) imposes exponentially increasing demands on the vehicle's data acquisition architecture. As the vehicle assumes full responsibility for monitoring the environment and making safety-critical decisions, the entire data pipeline must meet more stringent requirements.

Higher levels of automation necessitate a comprehensive, 360-degree, and redundant perception of the world, which requires a larger and more diverse suite of sensors to eliminate blind spots and ensure robust operation in a wide range of environmental conditions. This expanded sensor suite generates massive data streams. A single high-channel LiDAR can produce 10-70 MB/s of data, while a high-resolution camera can generate 20-40 MB/s, leading to a total data volume that can easily reach terabytes per day [16]. This immense data flow demands a high-bandwidth network to prevent data bottlenecks that could cripple the system. Furthermore, for an AV, safety is measured in milliseconds. For a vehicle traveling at 40 km/h, the entire perception-to-action control loop must be completed in

under 90 milliseconds to ensure it can react safely to events within a one-meter distance [16]. This imposes a hard real-time constraint on the system, demanding a real-time operating system and a high-performance middleware, that could guarantee low-latency data delivery [15]. Ultimately, as the system becomes the sole party responsible for safety, the standards for reliability become exceptionally high. Consequently, the entire data architecture must be developed in accordance with a rigorous functional safety framework, guided by key industry standards that govern both fail-safe behavior (ISO 26262) and the safety of the intended functionality in unforeseen scenarios (ISO 21448).

1.3. The Hardware Foundation: Sensors and Computational Core

The capabilities of any autonomous system are fundamentally defined and constrained by its physical hardware. This foundation is composed of two primary elements: the sensor suite, which serves as the vehicle's senses for perceiving the world, and the computational core, which acts as the brain, processing vast amounts of information to make intelligent decisions. This section reviews the state-of-the-art technologies in both of these critical domains.

1.3.1. The Sensor Suite: Perceiving the World

No single sensor can provide a complete, flawless, and reliable understanding of the complex and dynamic driving environment under all possible conditions. Each sensor technology has unique strengths and inherent weaknesses. For this reason, modern AVs employ a carefully selected suite of complementary sensors. By combining their outputs through a process known as sensor fusion, the system can create a world model that is more accurate, robust, and redundant than could be achieved with any single sensor alone [3, 22].

Cameras

Cameras are passive sensors that capture reflected light to produce two-dimensional images. They are the most common sensor on vehicles today. Their high-resolution output is particularly well-suited for modern deep learning algorithms, making them essential for tasks such as reading traffic signs and lane markings, detecting traffic light states, and classifying different types of objects (e.g., distinguishing an ambulance from a delivery truck). The primary strengths of cameras are their low cost and their ability to capture dense, semantically rich information. However, their performance is highly dependent on

environmental conditions. They are vulnerable to poor lighting at night and they can be blinded by direct sun glare [26]. Furthermore, their effectiveness is significantly degraded by adverse weather such as heavy rain, snow, or fog. As fundamentally 2D sensors, they cannot directly measure distance, which is a critical piece of information for safe navigation. Stereo cameras, however, overcome this limitation by mimicking human binocular vision. By using two cameras separated by a set distance (baseline), they capture two simultaneous images of the same scene from slightly different perspectives. The objects will consequently appear in different positions in the two images. A software is used to recognize the same objects in the two images and to compute the disparity, which is the difference in the horizontal coordinates of the same point as it appears in the left and right images. The disparity, the baseline, and the camera's focal length are then used to compute the precise distance of the objects through simple geometry [26].

LiDAR (Light Detection and Ranging)

LiDAR is an active sensor technology that operates by emitting pulses of laser light and precisely measuring the time it takes for these pulses to reflect off objects and return to the sensor. This time-of-flight measurement allows the system to calculate distance with very high accuracy. By rapidly scanning the environment, a LiDAR unit generates a dense, three-dimensional "point cloud" that represents the precise shape and location of surrounding objects [29]. The key strength of LiDAR is its ability to provide direct and highly accurate 3D geometric information, with centimeter-level precision, making it exceptionally effective for 3D object detection and creating detailed maps of the environment. Because it provides its own light source, its performance is consistent regardless of ambient lighting conditions, working equally well during the day and at night. The main technical weaknesses of LiDAR are its performance degradation in adverse weather like heavy fog, rain, or snow, which can scatter or absorb the laser pulses. Historically, LiDARs have always been a very expensive kind of sensor. Its cost has been a major barrier to its adoption in mass-market vehicles. However, recent technological developments are slowly lowering their price.

RADAR (Radio Detection and Ranging)

Like LiDAR, RADAR is an active sensor, but it uses radio waves instead of light. It emits radio signals and analyzes the properties of their reflections to determine the range, angle, and the relative velocity of objects. The standout advantage of RADAR is its exceptional resilience. Radio waves penetrate adverse weather conditions such as rain, snow, and fog with minimal attenuation, and they are unaffected by lighting conditions, making RADAR

an indispensable all-weather sensor [3]. Its ability to directly measure the velocity of other objects via the Doppler effect is also a significant advantage for tracking dynamic agents in the environment. The primary limitation of RADAR is its relatively low resolution compared to cameras and LiDAR. This makes it difficult to determine the precise shape of an object or classify it with high confidence, and it can sometimes struggle to reliably detect stationary objects.

Positioning Systems (GNSS & IMU)

Accurate localization is fundamental to autonomy. This is usually achieved by combining data from two complementary sensor systems: the Global Navigation Satellite System (GNSS) and the Inertial Measurement Unit (IMU). GNSS, which includes popular systems like GPS, provides an estimate of the absolute position on the Earth's surface by triangulating signals coming from a constellation of satellites. An IMU, on the other hand, uses internal accelerometers and gyroscopes to measure the vehicle's linear acceleration and angular velocity. By integrating these measurements over time, the IMU can track the vehicle's orientation and relative movement, a process known as dead reckoning.

Neither system is sufficient on its own. GNSS signals can become unavailable or inaccurate in urban canyons, tunnels, or under dense foliage, and their update rate is relatively low. The IMU provides very high-frequency updates and can continue to track the vehicle's position when GNSS is lost, but its small measurement errors accumulate over time, causing its position estimate to "drift". The standard solution is to tightly fuse the data from both sensors using a statistical estimation technique like a Kalman filter. The GNSS provides a periodic and absolute correction to the high-frequency relative updates from the IMU, resulting in a position estimate that is both continuous and accurate [3].

For the high levels of autonomy (Levels 3+), however, standard GNSS accuracy is insufficient. To achieve the required centimeter-level precision, the system is augmented with Real-Time Kinematic (RTK) corrections. RTK technology uses a stationary base station at a known location that receives the same satellite signals as the vehicle. By comparing its known position to the position calculated from the satellite signals, the base station can determine the exact error in the GNSS signal at that moment. It then broadcasts this correction data to the vehicle, which uses it to cancel out most of the errors in its own measurements, obtaining a very precise absolute position.

The Principle of Sensor Fusion

The necessity of sensor fusion arises directly from the complementary nature of the individual sensor modalities. No single sensor is sufficient because each has an operational blind spot. A camera can fail in the dark, LiDAR can be limited by fog, and RADAR may not see the shape of a stationary object. A robust autonomous system relies on redundancy, where the strengths of one sensor compensate for the weaknesses of another [3]. For example, a camera can semantically identify an object as a pedestrian, LiDAR can provide its exact 3D location and size, and RADAR can detect its presence even in heavy rain. This combined understanding is far more reliable and complete.

There are two primary strategies for combining this data, which differ mainly in the stage at which the fusion occurs: early fusion and late fusion [25].

- **Early Fusion (Low-Level Fusion)**: in this approach, raw data from different sensors is combined at the very beginning of the perception pipeline, before any significant processing has occurred. For example, LiDAR point clouds might be projected onto a camera image before being fed into a single neural network. This allows the perception model to learn complex, low-level correlations between the different data types, potentially improving accuracy. However, this method is computationally intensive, less modular, and requires precise sensor calibration, as a failure or delay in one sensor stream can disrupt the entire fusion process [25].
- **Late Fusion (High-Level Fusion)**: In this strategy, each sensor stream is processed independently by its own dedicated perception algorithms to generate a list of detected objects. These high-level object lists are then fused at the end of the pipeline to produce a final, unified environmental model. This approach is more modular and robust to the failure of a single sensor. However, by processing the data in isolation, it may lose out on valuable low-level correlations that could have improved detection accuracy, and information from detections with low confidence may be discarded prematurely [25].

A compromise between these two main strategies is Mid-Level Fusion. Instead of fusing raw data or final object lists, this approach fuses features that have been extracted from each sensor's raw data. For instance, it might combine color information from a camera with location features from a LiDAR point cloud. This method generates a smaller information space and requires less computational load than early fusion while retaining more detailed information than late fusion, offering a balance between the two extremes [25].

1.3.2. The Computational Core

The evolution of AV's computational core represents a fundamental change in automotive engineering. This transformation is not a matter of choice but a necessary response to the immense data processing and real-time decision-making demands imposed by higher levels of driving automation. Current automotive systems are characterized by a standard domain-based Electrical/Electronic (E/E) architecture [15]. This system uses dozens of independent Electronic Control Units (ECUs). Each ECU is a simple microcontroller dedicated to a specific function, such as engine valve timing or anti-lock braking. The communication between all the ECUs is generally performed through a low-bandwidth Controller Area Network (CAN) bus, ensuring predictable behavior for simple control tasks. This architecture is fundamentally unsuited for the demands of high-level autonomy for several reasons.

The first reason is that the distributed low-power processors lack the centralized computational capacity required for complex tasks like sensor fusion and other fundamental operations for autonomous driving. Furthermore, the huge volume of data coming from a modern sensor suite would overwhelm the low-bandwidth CAN bus that connects these ECUs. Wiring complexity is also a problem. Interconnecting dozens of ECUs spread across the vehicle requires a vast and intricate wiring harness. This complexity not only adds significant weight and cost but also makes maintenance and troubleshooting exceptionally difficult. Finally, the software on traditional ECUs is tightly coupled to the specific microcontroller hardware. Updating functionality often requires reflashing the entire unit, and the limited processing and memory resources make it nearly impossible to add significant new features. This lack of flexibility of a traditional E/E architecture doesn't reflect the modern concept of software-defined vehicles, where capabilities are expected to evolve through over-the-air updates [15].

To overcome these limitations, the automotive industry is shifting towards a centralized E/E architecture [15]. In this modern model, the computational workload is consolidated into a small number of powerful computers. These powerful nodes are interconnected by a high-bandwidth automotive Ethernet backbone, capable of handling the massive data throughput from sensors. In this new design, ECUs are no longer the brains of the operation. They simply collect raw data and pass it up to the central computer, or take simple commands from the central computer and execute them with an actuator.

An autonomous car has to do many different kinds of computing jobs. Some tasks, like recognizing objects from sensor data, require doing thousands of calculations in parallel, while others, like making driving decisions, are determined in a logical step-by-step order.

Because these jobs are so different, a single type of processor cannot handle all of them efficiently. For this reason, a modern AV computational core combines different types of processors, with each one specialized for a certain kind of job [16, 30].

The main parts of this computer brain are:

- **Central Processing Unit (CPU):** the CPU is the main manager of the system. It is very good at handling tasks one by one, making logical decisions, running the main operating system, and organizing the overall workflow. In a self-driving car, the CPU schedules all the high-level tasks and tells the other more specialized processors what to do [30].
- **Graphics Processing Unit (GPU):** GPUs are the processors for doing many calculations at the same time. They are built with thousands of smaller cores, which makes them perfect for the deep learning and machine learning math that is used to see and understand the world. Tasks like detecting objects, recognizing images, and combining data from different sensors depend heavily on the power of GPUs [30].
- **Application-Specific Integrated Circuits (ASICs):** These are custom-designed chips built to do only one thing, but they do it extremely well and use very little power.
- **Field-Programmable Gate Arrays (FPGAs):** FPGAs are a flexible middle ground. They are not as general as a GPU, but not as rigid as an ASIC. Their internal logic can be reconfigured after they are manufactured, which makes them great for tasks like preparing high-speed sensor data before it goes to the main processor [16].

The current trend is to put all of these different parts (CPUs, a GPU, accelerators, and other necessary components) onto a single piece of silicon. This is called a System-on-Chip (SoC). Integrating everything onto one chip is very important because it saves space, reduces power use, and allows the different parts to communicate with each other much faster. All of these are critical in a car, where space and power are limited [16].

1.3.3. Hardware Architecture in Practice: Real-World Configurations

When designing the hardware for a self-driving car, engineers face a difficult trade-off between three things: performance, power, and cost. To make a car more autonomous, it needs more performance. But higher performance hardware, especially powerful LiDARs

and GPUs, uses a lot more electricity. For an electric car, this is a big problem because it reduces the driving range. It also creates a lot of heat that needs to be managed with cooling systems. At the same time, these high-performance parts are very expensive, which increases the final price of the car. In this section are presented two real hardware configurations, in order to understand how all these hardware components could be combined on a real car. The described vehicles are: the EasyMile EZ10 Gen1, and the 5th-generation Waymo Driver.

The EasyMile EZ10 Gen1 (2018)

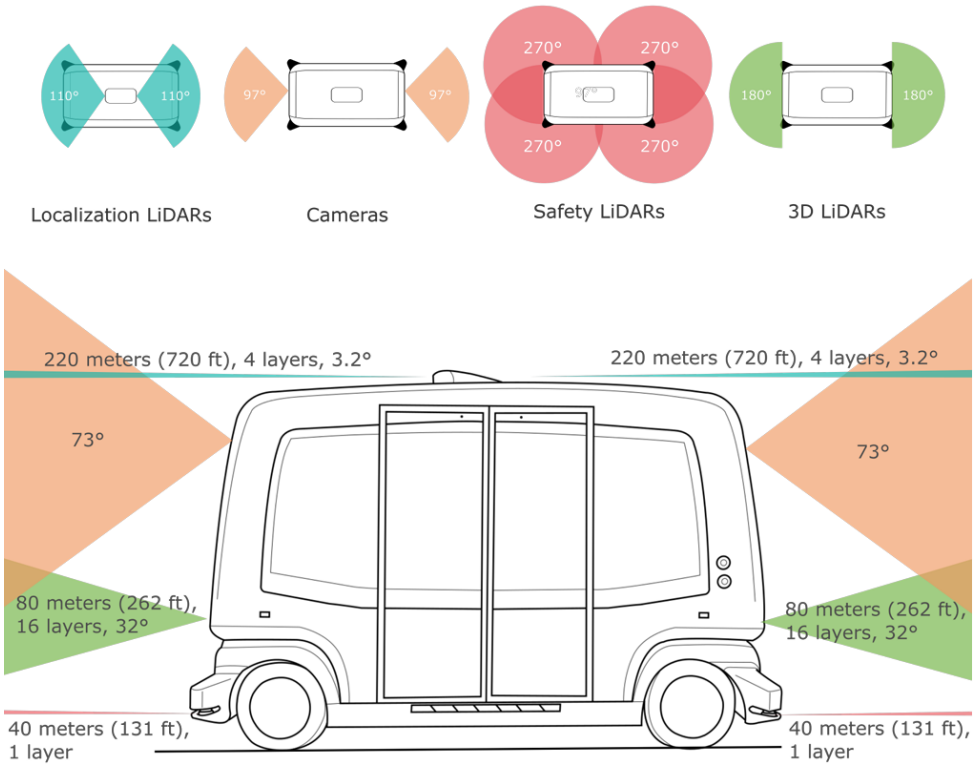


Figure 1.1: EasyMile EZ10 Gen1 sensor suite [8]

The EasyMile EZ10 Gen1 shuttle is an example of a Level 4 AV designed for a constrained ODD. As shown in Figure 1.1, the EZ10 is a low-speed mobility platform intended for use on predefined geofenced routes, such as those found on corporate campuses and airports [8]. Its hardware architecture is a practical reflection of this limited operational domain, prioritizing safety, reliability, and cost-effectiveness over the extreme performance required for unconstrained public road navigation.

A key design feature of the EZ10 is its symmetrical body, which allows for fully bidirec-

tional operation. This eliminates the need for complex turning maneuvers like U-turns, simplifying both the path-planning software [8]. The sensor and computational architecture is not a single integrated system like Waymo's, but is instead a hierarchical, double-level system that strategically separates safety-critical functions from complex navigation tasks.

The EZ10's sensor suite is best understood by the specific role each sensor plays within this hierarchical system:

- **Sensors for Localization:** the shuttle's primary method for determining its position is LiDAR-based SLAM. Two 2D LiDARs are mounted on the roof, continuously scanning the surrounding environment. The system compares these scans in real-time to a prerecorded map of the route to achieve precise localization. This is supplemented by data from an IMU and wheel encoders, which are used for dead reckoning to track the vehicle's position between LiDAR updates [8].
- **Sensors for Obstacle Detection:** the EZ10 employs a layered approach to obstacle detection. The foundation of its safety system is a "Safety Chain" built around four single-layer 2D LiDARs. These are mounted at the corners of the vehicle, approximately 30 cm above the ground, and each provides a 270-degree field of view, combining to create a complete 360-degree safety perimeter. The purpose of these sensors is not sophisticated object classification but simple presence detection. If any object enters in this predefined safety zone, these LiDARs send a direct signal to the emergency braking system. For more advanced and smoother navigation, the vehicle is equipped with two 16-layer 3D LiDARs, one at the front and one at the rear. These provide a volumetric point cloud to the high-level computer, allowing it to detect obstacles further down the path and plan avoidance maneuvers, such as slowing down, rather than initiating an emergency stop. The vehicle also has cameras, but their use for guidance is described as experimental in the Gen1 model [8].

The most distinctive feature of the EZ10's architecture is its dual-level computational core, which separates the system into a "Low Level" and a "High Level" [8].

- **The Low Level:** this system is based on an industrial-grade Programmable Logic Controller (PLC), a type of computer known for its reliability and predictability in safety-critical industrial automation. The PLC runs the "Safety Chain", directly connecting the four corner safety LiDARs to the vehicle's emergency braking actuators. This system is deliberately simple, robust, and operates completely independently of the PC-based navigation computer. Its sole function is to stop the

vehicle if an imminent collision is detected [8].

- **The High Level:** this system consists of an industrial PC that runs EasyMile's proprietary software for localization, path planning, and advanced obstacle avoidance. It processes the data from the localization LiDARs and the 3D LiDARs to perform the driving functions of the vehicle, such as following the route and avoiding obstacles [8].

This separation of safety and navigation is a key architectural decision. By implementing the emergency stop function on a PLC, EasyMile creates a robust safety block that can be easily certified. This allows the navigation software on the high-level PC to be developed and updated without requiring a full revalidation of the vehicle's core safety system each time. This modular approach represents a pragmatic solution for deploying autonomous technology safely and economically.

The 5th-generation Waymo Driver (2020)

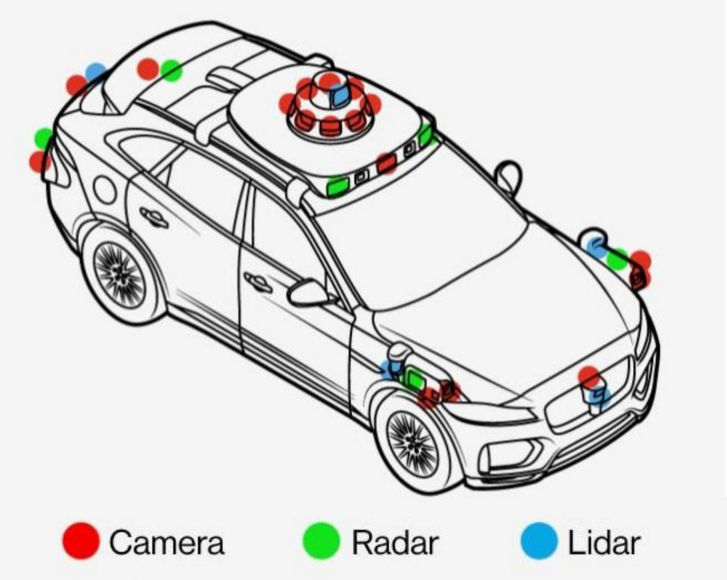


Figure 1.2: 5th-generation Waymo Driver sensor suite [23]

The 5th-generation Waymo Driver, traditionally integrated onto a Jaguar I-PACE, represents a state-of-the-art implementation of a Level 4 autonomous system designed for public roads within a pre-mapped ODD, like the cities of Los Angeles and San Francisco [13]. Its hardware architecture, as illustrated in Figure 1.2, employs 29 cameras, 6 radars, and 5 LiDARs. It is the result of a vertically integrated design philosophy where Waymo develops its sensors, computational hardware, and software in-house. This approach al-

lows for deep optimization across the entire system, creating a more capable system than one assembled from off-the-shelf components.

The foundation of the Waymo Driver's capability is its multi-modal sensor suite, which is engineered to provide a comprehensive, 360-degree, and highly redundant perception of the car's surroundings. This is achieved by combining all the main three complementary sensor technologies, ensuring that the limitations of one are compensated for by the strengths of the others.

- **LiDAR System:** the LiDAR system provides the primary, high-precision 3D geometric data. A central, roof-mounted 360-degree LiDAR delivers a long-range view of the environment, capable of detecting objects more than 300 meters away. This long-range capability is critical for safe operation at highway speeds, allowing the system enough time to react to distant obstacles. This primary sensor is supplemented by four perimeter LiDARs located at the corners of the vehicle. These units are optimized for close-range sensing, offering a wide field of view to detect nearby objects. This is essential for navigating tight urban traffic and eliminating potential blind spots on complex terrain. Together, this family of LiDARs generates a seamless and detailed 3D point cloud of the vehicle's surroundings under all lighting conditions [13].
- **Vision System:** while LiDAR provides the geometric structure of the world, the vision system provides rich semantic context. The vehicle is equipped with a dense network of high-resolution cameras that provide a complete 360-degree field of view. A suite of long-range cameras can identify critical details, such as pedestrians and the text on road signs, from over 500 meters away. This allows the system to understand and obey traffic laws. In addition, a peripheral vision system works together with the perimeter LiDARs to reduce blind spots. To ensure reliable performance in all conditions, the camera housings incorporate integrated cleaning and heating mechanisms to clear away dirt, rain, or snow [13].
- **Radar System:** the third layer of perception is provided by a custom-developed imaging radar system. Unlike traditional automotive radar, Waymo's imaging radar provides a higher resolution, 360-degree view that can better detect and track both moving and stationary objects. The primary advantage of radar is its robustness in adverse weather conditions like heavy rain, fog, and snow, where the performance of both LiDAR and cameras can be significantly degraded. It provides an all-weather capability to instantaneously measure an object's velocity, complementing the other sensors to create a truly resilient perception system [13].

This integrated suite is controlled by a powerful onboard computational core, which stands under the floor of the Jaguar I-Pace's trunk. The exact technical specifications of the computational core are not publicly disclosed, but some key aspects are available. It combines CPUs and GPUs to process the immense volume of sensor data in real-time. Beyond raw processing power, the entire vehicle architecture is built on a foundation of safety-critical redundancy. This includes a full secondary compute system that can take over in case of the primary one fails. There are also redundant steering, braking, and power systems. This ensures that there is no single point of failure for any function essential to the safe operation of the vehicle.

A critical aspect of the Waymo Driver's capability is its sophisticated approach to localization. Rather than relying just on GNSS, which can be unreliable in urban canyons or tunnels, Waymo's primary strategy is based on matching real-time sensor data to highly detailed prebuilt 3D maps. Before a vehicle begins operating in a new area, Waymo first maps the territory with immense detail, recording the location of static environmental features such as curbs, lane markers, and traffic signs. The vehicle then determines its exact position with centimeter-level accuracy by constantly comparing the 3D point clouds from its LiDARs and other sensor data against this map. IMU is then used for dead reckoning between updates, ensuring a continuous understanding of its location at all times.

1.4. Analysis of the Software Stack

The development of a reliable AV is fundamentally a software challenge. While the physical sensors and computers form the vehicle's body, the software stack acts as its central nervous system. It is responsible for perception, decision-making, and control. The choice of a software platform is one of the most critical decisions in AV development, as it defines the system's architecture, flexibility, and scalability. The landscape of available platforms is vast, reflecting different philosophies on how to best tackle the immense complexity of autonomous driving. These approaches range from collaborative open-source frameworks to tightly integrated commercial ecosystems and fully closed proprietary systems built by industry giants.

A significant factor driving the evolution of these software platforms is the radical transformation of the underlying vehicle hardware architecture. As analyzed in section 1.3, the traditional architecture, formed by a distributed network of specialized ECUs, proved unsuitable for the demands of autonomous driving. This led to a paradigm shift towards the SDV concept [15]. In this model, the vehicle's functions are no longer defined by its hardware but by its software, which runs on a centralized and high-performance com-

puting platform connected by a high-bandwidth automotive Ethernet backbone. This approach transforms the vehicle into an upgradable platform that can receive new features and improvements via over-the-air updates. This shift represents a new philosophy where software is the main driver of value and innovation in the automotive world.

To manage the complexity of this new paradigm, the industry has gathered in support of standardization. The most remarkable is the AUTOSAR (AUTomotive Open System ARchitecture) consortium [1]. While the AUTOSAR Classic Platform was designed for the traditional approach of deeply embedded ECUs, the AUTOSAR Adaptive Platform was created specifically for the high-performance computing needs of SDVs and autonomous driving. It provides a standardized and service-oriented architecture. This standard enables the dynamic deployment of software components, manages communication over high-speed networks, and provides a common framework for safety and security. The AUTOSAR Adaptive Platform serves as the industry's foundational layer, creating a standardized environment in which the diverse software platforms can operate. This section will explore the major software platforms and tools used nowadays, evaluating them based on their architectural philosophy, technical capabilities, and the trade-offs they present for research and development. For simplicity, these platforms are divided into three kinds of general approaches: open-source platforms, commercial platforms, and proprietary systems. These three very different strategies present their own strengths and weaknesses in the development of the software stack of AVs. Some of those strengths and weaknesses are presented in Figure 1.3

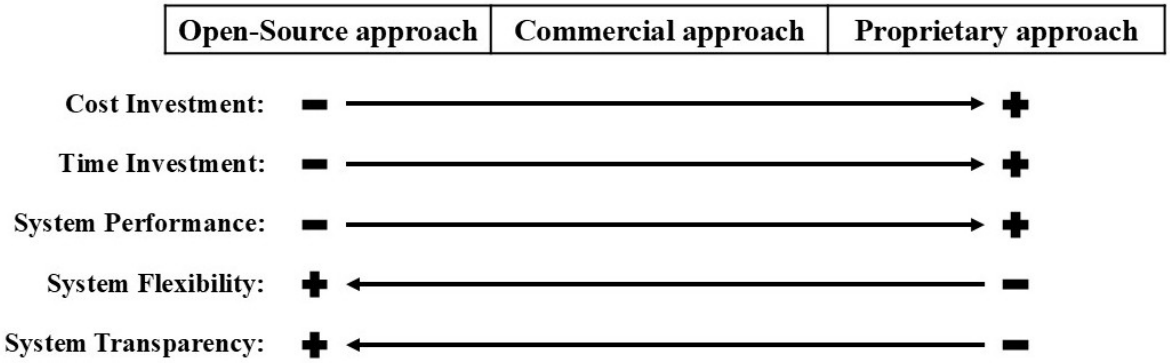


Figure 1.3: Comparison between the 3 approaches used to develop the software stack of an AV

1.4.1. Open-Source Platforms

Open-source platforms have been very useful in accelerating the pace of autonomous driving research by democratizing access to advanced technology. By making their source code publicly available, these platforms promote a global community of developers and researchers who collaborate to build, test, and refine the software stack. This collaborative model lowers the barrier to entry for academic institutions and smaller companies, enabling them to contribute to the field without the prohibitive cost of developing a full system from scratch [2]. However, "open-source" is not a monolithic category. It includes a diverse range of approaches, from full-stack solutions to foundational framework approaches that provide the building blocks for custom systems.

Full-Stack Platforms

Full-stack platforms aim to provide a complete, end-to-end software solution for autonomous driving, covering everything from sensor drivers to planning and control algorithms. The two most notable examples are Autoware and Baidu Apollo.

Autoware, first released in 2015, is recognized as the world's first open-source software for autonomous driving and is now managed by the Autoware Foundation [10]. Its architecture is built upon ROS2, which makes it inherently modular and flexible. This modularity allows researchers to easily experiment with the system by replacing individual components without altering the rest of the stack. For example, swapping a specific LiDAR-based object detection algorithm for an experimental one. For communication between its modules, Autoware relies on the Data Distribution Service (DDS) standard, which is the underlying middleware of ROS2. DDS is an open standard for reliable communication used in mission-critical systems such as military and aerospace applications [9]. The main advantages of Autoware are its flexibility, its strong alignment with the academic and research communities, and its compliance with open standards. However, its high degree of modularity can also present a steep learning curve, and the process of integrating and tuning all the necessary components for a specific vehicle can be complex. Furthermore, while DDS is excellent for reliability, its process of serializing data before transmission can introduce latency and become a bottleneck when dealing with the extremely high data rates of multiple high-resolution sensors [14].

Baidu Apollo, funded by the Chinese technology giant Baidu, offers an integrated all-in-one platform [4]. It provides a vast ecosystem that includes not only the core driving software but also a dedicated simulation platform, cloud services for data management and model training, and HD mapping tools. The most significant architectural difference

between Apollo and Autoware lies in their middleware. Instead of using the open DDS standard, Apollo employs a proprietary middleware called CyberRT. The core innovation of CyberRT is its use of a shared memory method for communication between processes running on the same machine. This approach avoids the computational overhead of serializing data into packets and copying it between processes, which is required by DDS [14]. The result is significantly lower latency and higher data throughput, which is a major advantage when processing data from high-bandwidth sensors like LiDARs. The primary strengths of Apollo are its maturity, its rich feature set, and its focus on commercial implementation, which can allow development teams to get a system running more quickly. However, this high level of integration comes at the cost of modularity. It can be more difficult to customize or replace individual components within the Apollo stack compared to Autoware. Its reliance on a proprietary middleware also makes it potentially harder to integrate with external tools that are built around open standards like ROS2 and DDS [14].

The Foundational Framework Approach

An alternative to adopting a complete full-stack platform is to use a foundational framework as a base upon which a custom system can be built. This approach, which is the one used in this thesis, offers maximum flexibility and control over the system's architecture. **ROS2** is the leading framework for this purpose.

While the original version of ROS was widely used in robotics research, it was not designed to meet the strict real-time and reliability requirements of safety-critical applications like autonomous driving. ROS2 was redesigned from the ground up to address these limitations [9]. Its suitability for AVs emerges from several key design choices. First, ROS2 is built on top of DDS, a robust middleware that provides configurable quality of service policies to guarantee message delivery and manage real-time data streams. This communication is based on an anonymous publish-subscribe model [9]. In this model, a software component, called a node, can publish data as a message to a specific channel, known as a topic. Any other node in the system can then subscribe to that topic to receive the messages asynchronously. This communication architecture is highly advantageous because it decouples the software components. Second, ROS2 is designed to support real-time programming practices. By using a real-time Linux kernel and adhering to rules like static memory allocation, which avoids the unpredictable delays caused by allocating memory during runtime, a ROS2-based system can achieve the deterministic low-latency performance required for an AV's control loop, from perception to control [24]. Finally, the distributed, node-based architecture of ROS2 naturally lends itself to creating modular

systems. Encapsulating each sensor driver or software algorithm in a separate node, as was done in this thesis, creates a system that is portable and easier to debug [24]. This foundational approach requires more initial integration effort but provides unparalleled freedom to design an architecture tailored to specific research goals.

Open-Source Advanced Driver-Assistance System

To provide a complete view of the open-source landscape, it is also important to consider specialized systems like **OpenPilot** [7]. Unlike Autoware and Apollo, OpenPilot is not a full-stack platform aiming for Level 4 or 5 autonomy. Instead, it is an open-source ADAS designed to provide Level 2 capabilities, such as lane-keeping assistance and adaptive cruise control. It is mainly vision-based and is intended to improve the functionality of existing consumer cars using affordable off-the-shelf hardware. OpenPilot represents a philosophy focused on improving the cars of today, rather than building the fully AVs of tomorrow. Its existence shows the diversity within the open-source community and its ability to address different segments of the autonomous driving challenge.

1.4.2. Commercial Platforms

In contrast to the democratic nature of open-source projects, commercial platforms are developed by companies that provide a tightly integrated package of hardware and software. While these platforms offer powerful and high-performance systems, this approach often sacrifices flexibility, locking customers into a single company's products.

NVIDIA DRIVE is a dominant player in this field. It operates using a business-to-business (B2B) model that provides an end-to-end platform for automotive companies [20]. NVIDIA's core strategy is to offer a unified and scalable ecosystem that covers the entire AV development pipeline, from AI model training in the data center to the real application on the vehicle. This ecosystem is built on three interconnected computing solutions: NVIDIA DGX for data center training, NVIDIA Omniverse and NVIDIA Cosmos for physically accurate simulation, and NVIDIA DRIVE AGX for in-vehicle computation. This integrated approach is a key strength, as it provides a complete workflow for developing, testing, and deploying autonomous cars. The hardware foundation of the platform is the DRIVE AGX family of powerful and scalable System-on-Chip computers that can deliver over 250 trillion operations per second [20]. This immense computational power is optimized for parallel processing via NVIDIA's CUDA architecture, which enables it to run the complex deep learning models that are central to NVIDIA's approach for autonomous driving. The hardware is powered by a comprehensive software stack.

The foundation is DRIVE OS, a safety-certified operating system that complies with ISO 26262 standards [20]. The DRIVE AV stack contains the software for perception, planning, and control, including pre-trained deep neural networks. This full-stack software architecture allows partners to accelerate development while ensuring compliance with the stringent automotive standards. NVIDIA's B2B approach has established a vast ecosystem of over 300 partners. Among them, there are a lot of major car manufacturers who are building their next-generation vehicles using the NVIDIA DRIVE platform, such as Mercedes-Benz, Jaguar Land Rover, Volvo Cars, and Toyota. This business model has made NVIDIA a key supplier for the software-defined vehicle era. However, this integrated ecosystem comes with big trade-offs. The deep coupling of software to NVIDIA's hardware creates strong vendor lock-in. Moreover, the high cost of the specialized hardware makes it inaccessible for academic research and smaller-scale projects.

Mobileye Drive, from the Intel-owned company Mobileye, represents another B2B solution for autonomous driving [18]. From being an established leader in vision-based ADAS, Mobileye has evolved its technology into a full-stack solution for AV. The core of their philosophy is redundancy. Two entirely separate and independent perception systems are used within the vehicle. The first is a full 360-degree system that relies only on cameras. The second is a parallel system that uses radars and LiDARs. The data from these two systems is not fused at the sensor level. Instead, each one generates an independent world model, and their outputs are only combined at the final decision-making stage. This ensures that the vehicle has a complete backup perception system if one modality fails. The entire software stack is designed to run on Mobileye's proprietary EyeQ System-on-Chip [18]. It features a heterogeneous computing architecture, combining general-purpose CPUs with a suite of specialized accelerators to deliver high performance for AI and computer vision tasks. This architecture is inherently scalable. The same foundational technology powers solutions ranging from basic ADAS, using a single EyeQ chip, to the fully autonomous Mobileye Drive platform, which uses four EyeQ6 High chips. This modularity allows automakers to integrate the system into various vehicle types and scale their cars from ADAS to full autonomy over time. Mobileye's approach has attracted numerous major automakers, including Volkswagen Group (for its Audi, Lamborghini, Porsche, and Bentley brands). Mobileye delivers a complete solution but offers very little flexibility for developers to modify its algorithms or integrate their own components, making it unsuitable for research.

1.4.3. Proprietary Systems

The proprietary approach represents the most closed and expensive approach. It is defined by a strategy of vertical integration, in which a single company designs the entire autonomous driving stack, from the physical sensors and the vehicle platform to the highest levels of decision-making software. This approach is pursued with the goal of achieving the highest possible level of optimization, as every component can be tailored to work perfectly together. However, this comes at the cost of important financial investment and creates a closed system that is inaccessible to the broader research community.

Waymo Driver, which originated as the Google Self-Driving Car Project, is the definitive example of this model [27]. Waymo's commercial focus is its SAE Level 4 autonomous ride-hailing service, Waymo One. The service is currently operational in several major US cities. Waymo has invested over a decade in designing its complete technology stack from the ground up. The strategy is to develop a single adaptable autonomous system that can be integrated into multiple vehicle platforms.The latest currently active model is the 5th-generation Waymo Driver, and its hardware is analyzed in section 1.3.3. This advanced hardware is tightly coupled with a proprietary software stack that has been continuously refined over tens of millions of miles on public roads and billions of miles in simulation [27]. These numbers highlight the maturity of Waymo's technology. The software architecture is increasingly leveraging large-scale AI. This AI-centric system is responsible for the entire decision-making process: it interprets the complex data from the sensor suite to perceive the world, predicts the behavior of all surrounding road users, and plans the vehicle's trajectory and actions in real-time.

Zoox, company owned by Amazon, represents an even more extreme form of vertical integration [31]. Zoox's strategy is to build a complete transportation solution from scratch, including a purpose-built entire car. Their vehicle is a revolutionary change from traditional automotive design. It is a bidirectional robotaxi with no steering wheel or pedals, designed exclusively for passengers. A design approach very similar to the EasyMile EZ10, that is analyzed in section 1.3.3. The hardware is engineered for complete redundancy across all critical systems. This includes dual steering and braking systems, two independent power systems with each its own battery, and an overlapping 360-degree sensor disposition that eliminates any blind spots. This sensor suite includes cameras, LiDARs, radars, microphones, and thermal cameras. All the data coming from the sensors is processed by multiple NVIDIA GPUs that run Zoox's proprietary software stack [31]. While the vehicle is designed with the capabilities for Level 5 autonomy, it is operated as a Level 4 service within constrained ODDs in some US cities. The strategy followed

by Zoox allows a passenger experience difficult to achieve by retrofitting existing vehicles. However, this strategy is also incredibly complex and expensive.

1.5. Current Challenges and Future Outlook

The progress in sensor technology and computational hardware, that was detailed in the previous sections, has introduced a new class of engineering challenges. While focus usually remains on high-level autonomy solutions, the foundational data acquisition layer faces obstacles that must be overcome. The transition from controlled test environments to the unpredictable reality of public roads is not just a software problem, but a system integration problem.

This section summarizes the primary challenges in data acquisition that directly motivate the need for the modular solution proposed in this thesis. It then explores the future outlook for these architectures, which is trending towards the same principles of modularity and scalability.

1.5.1. Current Challenges

The main challenges towards fully AVs stand in many different areas, from edge cases and regulation to cybersecurity. However, for researchers the more immediate challenges are found within the vehicle's own data pipeline.

- **Data Volume and Bandwidth Bottlenecks:** modern sensor suites, especially high-resolution LiDAR and stereo cameras, generate a large stream of data, usually exceeding hundreds of Mbps. Automotive networks like CAN bus are inadequate for this. While automotive Ethernet provides the physical bandwidth, the software architecture must be efficient enough to process this data in real-time without introducing too much latency or bottlenecks.
- **System Integration:** this is an important barrier to rapid research and development. Sensor software stacks (drivers, SDKs, and dependencies) are usually installed directly on the host operating system. This creates a fragile system where a single update for one sensor can create conflicts that break another. This tight coupling makes the system difficult to maintain. Furthermore, transferring the same system and driving algorithms to a different vehicle becomes complicated and time-consuming.
- **Reliability and Non-Deterministic Performance:** an autonomous control loop, from perception to actuation, depends on a predictable flow of data. A data acqui-

sition system that drops messages or delivers them with high-jitter can compromise safety. A good architecture must guarantee the quality of service for all data streams, ensuring that safety-critical messages are delivered reliably, while high-bandwidth data is delivered with the lowest possible latency.

1.5.2. Future Outlook

In response to these challenges, the industry is exploring some promising emerging technologies. These future trends will not only address current limitations but will also unlock new capabilities.

- **Emerging Sensors and Computing Technologies:** the sensor suite of future AVs is likely to evolve. Emerging sensors like thermal cameras can offer robust perception in low-light conditions, while event cameras can capture high-speed motion without blur, complementing the data from traditional cameras and LiDAR [17]. On the hardware side, the industry is moving from CPUs and GPUs toward highly specialized Systems-on-Chip that integrate accelerators for AI. Looking further ahead, new computing paradigms like in-memory processing, which integrates computation directly into memory chips, promise to dramatically reduce the energy and latency costs of data movement.
- **The Software-Defined Vehicle (SDV):** as already anticipated, this is a dominant trend in AVs. This approach, which is a core theme of this thesis, is essential for enabling updates, adding new features, and managing system complexity. Containerization, as explored in this work, can be a key enabling technology for the SDV.
- **V2X Communication and Cooperative Perception:** Vehicle-to-Everything (V2X) communication is a key enabling technology for the future. By allowing vehicles to communicate with each other (V2V) and with the surrounding infrastructure (V2I), a vehicle's perception is no longer limited by its own sensors. It can receive warnings about a stopped car around a blind corner or know that a traffic light is about to change before it is even visible. This "cooperative perception" creates a more complete understanding of the driving environment, significantly enhancing safety and traffic efficiency.

Addressing the current challenges and exploiting the potential of these future technologies will depend critically on the underlying software architecture. A rigid system cannot easily integrate new sensors or adapt to the dynamic world of V2X communication. This reality highlights the urgent need for a modular, scalable, and portable framework. This thesis

tackles exactly this issue. It creates a solid foundation for the next generation of research and development in autonomous driving.

2 | Software Design and Implementation

The previous chapter presented the data integration challenge at the heart of modern AV development. It detailed the complexity of sensor suites and computational hardware, concluding that traditional rigid system architectures are no longer suitable for managing the requirements of high-level autonomy in terms of scale, dynamism, and reliability. In the context of autonomous driving research, such traditional designs, where software is intrinsically linked to specific hardware, create significant bottlenecks. In particular, they impede innovation, complicate maintenance, and prevent the reuse of valuable engineering solutions across different vehicle platforms.

This chapter transitions from the problem's definition to its solution, detailing the design and implementation of the modular data acquisition architecture developed for this thesis. The core architectural philosophy guiding this work is the strategic decoupling of software functionality from the underlying physical hardware. This approach is a necessary paradigm shift that aligns with the broader industry trend towards the SDV. Achieving this requires an architecture that prioritizes modularity, portability, and reliability. The strategy that was used in order to realize this vision is made of two main parts. The first one is component encapsulation, which involves isolating individual software components into independent and portable units. The second element is the adoption of a standardized communication middleware. This middleware should provide a robust communication network for managing the real-time data exchange between the isolated software components.

This architectural philosophy is implemented through two foundational open-source technologies. Docker was chosen for component encapsulation, because its widely proven and recognized containerization capabilities. ROS2 was selected as the middleware, for the standardized real-time communication. This chapter is organized in three parts to fully detail this approach. The first section provides a rigorous justification for these software choices, describing them as the fundamental engineering tools that enable the entire ar-

chitectural vision of this thesis. Building upon this foundation, the chapter will then present the central contribution of this thesis: a step-by-step open-source workflow for creating modular data acquisition systems. Finally, the chapter will conclude by detailing the practical implementation of the real data acquisition architecture, from the specific hardware selected to the overall software structure deployed on the test vehicle.

2.1. Foundational Software: Docker & ROS2

The selection of a software platform is one of the most critical decisions in AV development, as it defines the entire system's properties. The foundational layer of the architecture proposed in this thesis is built upon the combination of Docker and ROS2. This pairing is not arbitrary. It is a strategic choice in which each technology addresses a distinct but complementary aspect of the system design challenge. Together, they form a powerful foundation that directly addresses the primary objectives of modularity, portability, and reliability. This section explains the role of each technology in more detail.

2.1.1. The Containerization Strategy: Docker

Containerization is a lightweight form of virtualization that packages an application and all its dependencies into an isolated unit called a "container" [28]. For example, a container will have all the libraries, system tools, and drivers needed by the application. This approach is central to the goal of separating software from hardware, which is a key challenge in AV development. Unlike traditional Virtual Machines (VMs), which create an entire virtual computer including a full guest operating system for each application, containers share the host machine's operating system kernel [19]. This fundamental difference makes containers significantly more efficient. They are smaller in size, start up almost instantly, and use far fewer resources, like memory and storage. As a result, many more containers can run on the same hardware compared to VMs. This is a critical advantage for the great computational demands of autonomous driving systems.

The Docker platform provides a simple and powerful way to implement containerization. It is built on three fundamental concepts that work together to create portable and consistent software environments [19]. The first is the **Dockerfile**, a plain-text configuration file that acts as a recipe. It contains a list of step-by-step instructions for building a specific software environment [19]. The second concept is the **Docker Image**, which is a read-only template created by executing the instructions in a Dockerfile. An image is a self-contained, static package that includes everything needed to run a piece of software. Because images are immutable, meaning they cannot be changed once created, they guar-

antee that the software environment is always consistent [19]. The final concept is the **Docker Container**, which is a runnable instance of an image. It is the isolated process where the application actually executes. From a single image, it is possible to launch many independent containers.

This ecosystem is supported by tools that simplify development. Docker Hub serves as a public registry. It is a vast online library of ready to use images that can be downloaded as a starting point, saving developers from reinventing the wheel [28]. Another essential tool is Docker Compose. It is used for managing complex applications that require multiple containers to work together. For example, for a system with separate containers for a camera, a LiDAR, and a control interface. It uses a simple configuration file to define and run all the related containers as a single coordinated service [28].

This container-based strategy directly achieves the project's primary goals of modularity and portability. The software needed for each sensor is packaged into its own distinct Docker image, including its specific drivers, software development kits (SDKs), and ROS2 nodes. In this way, the system becomes inherently modular. Each sensor's software stack becomes a self-contained "microservice" [5]. This means it can be developed, tested, updated, or even replaced independently without affecting the rest of the system [5]. This modularity leads to exceptional portability. Since a container includes all its dependencies, it completely eliminates the common problem of software working on one machine but failing on another. A containerized sensor microservice will run identically on a simple laptop, on the vehicle's onboard computer, or in a cloud-based simulation, regardless of the underlying operating system or other installed software [19]. This hardware independence is essential for creating a reusable workflow that can be adapted to any AV.

While it might seem that adding a layer of virtualization would slow a system down, recent studies on complex autonomous driving software have shown the opposite can be true. A containerized microservice architecture built on ROS2 can actually achieve lower end-to-end latency and better CPU and memory utilization compared to a traditional deployment where all software runs directly on the host operating system [5]. In one study, mean end-to-end latency was improved by 5-8%, and maximum latencies were significantly reduced [5]. This surprising performance benefit comes from the way containers provide isolation. On a non-containerized system, all processes, from sensor drivers to operating system services, compete for the same resources, which can cause unpredictable delays. Docker uses built-in features of the Linux kernel, called namespaces and cgroups, to create a controlled environment for the application [5]. Namespaces ensure that a process inside a container can only see its own dedicated set of system resources, such as its own network stack or process list, making it feel like it is running on its own operating

system. Cgroups (or control groups) manage how much of the host's resources a container is allowed to use. So the amount of CPU time and memory. This isolation prevents other processes on the host computer from interfering with the ROS2 nodes running inside the container. By grouping related processes, this structure also allows the operating system's scheduler to manage resources more efficiently, leading to more predictable performance and lower latency. In a complex system like an AV's software stack, this optimized execution environment results in concrete performance gains, making containerization a powerful strategy for real-time systems [5].

2.1.2. The Communication Middleware: ROS2

While Docker provides the tool to encapsulate software components into independent containers, a robust communication middleware is required to connect them into a unified system. This middleware acts as the central nervous system of the data acquisition architecture, managing the flow of information between all the isolated modules. ROS2 was selected to solve exactly this critical problem. As already introduced in section 1.4.1, ROS2 was redesigned starting from ROS1 in order to meet the demanding requirements of modern robotic applications, such as autonomous driving [9]. This section provides a deeper technical analysis of the specific architectural features of ROS2 that make it the ideal communication middleware for this project, focusing on how its design directly enables the goals of modularity, real-time performance, and reliability.

The Publish-Subscribe Model of ROS2

The core design of ROS2 is built on an asynchronous communication pattern known as the publish-subscribe model. This model is fundamental to achieve the software decoupling required for a modular architecture [9]. The system is organized as a distributed network of independent programs, called nodes. Each node is designed to perform a single logical task. For example, in this project's architecture, one node is responsible for interfacing with the Ouster LiDAR, while a completely separate node manages the ZED stereo camera.

Nodes communicate by passing messages over labeled channels called topics. A node that produces data, such as the LiDAR driver, is a publisher. It sends its data onto a specific topic, for example, `/lidar/points`. Any other node that needs this data, such as a perception algorithm or a data recording tool, can become a subscriber to that topic to receive the messages [9]. The messages themselves have specific predefined structures, ensuring that all components in the system agree on the format of the data

being exchanged. For instance, the `/lidar/points` topic would carry only messages of a specific type like `sensor_msgs/PointCloud2`.

This communication architecture is described as "anonymous" because a publisher sends data without any knowledge of which nodes are subscribed to it. Likewise, a subscriber receives data without needing to know which node originally published it [9]. This decoupling enables the system's modularity and flexibility. It allows new components to be added or removed from the system without affecting existing ones. For instance, a future researcher could easily start a new experimental object detection node that subscribes to the ZED camera's image topic. This action would require no changes or restarts of the original camera driver node, making the platform exceptionally flexible and easy to extend for future research.

Real-Time Reliability of ROS2

ROS2's capabilities for building reliable real-time systems are not features built from scratch. They are inherited from its underlying communication middleware, the Data Distribution Service (DDS). DDS is an open industry standard for high-performance and real-time data exchange. It is proven and trusted in critical systems where failure is not an option, including military and aerospace applications [9]. Building ROS2 on this robust foundation was a deliberate engineering decision that makes it suitable for research in autonomous driving.

One of the most significant architectural improvements of ROS2 over its predecessor, ROS1, is its decentralized discovery mechanism. ROS1 relied on a central "roscore" master node to coordinate communication, which created a single point of failure. In contrast, ROS2 uses the peer-to-peer discovery provided by DDS, where nodes find each other automatically on the network [9]. This is essential for the reliability of a distributed multi-computer architecture, like the one in this thesis. With nodes running on both an NVIDIA Jetson and an Intel NUC, the failure of one computer will not bring down the communication network on the other.

Furthermore, DDS provides a powerful set of configurable Quality of Service (QoS) policies. These policies allow for fine control over how data is handled, enabling the communication to be tailored to the specific needs of different data streams [9]. This is particularly important for an AV, which handles a variety of sensor data with different requirements:

- **High-Bandwidth Sensor Data:** for sensors like LiDAR and cameras, the most recent data is always the most valuable. An old image or point cloud is of little use for real-time decision-making. For these topics, a "best-effort" reliability policy

is ideal. It prioritizes low latency by not attempting to retransmit a dropped data message, which would only deliver old information. The history policy can be set to "keep-last" with a small queue depth to ensure that the system only processes the very latest sensor reading [9].

- **Control and State Data:** for lower-bandwidth but critical information, such as vehicle status from the CAN bus or localization data from the GNSS, losing a message could be problematic. For these topics, a "reliable" reliability policy is used to guarantee that every message is delivered [9].
- **Static Configuration Data:** for data that is published infrequently but must be available to any node that joins the network at any time, the durability policy can be set to "transient-local". This saves the topic, ensuring that the last published message is saved and automatically sent to any new subscriber that appears on the network [9].

The possibility to precisely tune the communication behavior for each data stream is a key feature that makes ROS2 a good choice for high-performance data acquisition systems.

Combining ROS2 with a Containerized Approach

Docker and ROS2 create two distinct but complementary layers of abstraction. Docker provides hardware and operating system abstraction by packaging software into portable containers. ROS2 provides software component abstraction through its anonymous publish-subscribe model. Together, they create a system that is exceptionally modular and portable. The ROS2 communication layer works across the isolated Docker containers, assembling them in a unified system [5]. This is achieved in practice by configuring the containers to use the host machine's network stack (`network=host`), allowing the underlying DDS middleware to discover and communicate with all other ROS2 nodes on the network.

2.2. The Developed Open-Source Workflow

The central contribution of this thesis is not a single data acquisition system, but rather a comprehensive and repeatable workflow for creating one. This workflow provides a structured step-by-step strategy designed to guide researchers and developers in building modular, portable, and scalable data acquisition architectures for any AV. By leveraging the foundational technologies of Docker and ROS2, this process systematically decouples the used software stack from the underling operating system, solving the critical integra-

tion challenges that often slow down progress in autonomous driving research.

The main purpose of this workflow is to allow research teams to focus their efforts on high-level autonomy tasks, instead of wasting time over low-level complexities. The same approach can be applied to different vehicles, sensor suites, and computational platforms with minimal modification. The following sections detail each phase of this open-source workflow: initial hardware integration, host system preparation, containerization, system orchestration, and final validation. In order to simplify the understanding of those steps, there are some practical examples describing the real implemented system that was built for the validation of the thesis.

2.2.1. Step 1: Hardware Architecture and Physical Integration

The first phase of the workflow involves designing the physical architecture of the system. This requires careful selection of hardware components and a clear plan for the final capabilities of the vehicle.

Component Selection

The choice of sensors and computers is dictated by the specific research goals and the operational design domain of the vehicle. The workflow does not prescribe specific models but provides a framework for making these decisions:

- **Vehicle Platform:** the ideal vehicle for a research platform is one with an accessible and documented drive-by-wire system. This simplifies the process of sending control commands (steering, acceleration, braking) via the CAN bus, as vehicles without this feature would require the complex installation of an additional control system with physical actuators to manipulate the pedals and steering wheel. The Zhidou D1 was an excellent choice for this project due to its simple electric powertrain and straightforward drive-by-wire interface.
- **Sensor Suite:** a complete sensor suite should be selected to provide redundant 360-degree perception. Key considerations include the sensor's data rate, its physical interface, power consumption, and its resilience to different environmental conditions. For this project, a state-of-the-art suite was chosen to cover multiple modalities: a high-resolution Ouster OS1 LiDAR for 3D geometric data, a ZED X stereo camera for rich visual and depth information, a dual-antenna Piksi Multi GNSS system for precise localization with RTK corrections, and a MicroStrain IMU for high-frequency orientation and motion tracking.

- **Computational Hardware:** the computers must be selected to meet the processing demands of both the chosen sensor suite and the high-level computational tasks required for autonomy. It is often advantageous to use a multi-computer setup where the CPU and GPU work well together to distribute the workload. For example, tasks that are computationally intensive and parallel, such as processing raw camera data and running deep learning models, are best suited for a machine with a powerful GPU. The general-purpose CPU complements this by handling other system processes and sequential tasks. This distributed approach prevents a single computer from becoming a bottleneck and improves overall system scalability. In the practical implementation, an NVIDIA Jetson AGX Orin and an Intel NUC were chosen for this role

Network Topology and Wiring

Once the components are selected, they must be physically integrated. The goal is to create a robust network that can handle high data throughput without bottlenecks.

- **High-Bandwidth Data:** a dedicated Ethernet network is the backbone of the architecture. An automotive-grade Ethernet switch should be used as the central hub. All components that generate high volumes of data and communicate via Ethernet are connected directly to this switch. In this project, the NVIDIA Jetson, the Intel NUC, the Ouster LiDAR, and the two Piksi Multi GNSS evaluation boards were all connected to the central switch.
- **Direct-Attached Peripherals:** sensors with other interface types are typically connected directly to the computer best suited to process their data. For example, the ZED X stereo camera, which uses a high-speed GMSL2 interface, is connected to the NVIDIA Jetson via a GMSL2 capture card.
- **Vehicle Interface:** defining a connection link to the vehicle's CAN bus is critical for both receiving vehicle status data, like speed or steering angle, and sending control commands. A CAN to USB adapter, such as the PCAN-USB cable used in this project, provides the physical interface between the vehicle's internal network and one of the onboard computers (in this case, the Jetson).

The outcome of this first step is a complete and well-documented hardware schematic that serves as the physical foundation for the software architecture.

2.2.2. Step 2: Host System Preparation

The goal of this step is to keep the host computer's disk as clean as possible. This helps reproducibility and stability. The majority of software dependencies will be managed inside the Docker containers, not on the host.

- **Operating System:** choosing a standardized operating system (OS) for all computational units is a key step. A Linux distribution is an excellent choice for this role due to its open-source nature and strong compatibility with Docker. For this project, Ubuntu 22.04 LTS was selected as it is the primary supported OS for ROS2 Humble Hawksbill.
- **Essential Drivers:** the only software that needs to be installed directly on the host OS are the low-level drivers required to interface with the physical hardware. This includes GPU drivers (e.g., NVIDIA JetPack for the Jetson), drivers for specialized interface cards (e.g., the ZED GMSL2 capture card driver), and drivers for peripherals like the PCAN-USB adapter.
- **Containerization Engine:** the final piece of software required on the host is the Docker engine itself. This provides the runtime environment and all the tools for the containerization approach.

By the end of this second step, the host computers are prepared, but they still do not contain any of the SDKs or ROS2 libraries that will be used to interact with the sensors. This clean separation is important to the desired final modularity.

2.2.3. Step 3: The Containerization Process

This step is the heart of the workflow. During this phase, each software component of the data acquisition system is encapsulated into an isolated and portable Docker container.

Standardized Directory Structure

To ensure consistency and maintainability, each containerized module should follow a common directory structure. For a given sensor, a parent folder is created, containing the following key elements:

- `Dockerfile`: the recipe for building the container image.
- `/ROS2_ws/src`: the directory containing the source code for the ROS2 nodes.
- Launch and script files: all the scripts to build the image (`build.sh`), start the

container (`start.sh`), and record ROS2 bags (`record.sh`).

Assembling the Dockerfile

The Dockerfile is a script that contains all the commands needed to assemble the container image. The process for writing a Dockerfile for a sensor node follows a standard pattern:

1. **Base Image:** start from an official ROS2 base image (e.g., ros:humble-ros-base). This provides a clean environment with ROS2 already installed.
2. **Install Dependencies:** add the entire list of the commands to install all necessary system libraries and dependencies. This includes libraries required by the sensor's specific SDK.
3. **Set Up the ROS2 Workspace**: create a ROS2 workspace within the image and copy the source code for the sensor's ROS2 nodes into the `/ROS2_ws/src` directory.
4. **Build the Workspace:** use `colcon build` to compile the ROS2 nodes within the container image.

Sourcing and Developing ROS2 Nodes

The ROS2 software that runs inside the container is responsible for communicating with the sensor and for publishing its data onto ROS2 topics.

- **Using Official Wrappers:** whenever possible, the workflow recommends using the official ROS2 drivers and wrappers provided by the sensor manufacturer. These are typically optimized for the specific hardware. For example, official ROS2 wrappers from GitHub were used in this project for the ZED camera (zed-ros2-wrapper), Ouster LiDAR (ouster-ros), MicroStrain IMU (microstrain_inertial), and Swift Navigation GNSS (swiftnav-ros2).
- **Adapting Existing Nodes:** sometimes, minor modifications are required to suit the project's specific needs. For example, the official swiftnav-ros2 package was adapted to create a custom gnss_bringup package for a dual-antenna setup. This allowed for launching two separate nodes, each configured for a different antenna, and publishing their data to distinct ROS2 topics.
- **Creating Custom Nodes:** for hardware without an existing ROS2 driver, a custom node must be created. This was the case for the vehicle's CAN bus interface. Using the vehicle's CAN database (DBC) file, which defines the rules for encoding and decoding CAN messages, two custom Python nodes were created:

can_decoder to read all messages from the bus and publish them as ROS2 messages, and can_commander to subscribe to control topics and publish corresponding command messages back onto the CAN bus.

Building the Immutable Image

After the Dockerfile and ROS2 nodes are prepared, the `docker build` command is executed. This creates a self-contained Docker image for that specific piece of hardware. This image packages the sensor driver, all its dependencies, the ROS2 nodes, and the exact runtime environment into a single and portable container. This process is repeated for every sensor in the suite, resulting in a library of independent and modular components.

2.2.4. Step 4: System Orchestration with Docker Compose

With a complete set of Docker images, the next step is to define how to run together all of them with a single command. Docker Compose is the tool used for this orchestration. It uses a YAML file (`docker-compose.yml`) to configure and launch all the containers and all the nodes.

- **Computer-Specific Compose Files:** since this project uses a multi-computer architecture, a separate "docker-compose.yml" file is created for each machine (one for the Jetson and one for the NUC). Each file defines which containers will run on that specific computer.
- **Service Configuration:** within the Docker Compose file, each container is defined as a service. It is important to set all the parameters that define the service. Those parameters are:
  - **image:** the name of the Docker image to use.
  - **network mode:** it has to be set to "host". This is a crucial setting for allowing ROS2 to work correctly as the middleware of the architecture. It instructs Docker to bypass its virtual network layer and allow the container to share the host machine's network interface directly. This enables the underlying DDS middleware to discover and communicate with other ROS2 nodes on the network smoothly and with minimal latency.
  - **volumes:** this is used to map files or directories from the host machine into the container. This is essential for two purposes: giving the containerized software access to physical hardware devices and providing shared configuration files.

  - **commands:** this specifies the commands to execute when the container starts. This is typically some ROS2 launch command that runs the specific launch files for the sensor nodes inside the container.
  - **privileged:** for some hardware devices it has to be set to "true". This flag is necessary to grant the container the same permissions of a root user on the host system. For example, it is used for the ZED camera's container to identify which port the camera is connected to.

- **Configuring Multi-Host DDS Communication:** by default, ROS2 nodes only discover other nodes running on the same machine. To enable communication across the network between the Jetson and the NUC, the DDS middleware must be configured. This workflow uses CycloneDDS instead of the standard FastDDS, because it is the best choice for high-bandwidth data transfers. To do this modification a XML configuration file ("cyclonedds.xml") is necessary. This configuration file defines specific network parameters to ensure reliable discovery. Instead of relying on automatic discovery, which can be unpredictable on systems with multiple network interfaces, this file explicitly lists the interfaces to be used and disables auto-determination. This approach restricts all ROS2 data traffic only to the intended network. This single XML configuration file is mounted into every container across both computers, and the `CYCLONEDDS_URI` environment variable is set to point to it. This ensures that all nodes will join the same distributed ROS2 network.

2.2.5. Step 5: Final Deployment and Validation

The final phase of the workflow involves running the entire system and verifying its operation.

- **System Launch:** thanks to Docker Compose, the entire distributed data acquisition system can be launched with a single command on each machine: `docker-compose up -d`. This command reads the "docker-compose.yml" file, pulls the required images if they are not present, and starts all the defined containers in the correct configuration.
- **Verification:** once the containers are running, standard ROS2 command tools can be used inside any container of the network to inspect the system's state:
  - `ros2 topic list`: this command should display the complete list of all topics from every sensor, confirming that nodes on both the Jetson and the NUC are communicating over the same network.

  - `ros2 topic echo /<topic_name>`: this command can be used to subscribe to any simple topic, like booleans or float values, and view the data being published in real-time, verifying that a specific sensor is operating correctly.
  - `rviz`: this is another real-time visualization tool that can be used to check more complex topics, like images or point clouds.
  - `ros2 topic hz /<topic_name>`: this command measures and reports the publishing frequency of any given topic. It is useful for verifying that a sensor or node is publishing data at the expected rate.

- **Data Recording:** to collect data for offline analysis and algorithm development, the ROS2 bag tool is used. The command `ros2 bag record -a` subscribes to all topics on the network and saves the messages into a single file, creating a time-synchronized recording of the entire sensor suite. However, with advanced sensors like those used for autonomous driving, the volume of data generated can be immense. Recording all topics can result in extremely large bag files. It is a good practice to specify only the desired topics to be saved, resulting in much more manageable ROS2 bags.

By following these five steps, this workflow enables the creation of a multi-sensor and multi-computer data acquisition system. The resulting architecture is modular and portable, providing a good foundation for autonomous driving research. In fact, if ever the research team wanted to add a new sensor, they just need to follow the same exact steps only for the new container of the sensor, maintaining the other containers as they already are.

2.3. Implementation of the Real Data Acquisition Architecture

The open-source workflow described in the previous section provides the methodology for creating a modular data acquisition system. To validate this workflow and demonstrate its practical application, a real-world architecture was designed, implemented, and tested. This implementation serves as a concrete case study, translating the abstract principles of the workflow into a tangible system.

This section details the specific hardware and software components used to build this system. It begins by outlining the physical layer that is composed by the computers, sensors, and their interconnections. It then dives into the software layer, describing the contents of each individual Docker container.

2.3.1. The Hardware Configuration

The hardware foundation was designed to be representative of a modern AV research platform. It is composed of a complete sensor suite and a heterogeneous computing environment.

Computational Core: the system's computational tasks are distributed across two distinct computers to balance the processing load:

- **NVIDIA Jetson AGX Orin:** this high-performance System-on-Chip (SoC) serves as the primary processing unit. It is responsible for managing the data stream coming from the majority of the sensors, and for the communication with the CAN bus. Its powerful integrated GPU is crucial for handling computationally intensive tasks. In particular for the processing of high-resolution image data from the stereo camera. The Jetson is equipped with a 2TB SSD to handle the high data rates of ROS2 bags recording.
- **Intel NUC 13 PRO:** this compact x86-based computer acts as the secondary processing unit. It is responsible for processing the high-bandwidth data packets coming from the LiDAR. The high-level autonomous driving algorithms for perception, planning, and control, can be run either on this NUC or on an external processing unit that communicates with this system through MQTT.

Sensor Suite: a full suite of sensors was selected to provide redundant multi-modal perception of the vehicle's state and surroundings:

- **Stereo Camera:** a ZED X stereo camera is mounted at the front of the vehicle. It provides high-resolution images and dense 3D depth information. It is essential for visual perception tasks.
- **LiDAR:** an Ouster OS1 LiDAR with 128 channels is mounted on the front of the vehicle. It is fixed with a 13.5° angle towards the front, in order to faithfully recreate the space in front of the vehicle with a dense point cloud. It provides precise geometric data for object detection and localization.
- **GNSS:** for precise localization, the system uses two Piksi Multi Evaluation Boards from Swift Navigation, each connected to a GPS500 antenna. One antenna is placed at the front of the car and the other at the rear, enabling position, velocity, and heading estimation with RTK corrections.
- **IMU:** a MicroStrain HBK 3DM-GV7-AHRS IMU provides high-frequency data on the vehicle's orientation and acceleration.

Network and Vehicle Interface: the components are all interconnected to form a single unified network:

- **Ethernet Switch:** a central Ethernet switch acts as the network backbone. The Jetson, the NUC, the Ouster LiDAR, and the two Piksi Multi GNSS boards are all connected to this switch via Ethernet cables, enabling high-speed data transfer between them.
- **Direct Connections:** the ZED X camera is connected directly to the Jetson using a GMSL2 Fakra cable and a GMSL2 Capture Card Duo, which is a specialized interface mounted on the Jetson board. The MicroStrain IMU is also connected directly to the Jetson via a USB cable. Two USB-DB9 adapters connect the Jetson directly to the two GNSS boards, in order to allow the RTK correction messages to arrive to the GNSS boards.
- **CAN Bus Interface:** to communicate with the vehicle's internal network, a PCAN-USB cable connects the Jetson to the Zhidou D1's CAN bus. This interface allows the system to subscribe to all vehicle status messages and to publish control commands.
- **MQTT:** this protocol is used to establish a communication link with an external computing unit. This connection is dedicated to V2I communication, enabling the system to send and receive data from an external infrastructure.

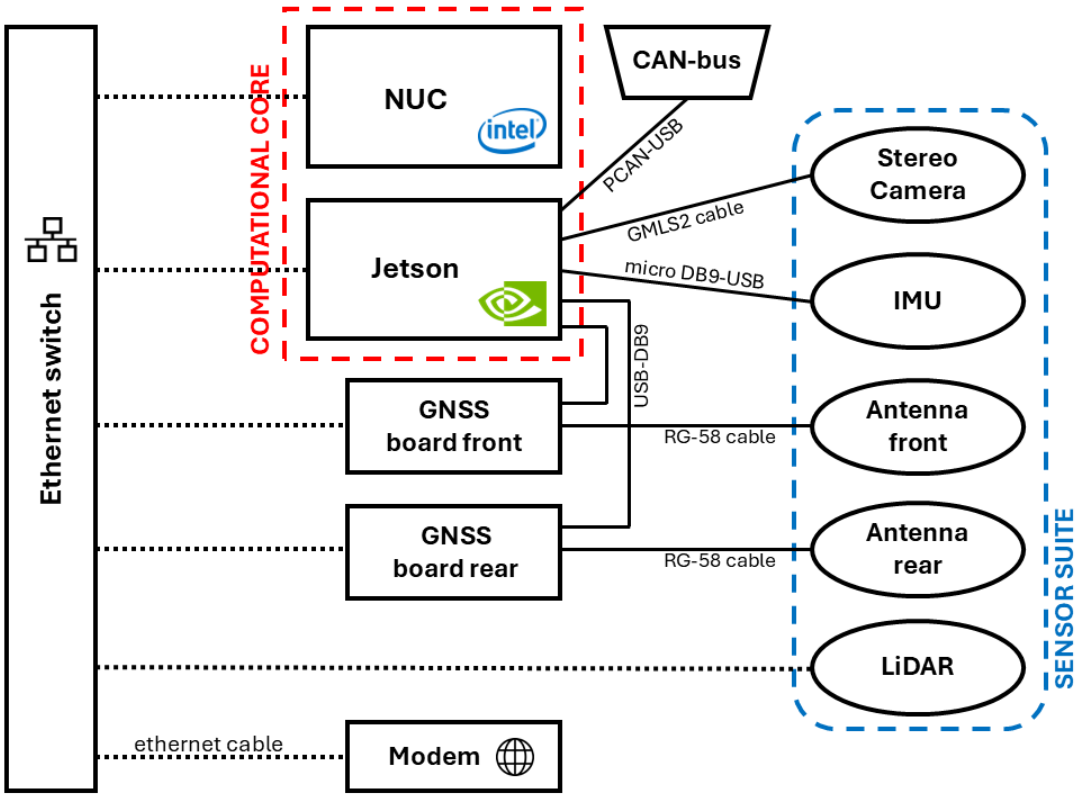


Figure 2.1: Diagram of the hardware configuration

2.3.2. The Docker Containers

Following the workflow described in section 2.2, the software for each sensor was encapsulated into a dedicated Docker container. This approach isolates each component and its dependencies, creating a set of modular and portable building blocks that form the core of the data acquisition architecture. This microservice philosophy ensures that each sensor's software stack can be developed and used independently. Five distinct Docker images were created, each designed to manage a specific piece of hardware. Each image was built using a Dockerfile that starts from a base image and then installs the specific drivers, libraries, tools, and ROS2 nodes required. The following sections provide a detailed analysis of each of these five modular components.

Docker Image for the Stereo Camera: zed-ros2-image

The image is used for managing the ZED X stereo camera. This is one of the sensors in the suite that produces the most data and requires the most computation. Its main job is to get synchronized stereo image pairs and process them in real time. This generates a rich set of data, including color images with high resolution, dense depth maps, 3D

point clouds, and information on visual odometry. These tasks rely heavily on parallel processing. For this reason, the container is designed to run on the NVIDIA Jetson AGX Orin to use its powerful integrated GPU.

While the other four Docker images follow a standardized build pattern starting from an official ROS2 base image, the one for the ZED stereo camera requires a specialized one. Its Dockerfile starts from the "stereolabs/zed:4.2-tools-devel-jetson-jp6.0.0" image. It is provided by the manufacturer (StereoLabs), and it is already configured and optimized for the Jetson platform. It includes the ZED SDK with the exact versions of the NVIDIA drivers and CUDA toolkit. These are needed for stereo processing that is accelerated by the hardware properties. This method makes the build process much simpler and less likely to have errors. It ensures the best performance and stability. However, this optimization makes the image less portable. The resulting image is closely tied to the specific SoC architecture of the NVIDIA Jetson. On top of this special base, the Dockerfile installs the needed ROS2 environment and tools. It then clones the official zed-ros2-wrapper, also provided directly by the manufacturer. Finally, the ROS2 environment is compiled using the ROS2 command `colcon build`.

Inside the container, the main ROS2 node is: **zed_wrapper**. Once it is running, this node publishes many topics. The most important of them for research in autonomous driving are:

- **/zed/zed_node/rgb/image_rect_color:** a sensor_msgs/Image message containing the rectified color image from the left camera. This is the primary input for visual perception algorithms like object detection and semantic segmentation.
- **/zed/zed_node/depth/depth_registered:** a sensor_msgs/Image message representing a depth map. It is a black and white image where each pixel's value corresponds to its distance from the camera.
- **/zed/zed_node/point_cloud/cloud_registered:** a sensor_msgs/PointCloud2 message that provides a dense 3D point cloud generated from the stereo depth information, offering a geometric representation of the scene with associated color data.
- **/zed/zed_node/odom:** a nav_msgs/Odometry message that publishes the camera's estimated pose and velocity, calculated using an algorithm that fuses data from the camera and its built in IMU.

zed-ros2-image	
Hardware Managed	ZED X camera
Base Image	stereolabs/zed:4.2-tools-devel-jetson-jp6.0.0
Key Libraries/SDKs	ZED SDK 4.2
ROS2 Nodes	zed-ros2-wrapper

Table 2.1: Key information about zed-ros2-image

Docker Image for the LiDAR: ouster-ros2-image

This image is for managing the Ouster OS1 LiDAR. Its job is to connect with the sensor over the Ethernet network, process the raw data it receives, and publish on the ROS2 environment a 3D point cloud with high resolution. This point cloud gives a precise geometric map of the area in front of the vehicle. The point cloud is a fundamental part of the perception system. It is essential for tasks like object detection, SLAM, and avoiding collisions.

The ouster-ros2-image is built starting from the standard "ros:humble-ros-base" image. Its Dockerfile follows a more traditional pattern for making images. First of all, it installs a set of C++ libraries. These are required for the Ouster sensor's main SDK. They include "libeigen3-dev" for math operations on point cloud data, "libjsoncpp-dev" for reading the sensor's configuration and metadata, and "libtins-dev" for handling basic network packets. This shows that even when using an advanced ROS2 wrapper, it is important to manage the dependencies of the core driver that are not part of ROS. The image then includes and builds the official ouster-ros driver package provided by the manufacturer.

This image has only one ROS2 node: **ouster-ros**. It is launched with a simple command in which you need to specify the LiDAR's static IP address. This confirms that the sensor communicates directly over the vehicle's Ethernet network. In the launch command of the node, it is also possible to set some variables, like the resolution of the point cloud and the start and end points of the azimuth window. For example, in the practical implementation on the car, the LiDAR was set to scan only a 180° horizontal window in front of the car. The main ROS2 topics that this container uses are:

- **/ouster/points:** a sensor_msgs/PointCloud2 message. This is the main output, providing the dense 3D point cloud that represents the surrounding environment. Each point in the cloud contains its x, y, z coordinates, as well as intensity, reflectivity, and other data specific to the sensor.

- **/ouster/imu:** A sensor_msgs/Imu message that publishes data from the LiDAR's integrated inertial measurement unit. This data is valuable for correcting distortions in the point cloud caused by the vehicle's movement during a scan.
- **/ouster/range_image, /ouster/signal_image, /ouster/reflec_image:** a series of sensor_msgs/Image topics that provide 2D image like representations of the LiDAR data. These can be useful for certain visualization or processing techniques that are optimized for image formats.

ouster-ros2-image	
Hardware Managed	Ouster OS1 LiDAR
Base Image	ros:humble-ros-base
Key Libraries/SDKs	libeigen3-dev, libjsoncpp-dev, libtins-dev, (...)
ROS2 Nodes	ouster-ros

Table 2.2: Key information about ouster-ros2-image

Docker Image for the GNSS: gnss-ros2-image

The gnss-ros2-image manages the GNSS system, which is composed of two Piksi Multi Evaluation Boards from Swift Navigation and two GPS500 antennas. Its job is to provide a very accurate absolute position, velocity, and heading for the vehicle, with precision at the centimeter level. It is designed to communicate with both GNSS boards and allow for the input of RTK correction data to get the best precision.

This image is built from the "ros:humble-ros-base" image. It uses a strong strategy for managing an important external dependency. The Swift Navigation ROS2 driver talks to the Piksi devices using the Swift Navigation Binary Protocol (SBP). To read these messages, the driver needs the "libsbp" C library. The Dockerfile clones the "libsbp" repository from GitHub, selects a specific version (v4.11.0), and compiles it from the source code inside the image. This method ensures a stable and repeatable build. It locks the dependency to a version that is known to work. This prevents future updates to the library from compromising the image. The Dockerfile also installs the Python libraries "paho-mqtt" and "pyserial". They are used by the "**rtk-write.py**" script to send RTK correction data to the GNSS boards through the USB-DB9 adapters. The two Python libraries are also used by a ROS2 node (**ros2_mqtt_bridge**) that establishes a communication link with an external MQTT broker and publishes the received data in the ROS2 environment. This node is used in V2I communication. The transmitted data

can be of different types. For example, during the final tests, this node was used to get a simple boolean value. The ros2_mqtt_bridge node is not directly related to the GNSS system. The gnss-ros2-image was selected to also contain this extra ROS2 node because it is the only one with the libraries "paho-mqtt" and "pyserial".

The container holds the official swiftnav-ros2 driver and a custom **gnss_bringup** package. This custom package was made to adapt the official driver for the project's unique hardware configuration. It has two separate launch files ("antenna_front.launch.py" and "antenna_rear.launch.py"). These files start and configure a driver node for each antenna on its own. This is done in order to have the position messages of the two antennas published on two separate topics (/antenna_front and /antenna_rear). Three separate processes are started inside the container: the "rtk_write.py" script and one launch file for each antenna. This shows an important practice adopted. The configuration and launch files are changed to fit the specific hardware without changing the main driver code provided by the manufacturer.

The most important topics used by this container are:

- **/antenna_front/navsatfix:** a sensor_msgs/NavSatFix message providing the absolute geographic coordinates (latitude, longitude, altitude) from the front antenna.
- **/antenna_front/twistwithcovariancestamped:** this topic will contain messages of the geometry_msgs/TwistWithCovarianceStamped type providing the linear and angular velocity of the front antenna.
- The same set of topics is published under the **/antenna_rear/** namespace for the rear antenna.

gnss-ros2-image	
Hardware Managed	dual-antenna Swift Navigation GNSS
Base Image	ros:humble-ros-base
Key Libraries/SDKs	libsbp pyserial, paho-mqtt (for RTK)
ROS2 Nodes	swiftnav-ros2, gnss_bringup ros2_mqtt_bridge (for V2I communication)
RTK Script	rtk_write.py

Table 2.3: Key information about gnss-ros2-image

Docker Image for the IMU: imu-ros2-image

This is a container with the specific job of connecting with the MicroStrain HBK 3DM-GV7-AHRS IMU. It provides measurements of the vehicle's linear acceleration and angular velocity at a high frequency. This data is essential for algorithms that estimate the vehicle's state, like an Extended Kalman Filter. These algorithms combine it with data from sensors that update less often, like GNSS, to create a smooth and continuous estimate of the vehicle's state.

The container is built starting from the "ros:humble-ros-base" image. Its Dockerfile installs many dependencies needed by the ROS2 driver used. These dependencies include some ROS2 packages and "libgeographic-dev". This is a C++ library for converting geographic coordinates, which the driver needs to handle global coordinate frames.

The container runs the official microstrain_inertial ROS2 driver from the manufacturer (HBK). In the command used to launch it, you need to specify the model that you are using. In this case, is the GV7 series of IMUs. This ensures the correct communication parameters and device settings. The main topics that this container publishes are:

- **/imu/data:** a sensor_msgs/Imu message. This is the main data output. It contains the vehicle's orientation (as a quaternion), angular velocity (in rad/s), and linear acceleration (in m/s^2). The message also includes covariance matrices that show the uncertainty of these measurements, which is important for sensor fusion algorithms.
- **/imu/data_raw:** a sensor_msgs/Imu message that provides the unfiltered data directly from the IMU's internal sensors. This can be useful for advanced filtering or for diagnostics.

imu-ros2-image	
Hardware Managed	MicroStrain IMU
Base Image	ros:humble-ros-base
Key Libraries/SDKs	libgeographic-dev
ROS2 Nodes	microstrain_inertial

Table 2.4: Key information about imu-ros2-image

Docker Image for the CAN bus: can-ros2-image

The can-ros2-image container is the link between the autonomy software that uses ROS2 and the vehicle's internal network. It works in two directions. First, it listens to all activity on the vehicle's CAN bus. It decodes the raw binary messages into useful physical values, like vehicle speed or steering wheel angle, and publishes them as ROS2 messages. Second, it subscribes to a set of ROS2 command topics. It translates complex commands, like "set the brake pressure to 20 bar", into the correct CAN messages needed to control the car by wire.

This container is built starting from the "ros:humble-ros-base" image and has several key tools for CAN communication. The Dockerfile installs "can-utils" and "iproute2". These are standard Linux tools for setting up and debugging CAN interfaces on the host machine. More importantly, it installs two essential Python libraries. The first one is "python-can", which provides the basic software interface to the CAN hardware through the host's PCAN driver. The second one is "cantools", which is a powerful library for reading DBC files.

This container depends completely on the vehicle's DBC file, "ZD1_plus.dbc". A CAN bus sends raw data frames that contain a numerical ID and a series of bytes. This data has no meaning without a guide to explain it. The DBC file is this guide. It defines the rules for how to decode the bits and bytes of a message with a certain ID into a physical value.

can-ros2-image	
Hardware Managed	CAN bus
Base Image	ros:humble-ros-base
Key Libraries/SDKs	python-can, cantools, can-utils, iproute2
ROS2 Nodes	can_decoder, can_commander

Table 2.5: Key information about can-ros2-image

This container is unique because it has two custom Python nodes that were developed for this project: **can_decoder** and **can_commander**. This design choice cleanly separates the functions for reading and writing.

- The can_decoder node subscribes to all incoming CAN traffic. It publishes many topics that match the signals defined in the DBC file. Key topics include /vehicle/velocity and /can_bus/scu__steering_wheel_angle.

- The can_commander node subscribes to a set of command topics. These topics are the control interface for any advanced planning software. These topics include /cmd/ref_steering_wheel_angle, /cmd/ref_throttle, /cmd/ref_brake_pressure, and several boolean flags like /cmd/ref_speed_control_enable that act as safety features.

3 | Tests and Results

This chapter presents a systematic and rigorous evaluation and validation of the proposed modular data acquisition architecture. The primary objective of the experimental campaign is to empirically validate the key architectural properties central to this thesis: modularity, portability, reliability, and real-time performance. The successful demonstration of these properties is fundamental to validate the architecture as a reusable and scalable framework for autonomous driving research.

The chapter begins by describing the experimental platform and the methodology employed for data collection, ensuring the reproducibility and scientific credibility of the results. Following this, dedicated sections address each architectural property under investigation. Each section outlines the specific tests conducted, presents the data collected, and provides an analysis of the findings. The results are presented through a combination of tables and graphical visualizations to facilitate interpretation and to support the conclusions drawn about the system's capabilities.

3.1. The Test Platform: The Car

Figure 3.1: The experimental vehicle with the implemented data acquisition architecture

Powertrain & Performance	
Engine Type	Battery Electric Vehicle
Max Power Output	24 hp (approx. 18 kW)
Max Torque	82 Nm
Top Speed	80 km/h
Drive Type	Front-Wheel Drive
Battery & Charging	
Battery Gross Capacity	11.5 kWh
Battery Chemistry	Lithium-ion
Distance Range	120 km - 145 km
Charging Standard	220V AC
Dimensions & Weight	
Body Style	3-door, 2-seat Hatchback
Length - Width - Height	2763 - 1539 - 1524 mm
Wheelbase	1765 mm
Curb Weight	670 kg
Chassis & Brakes	
Front Suspension	Independent McPherson type
Rear Suspension	Dependent spring suspension
Brakes (Front/Rear)	Disc / Disc
Tires	145/60 R13

Table 3.1: Zhidou D1 Technical Specifications

The modular data acquisition architecture proposed in this thesis can be integrated in every possible AV. In order to carry out some real-world experimental tests, it was mounted on an autonomous driving research car of the Mechanical Department of the Politecnico di Milano. It is based on a two-seat all-electric microcar, called Zhidou D1 (ZD D1). This vehicle was selected for its suitability within a controlled university campus environment. Its small dimensions, low weight, and modest top speed make it a safe and manageable vehicle for experimental development and testing.

The ZD D1's core driving functions (acceleration, braking, and steering) can all be electronically controlled. This is a crucial feature for an AV project, as it allows for a straightforward control solution through the vehicle's CAN bus. It enables drive-by-wire control

without requiring extensive mechanical modifications. In practice, the can_commander node will publish digital messages on the CAN bus, that will then be read by the ECUs, that finally will actuate the vehicle. The key technical specifications of the Zhidou D1 are summarized in Table 3.1.

A steel structure was installed on the roof of the car, in order to simplify the mounting of the sensors. The stereo camera, the LiDAR, and the rear GNSS antenna are fixed on that same structure. The LiDAR is mounted with an angle of 13.5°, in order to better scan the area in front of the car. The front antenna is attached directly onto the car hood. Back in the trunk there is the entire computational core, together with the IMU, the ethernet switch, and the internet modem. A complete visualization of the hardware setup is displayed in Figure 3.2.

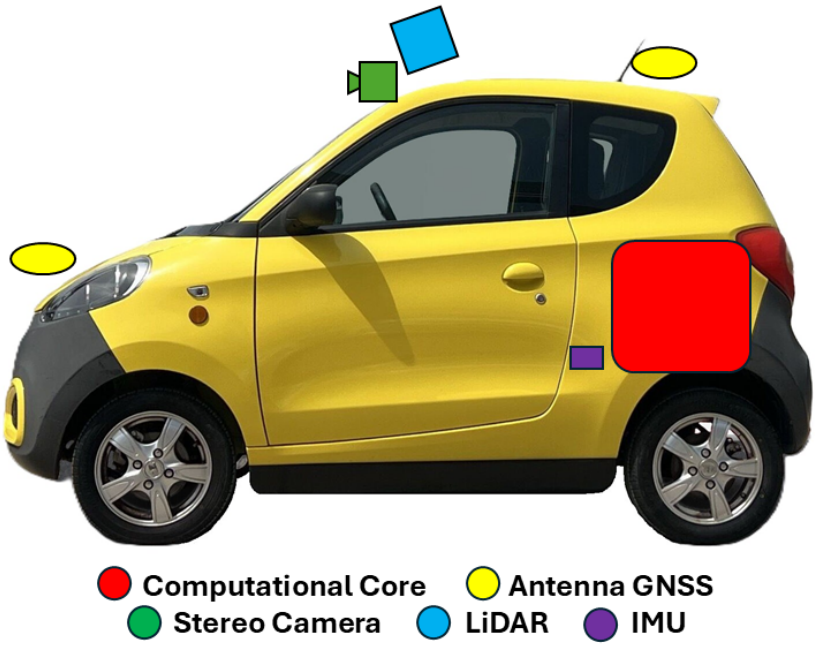


Figure 3.2: Diagram of the hardware configuration on the car

Two distinct operational conditions were used for testing:

- **Stationary tests:** the vehicle was powered on with all systems active, but it remained stationary. This condition provides an initial baseline for measuring system performance and stability without the complications associated with vehicle motion. In this setup it is easy to check the conditions of the entire instrumentation at all time.
- **Dynamic tests:** the vehicle was manually driven in a controlled environment within the Politecnico di Milano campus. The data gathered under this condition represents

a real-world scenario, as the architecture is subjected to the vibrations and dynamics characteristic of road operations.

3.2. Modularity and Maintainability

A main point of this thesis is that using containers makes the architecture proposed more modular than traditional software systems. Modularity means the system is made of separate blocks that are independent and interchangeable. This section presents the results of functional tests designed to provide empirical evidence supporting this claim. The goal is to demonstrate that the software components are effectively decoupled, enabling them to be developed, used, and updated in isolation.

3.2.1. Test: Component Isolation and Independent Operation

This test was designed to verify that each sensor's software stack, encapsulated within its own Docker container, can be managed independently without causing cascading failures or interference to other components. This is a fundamental prerequisite for a truly modular system.

Methodology

The test was conducted under stationary conditions with the full system operational. The normal operation of the system was first established by using `ros2 topic hz` to confirm that all key topics were being published at their nominal rates. The following sequence was then executed:

1. The ZED stereo camera container (zed_ros_container) was stopped using the `docker stop` command.
2. Immediately following the stop command, the status of all key topics was reevaluated using `ros2 topic list` and `ros2 topic hz`.
3. The ZED stereo camera container was then restarted using `docker start`.
4. The topic status was verified again to confirm that the stereo camera node had successfully rejoined the ROS2 network and resumed publishing.
5. This stop-start cycle was repeated sequentially for all the other containers.

Results and Analysis

It is possible to see a `rqt_bag` visualization of the results in Figure 3.3. In particular, it was recorded a bag containing a single topic for each one of the five containers. The `rqt_bag` tool reports the published messages in time of all five topics. In every sequence, the stopping of a specific container resulted only in the termination of the topics published by that container. All other components continued to operate without interruption, publishing their data streams at the nominal frequencies. For example, when the zed_ros2_container was stopped, the `/zed/zed_node/rgb/image_rect_color` topic paused publication, but the `/ouster/points` and `/vehicle/velocity` topics remained stable. After restarting the container, its corresponding topics reappeared on the network and resumed publishing data.

These results provide evidence for the effective decoupling of the system's software components. The containerization strategy successfully isolates each sensor's runtime environment, preventing any failure in one module from affecting the rest of the system. This is a key feature for a research platform, as it allows developers to work on a single component without needing to restart the entire data acquisition stack.

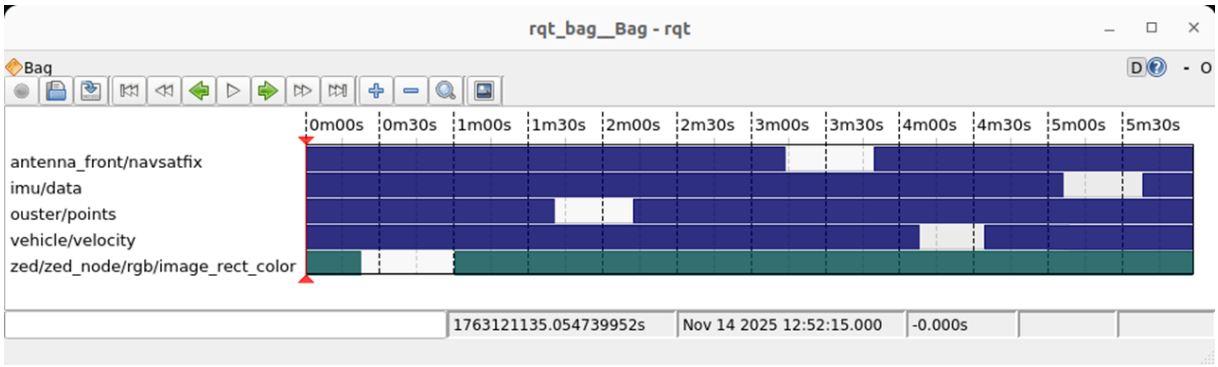


Figure 3.3: Visualization of the test "Component Isolation and Independent Operation" using `rqt_bag` tool.

3.2.2. Test: Portability and Resource Reallocation

This test aimed to demonstrate the portability of the containerized components, a key advantage of decoupling software from the underling operating system. The objective was to migrate a software module from one physical computer (the NVIDIA Jetson) to the other (the Intel NUC) with minimal configuration changes, proving that the architecture is not tied to a single configuration.

Methodology

The Ouster LiDAR container (ouster_ros2_container) was selected for this test. The LiDAR connects via Ethernet to the central network switch, making its data already accessible to any computer on the network. For this reason, no changes to the hardware configuration were needed.

1. First, the system was set up with the LiDAR container running on the NVIDIA Jetson. This was defined in its "docker-compose.yml" file.
2. The system was started, and the successful publication of the `/ouster/points` topic was verified.
3. The system was then shut down. The same exact container was then built in the NUC. The "docker-compose.yml" files for both the Jetson and the NUC were modified. The service definition for the LiDAR was removed from the Jetson's file and copied into the NUC's file. No changes were needed to the container itself. The only modification were related to the display permissions in the starting script.
4. The whole system was started again using `docker compose up` on both machines.
5. The ROS2 network was inspected to verify that the `/ouster/points` topic was being published correctly from the NUC and was accessible to all the nodes running on the Jetson.

Results and Analysis

The reallocation was successful. After moving the container, the LiDAR node started on the NUC and began publishing data to the unified ROS2 network. Nodes running on the Jetson were able to subscribe to the `/ouster/points` topic without any issues, demonstrating good communication across the Ethernet connection. The same results of portability could be obtained with the other three containers used for the GNSS, IMU, and CAN bus. On the other hand, the ZED stereo camera container can't be used on the NUC, as it has some hardware dependencies on the NVIDIA Jetson.

The key result is that the reallocation was very simple. No dependencies had to be re-compiled, and no code had to be changed substantially. The entire process consisted of rebuilding the same container on the other computer, using the exact same Dockerfile, slightly correcting just the starting script. This test powerfully validates the claim of portability. It proves that, if hardware dependencies are satisfied, the containerized modules are self-contained blocks that can be easily moved across different computers in the

architecture. This flexibility is invaluable for a lot of different practices, like balancing computational load, hardware upgrades, and adapting the architecture to different vehicle platforms with different computational resources.

3.3. Data Integrity and System Reliability

For an autonomous driving car, the ability to transmit data correctly and operate stably over extended periods is not negotiable. This section evaluates these critical properties. The tests are designed not just to check for errors, but to validate that the system correctly implements the chosen Quality of Service (QoS) policies defined by the ROS2 middleware. The integrity of a data stream in a real-time system is not always about ensuring the delivery of every single message. It is about matching the communication behavior with the specific requirements of the data being transmitted.

3.3.1. Test: Data Completeness under "Best Effort" QoS

High-bandwidth sensor data, such as LiDAR point clouds and camera images, is often transmitted with a "Best Effort" QoS policy. In this configuration, the middleware prioritizes low latency delivery of the most recent data, accepting that occasional packet loss may occur under high network load. This test was designed to quantify the message loss for these topics under real-world operating conditions.

Methodology

1. A ROS2 bag file was recorded during a 5-minute dynamic test. The bag saved the content of the two high-bandwidth topics: `/ouster/points` (nominal 10 Hz) and `/zed/zed_node/rgb/image_rect_color` (nominal 30 Hz).
2. A Python script was written to process the recorded data.
3. The script went through all messages for both topic in the bag file, simply counting the total number of messages received and stored.
4. The expected number of messages was calculated based on the bag's duration and the sensor's nominal publishing frequency (e.g., $300\ \text{s} \cdot 10\ \text{Hz} = 3000$ messages for the LiDAR).
5. The message loss percentage was then calculated as: $(1 - \frac{\text{Actual Messages}}{\text{Expected Messages}}) \cdot 100$.

Results and Analysis

The results of the data loss analysis are presented in Table 3.2. A very small percentage of message loss was observed. In particular for the point clouds, there wasn't even any. The loss of a single image frame is generally not a problem for perception algorithms, which value the most recent data far more than a complete history of all frames. The observed loss rate of less than 1% demonstrates that the network backbone is not excessively congested and that the "Best Effort" policy is functioning correctly.

In order to solve the message loss above average for the image frames topic, it is possible to use its compressed version. Using it will reduce considerably bandwidth usage. This comes with a little increase of computational power used for the activities of compression and decompression of the messages.

	Expected Messages	**Recorded Messages**	**Message Loss**
`/zed/zed_node/rgb/image_rect_color`	9000	8921	0.88%
`/ouster/points`	3000	3000	0%

Table 3.2: Data loss analysis for high-bandwidth topics using "Best Effort" QoS

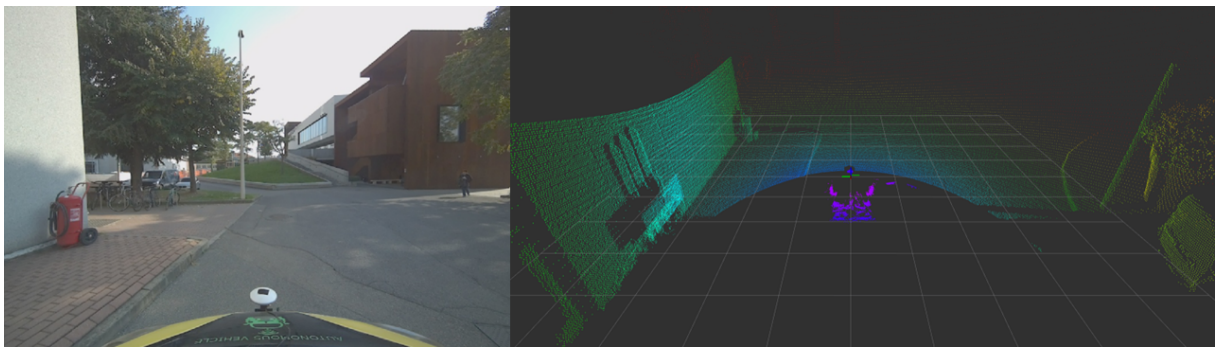


Figure 3.4: Visual representation of the content of `/ouster/points` and `/zed/zed_node/rgb/image_rect_color`

3.3.2. Test: Guaranteed Delivery with "Reliable" QoS

In contrast to sensor data, critical information such as vehicle state and control commands must be transmitted with absolute certainty. For these topics, a "Reliable" QoS policy is used. In this configuration, the DDS middleware is instructed to guarantee delivery,

retransmitting messages if necessary. This test was designed to verify that this guarantee persists, even across the distributed multi-computer network.

Methodology

1. The container of the IMU was run on the secondary processing unit, the NUC. While all the other four container were running on the Jetson.
2. A ROS2 bag file was recorded during a 2-minute dynamic test run, saving some critical topics, like: `/vehicle/velocity`, `/antenna_front/navsatfix`, and `/imu/data`.
3. A script was used to count the total number of messages for the three topics in the recorded bag file.
4. The expected number of messages was calculated based on the recording duration and the sensor's nominal publishing frequency, as done for the previous test.
5. The packet loss was calculated by comparing the recorded message count to the expected count. For a "Reliable" topic, the expected loss is zero.

Results and Analysis

The test confirmed the robustness of the "Reliable" QoS policy. The system recorded every message that was expected to be sent, as can be seen in Table 3.3.

	Expected Messages	**Recorded Messages**	**Message Lost**
`/vehicle/velocity`	12000	12000	0%
`/antenna_front/navsatfix`	1200	1199	0.08%
`/imu/data`	12000	12004	+0.03%

Table 3.3: Data loss analysis for critical topics using "Reliable" QoS

This result of nearly zero data loss over a sustained period and across the physical network link provides strong validation for the architecture's ability to handle safety-critical data. It demonstrates that the underlying DDS middleware is correctly configured and is successfully guaranteeing the delivery of all essential information.

3.3.3. Test: System Stability in Extended Operation

Beyond short-term performance, the data acquisition system of an autonomous driving car must be reliable for extended operational periods. This endurance test was designed to identify any potential issues like process crashes or thermal throttling that might only manifest over extended operations.

Methodology

1. The complete data acquisition system was launched on the vehicle in the stationary test condition.
2. The system was left to run continuously for a period of 14 hours.
3. During this period, two monitoring processes were active. First, a script periodically saved the output of `docker ps` to track the status and uptime of all running containers.
4. Second, another script periodically executed `ros2 topic hz` on five key topics and logged the output to a time series file.
5. At the conclusion of the 14-hour period, the system was shut down, and the output files were analyzed for any anomalies.

Results and Analysis

The architecture demonstrated stability throughout the 14-hour test. The `docker ps` output confirmed that all five containers (zed, ouster, gnss, imu, can) remained in the "Up" state for the entire duration, with no unexpected restarts or crashes.

The analysis of the topic frequency logs further confirmed this stability. Figure 3.5 shows the publishing rate of five key topics over the 14-hour period. The rates remained consistently stable around their nominal values, with no dropouts or periods of inactivity. This indicates that the ROS2 nodes within the containers were operating without crashing and that the communication across the ROS2 network remained robust. The results of this long-duration test provide evidence of the architecture's reliability for research activities.

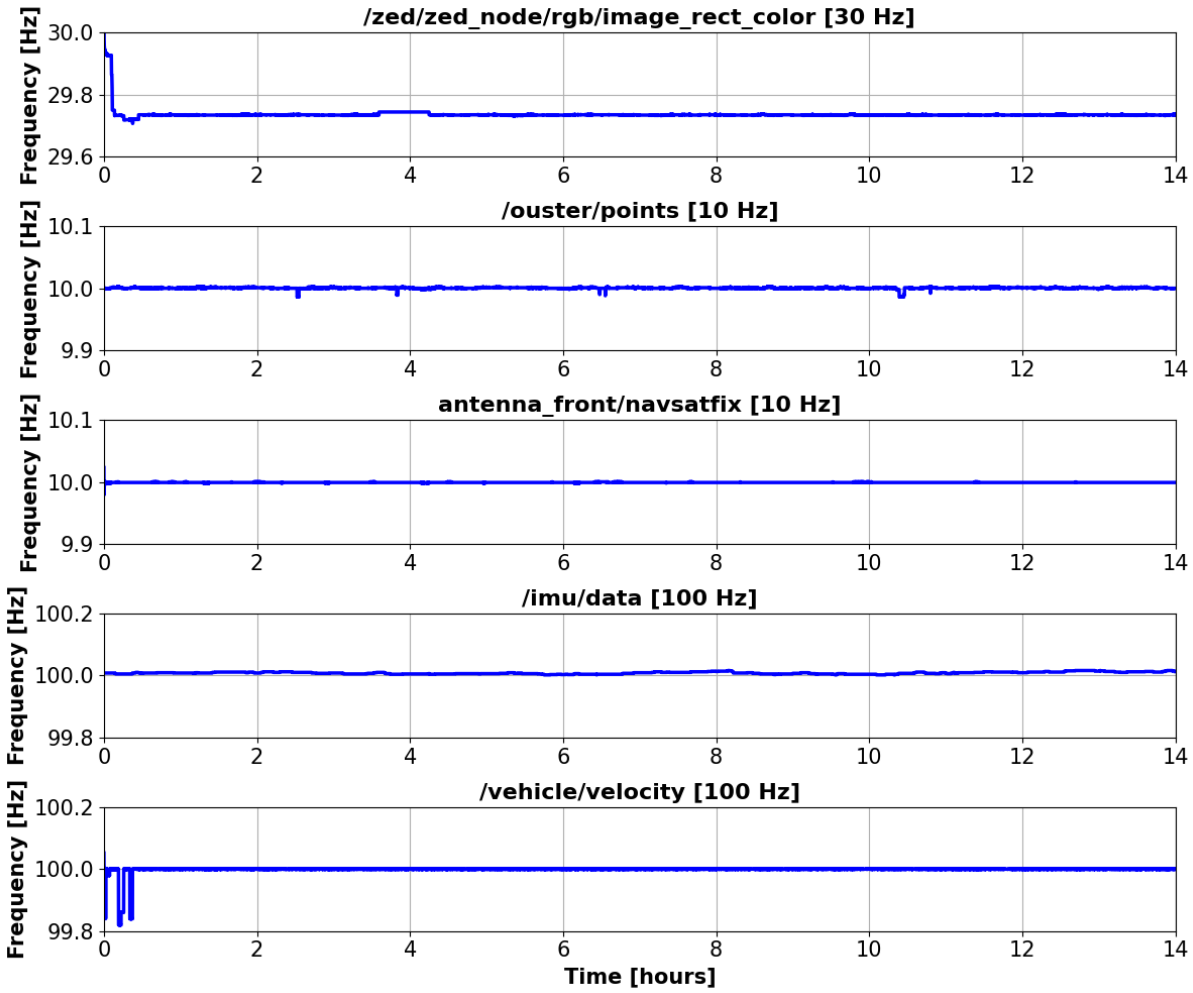


Figure 3.5: Publishing frequency of 5 key sensor topics for a 14-hours long stability test

3.4. Real-Time Performance and Latency Analysis

For autonomous driving applications, it is important for safety and performance to process data quickly. The whole process from sensing to acting must be very fast, usually in the order of a tenth of a second. For this reason, a detailed analysis of the system's speed is necessary. This section evaluates the system's real-time performance. It measures message frequency, total delay, and network usage. The analysis looks at the delays caused by ROS2 and by the network cable between the two computers.

3.4.1. Test: Publishing Frequency and Jitter Analysis

This test measures the actual publishing rate of each key sensor topic. It also evaluates how consistent the rate is. The variation in time between messages is called jitter. Low

jitter is good because it makes the system more predictable, which is important for control algorithms.

Methodology

1. The test was done during a 2-minute drive to measure performance in a realistic situation.
2. The `ros2 topic hz` command was run for each of the 5 key topics listed in Table 3.4.
3. The output was recorded for each topic. It includes the average rate, minimum and maximum delay between messages, and the standard deviation of the delay (jitter).

Results and Analysis

The results are summarized in Table 3.4. The average publishing rates for all topics were very close to their nominal values. This confirms that the sensor drivers and ROS2 nodes are working correctly and have enough computing resources.

	Average Freq. [Hz]	**Min Delay [s]**	**Max Delay [s]**	**Jitter [s]**
`/zed/.../image_rect_color`	29.735	0.014	0.090	0.00477
`/ouster/points`	10.000	0.084	0.120	0.00348
`/antenna_front/navsatfix`	9.999	0.073	0.128	0.00603
`/imu/data`	100.008	0.007	0.012	0.00033
`/vehicle/velocity`	100.000	0.008	0.012	0.00027

Table 3.4: Publishing Frequency and Jitter of some key topics

The jitter was low for all topics. This was especially true for the high-frequency IMU and CAN bus data, like the vehicle's velocity. This means the data streams are very predictable in their timing. To show this better than a simple average, a bar chart of the time between messages for the /ouster/points topic is shown in Figure 3.6. The plot shows that most messages arrive very close to the 100 ms median time. There are very few outliers. This low-jitter performance is very important for sensor fusion algorithms that need predictable data arrival times.

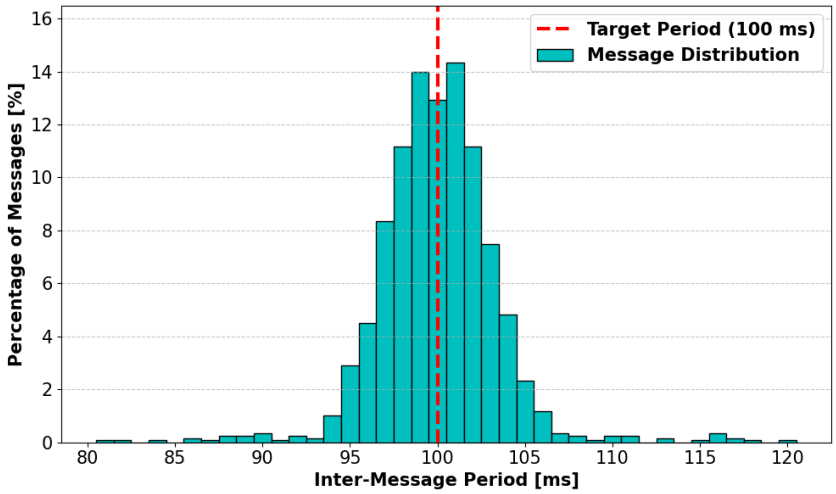


Figure 3.6: Distribution of inter-message period for /ouster/points topic in a 2-minute window

3.4.2. Test: End-to-End Latency Analysis

This test was designed to measure the real-world end-to-end latency of a complete communication loop. This is a very important data, because it defines the safety of the system. The configuration that was used in order to complete this evaluation reflects the V2I principle of modern AVs. This test goes beyond simple internal network measurements. It measures the time for data to leave the vehicle, be processed by an external computer, and return back as new processed information.

Methodology

1. The internal clocks of the Jetson and of the external computer were synchronized using chrony.
2. The test was run while the car was manually driven in a controlled environment.
3. The ros2_mqtt_bridge node, located in the gnss_ros2_container, was active. It subscribed to the internal /antenna_front/navsatfix topic and published the RTK-corrected GNSS position to an online MQTT broker.
4. The external computer was subscribed to this GNSS topic on the MQTT broker. This computer ran a high-level algorithm for obstacle detection (wrote by a fellow student).
5. This algorithm used the position of the car and measurements from an extra LiDAR to predict the time to collision with any obstacle, like pedestrians or other cars. The

output of this algorithm was a simple boolean message that was published on the MQTT broker with a 10 Hz frequency. If the boolean was "true" the collision was imminent.

6. The second node (mqtt_warning_bridge_node) on the car was subscribed to this warning topic. When it received the boolean message, it immediately published it to the internal ROS2 topic /pedestrian_warning.

7. The latencies to be measured are two. The first one was measured from the moment the GNSS position of the car is acquired ($T1$) to the moment it is available to the external computer ($T2$). The second latency defines the time that the boolean message takes to go from the external computer ($T3$) to the ROS2 environment of the car ($T4$), passing through the MQTT broker.

Results and Analysis

The final results are extracted from two 5-minute bags that were recorded during the test on the two computers used. The results in Table 3.5 show the statistics for the two latencies defined before. The total time of the first latency ($T2-T1$) includes: the time the ros2_mqtt_bridge (ROS2 to MQTT) takes to publish the message on the online broker, and the time that the external computer takes to get the message from the broker. On the other hand, the second latency ($T4 - T3$) includes: the time the external computer takes to publish the boolean value on the online broker, and the time needed by the mqtt_warning_bridge_node (MQTT to ROS2) to get the boolean value from the broker and publish it in the ROS2 environment of the car.

	Average Latency [ms]	**Minimum Latency [ms]**	**Maximum Latency [ms]**	**Standard Deviation [ms]**
$T2 - T1$	39.68	32	51	6.24
$T4 - T3$	40.71	32	54	6.31

Table 3.5: V2I end-to-end latency statistics

3.4.3. Test: System Bandwidth Utilization

This test measures the total amount of data produced by the sensors. It also measures the actual bandwidth used on the network cable. This information is important to check that the network is not a bottleneck and to plan for adding more sensors in the future.

Methodology

1. The system was run during a drive to create a realistic data load. All the containers were active. In particular, the NUC was subscribed to some topics coming from the Jetson, including also the compressed frame images of the ZED camera. And the Jetson was subscribed to the point clouds messages coming from the NUC.
2. The standard Linux tool `iftop` was used on the NVIDIA Jetson and on the NUC to monitor the traffic on their Ethernet ports.
3. `iftop` shows the real-time transmit and receive rates. The total bandwidth use was watched for 5 minutes, and the peak values were recorded.

Results and Analysis

During the drive, the system produced a large and constant stream of data. The Ouster LiDAR and the ZED camera were the main sources of network traffic. The measured bandwidth use of the entire system is summarized in Table 3.6

Communication Channel	**Average Bandwidth**	**Peak Bandwidth**
Jetson ⇒ NUC	168.4 Mbps	171.3 Mbps
Jetson ⇐ NUC	45.1 Mbps	53.3 Mbps
NUC ⇒ LiDAR	3.7 Kbps	4.8 Kbps
NUC ⇐ LiDAR	127.0 Mbps	127.9 Mbps
Jetson ⇒ GNSS	14.1 Kbps	15.4 Kbps
Jetson ⇐ GNSS	50.6 Kbps	53.6 Kbps
	TX peak	**RX peak**
Jetson	171.3 Mbps	53.4 Mbps
NUC	53.3 Mbps	299.2 Mbps

Table 3.6: Bandwidth utilization of each communication channel

This measurement confirms that the sensors produce a lot of data. However, the peak of 299.2 Mbps (RX peak of the NUC) is lower than the 30% of the total 1 Gbps capacity of the Ethernet network. This leaves a lot of network capacity for the future. This could be used for adding more sensors or sending higher resolution data.

3.5. System Resource Utilization

It is important to know how much computer power the data acquisition architecture uses. This helps to check its efficiency and to make sure there is enough power left for the main autonomy algorithms. This section measures the use of CPU, RAM, and GPU. It looks at the usage for each container and for the whole system.

3.5.1. Test: Per-container Resource Use in Extended Operation

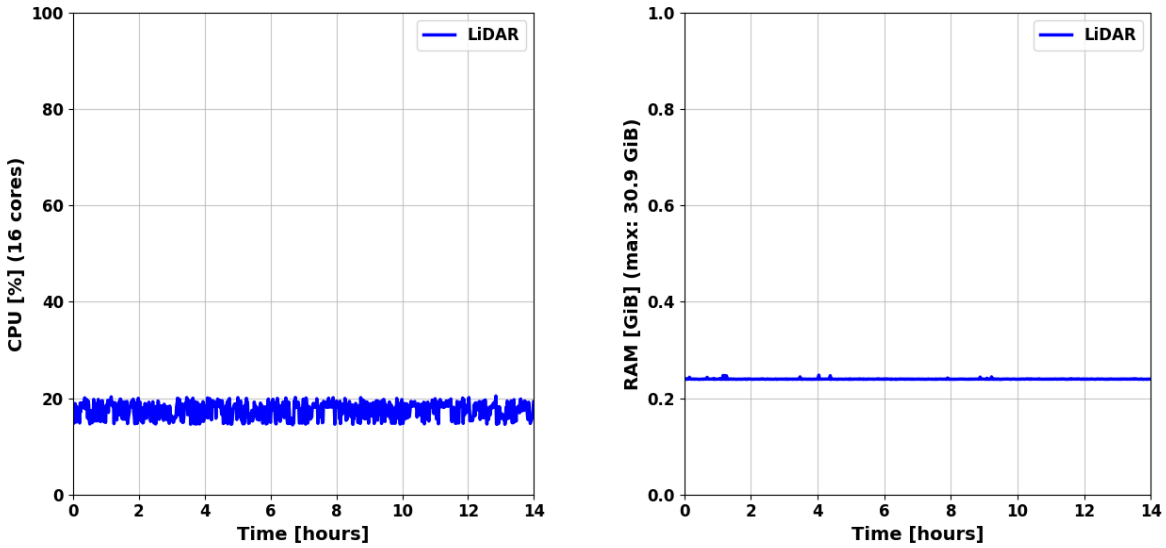
This test measured the normal CPU and RAM usage of each Docker container during a 14-hour static test. This was done to find which parts of the architecture use the most resources.

Methodology

1. The full system was run while the car was not moving to get stable and repeatable measurements.
2. The standard `docker stats` command was used on both computers. This command shows a live table of resource use for all running containers.
3. The system ran for 14 hours while CPU and RAM usage for each container were recorded.

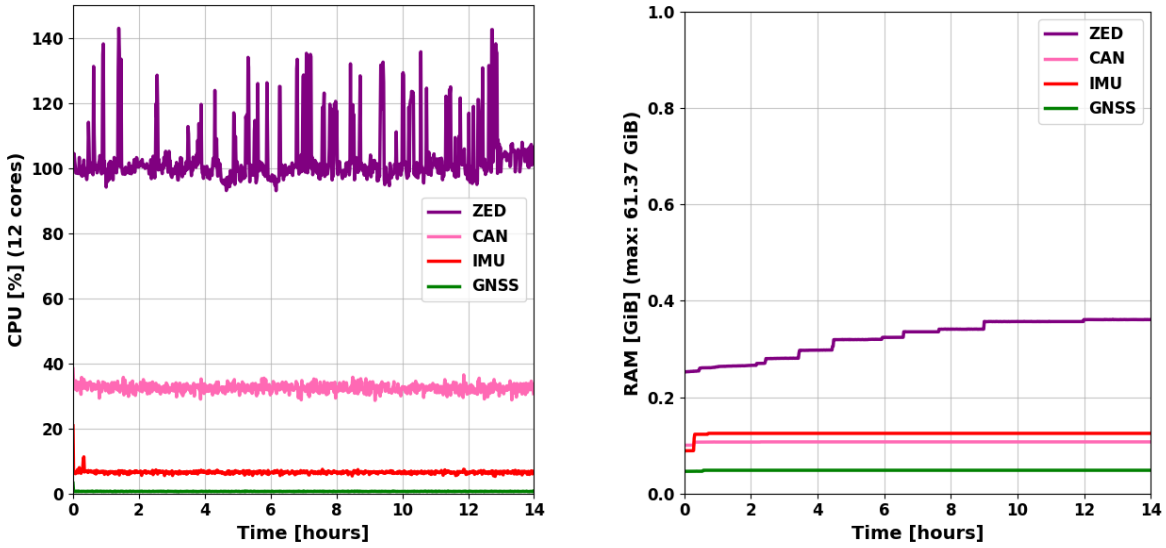
Results and Analysis

The resources used for each container are shown in Figures 3.7 and 3.8. The results clearly show that all the containers together are far away from reaching the maximum amount of computing power available. The zed_ros2_container is the one that uses the most resources in the Jetson, both in term of CPU and RAM. This is expected, because the ZED software does complex calculations for stereo depth and visual odometry. This analysis gives a clear view of the computing cost of each module of the system. The majority of the computing power is still available for the high-level autonomous driving algorithms, that can be run either directly on the car or computing core on an external computer talking via MQTT to the car.



(a) (b)

Figure 3.7: CPU and RAM used by the LiDAR container running in the Intel NUC during the long duration test.



(a) (b)

Figure 3.8: CPU and RAM used by the 4 containers running in the NVIDIA Jetson during the long duration test.

3.5.2. Test: System-Wide Load and GPU Analysis on Jetson

This test measured the total system load on the NVIDIA Jetson AGX Orin. It focused on measuring the use of its integrated GPU. Since the hardest perception tasks use the GPU, this number is a key indicator of the system's computational bottleneck.

Methodology

The `jtop` command was used to monitor the overall system statistics on the Jetson. A comparison of two scenarios was used to see the impact of the stereo camera software:

- Scenario 1 (Baseline Load): only the containers that do not base their functioning on the GPU were started on the Jetson: can_ros2_container, gnss_ros2_container, and imu_ros2_container. The system was left to stabilize, and the overall CPU, GPU, and RAM use were recorded for 5 minutes.
- Scenario 2 (Full Load): all containers for the Jetson were started, including the zed_ros2_container. Again, the system was left to stabilize, and the overall resource use was recorded for 5 minutes.

It is important to know that while `jtop` is good for monitoring the whole GPU, it doesn't get directly the GPU use for each container. For this reason, this particular comparison method was used to estimate the load of the zed_ros2_container.

Results and Analysis

The comparison of GPU use on the Jetson is shown in Figure 3.9. The results clearly show that the perception software of the stereo camera is the only container using GPU. This test confirms that it was a good decision to use a powerful computer like the NVIDIA Jetson AGX Orin. The heavy use of the GPU shows the importance of hardware acceleration for modern autonomous driving perception software.

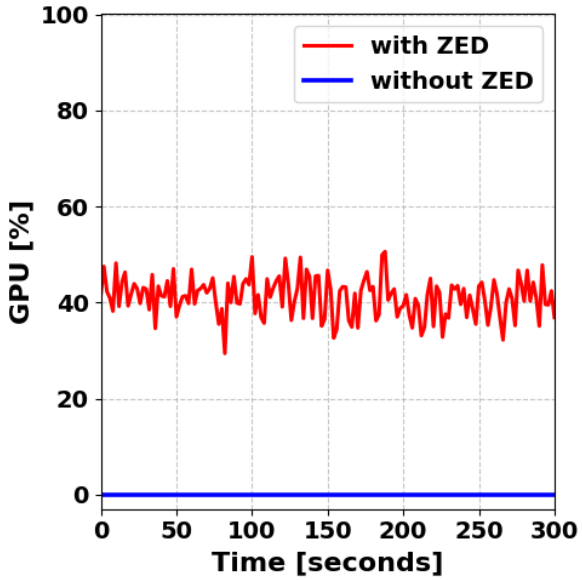


Figure 3.9: Comparison of the GPU use in the Jetson when the ZED container is active or not.

3.6. Summary of Validation Results

The tests described in this chapter provide strong proof that the proposed data acquisition architecture works well. The tests confirmed the main design goals: modularity, portability, reliability, and real-time performance. The main results from the tests are summarized in Table 3.7. This table shows what the system can do.

The results confirm that using Docker successfully isolates the software components and makes them portable. The reliability tests showed that the ROS2 QoS policies work correctly. They provide guaranteed delivery for the important data and efficient transmission for the high-bandwidth sensor data. The performance analysis showed that the architecture delivers data with low delay and low jitter. Even using an external computer for some high-level algorithms, the delays are within the limits required for real-time autonomous driving. Finally, the resource profiling measured the computational load, confirming the hardware choices.

Modularity	Confirmed
Portability	Confirmed
Data Integrity (Best Effort QoS)	$<$ 1% Packet Loss
Data Integrity (Reliable QoS)	$\simeq$ 0% Packet Loss
System Stability	$>$ 14 hours continuous operation
V2I Latency & I2V Latency	$\approx$ 40 ms

Table 3.7: Summary of the results from the validation tests

4 | Conclusions and Future Developments

This thesis confronted the foundational challenge of system integration that deeply affects autonomous driving research. As already explained throughout the thesis, traditional development approaches, which tightly couple software to specific hardware, result in fragile systems that are difficult to work with. This integration barrier slows the pace of innovation and forces researchers to repeatedly engineer solutions for data acquisition rather than focusing on higher-level autonomy tasks.

For this reason, the main goal of this thesis was not to build just a single system, but to design a complete open-source workflow for the creation of data acquisition architectures that are modular, portable, and reliable. These same three properties were proven by the extensive tests described in Chapter 3. They not only verified the quality of the final result, but also provided an empirical validation of the workflow's effectiveness.

This final chapter synthesizes the accomplishments of the thesis by summarizing its core contributions and validating its success respect to its original objectives. It concludes by presenting the future research possibilities that are now accessible as a direct result of this work. It defines the developed platform as a foundational tool for the Department of Mechanical Engineering's future research in AVs.

4.1. Conclusions

The central hypothesis of this thesis was that a strategic combination of containerization and modern middleware could effectively decouple the software components of an AV from the underlying physical hardware. This hypothesis was realized through the adoption of two foundational open-source technologies: Docker, that allows encapsulating each sensor's software stack into an isolated container, and ROS2, for its robust and decentralized publisher-subscriber communication middleware.

The primary contribution of this thesis is "The Developed Open-Source Workflow", de-

tailed in Section 2.2. This 5-step process (Hardware Integration, Host Preparation, Containerization, System Orchestration, and Validation) represents the reusable framework intended as the thesis's main goal. It provides a systematic and repeatable methodology for converting a complex set of sensors and computers into a cohesive and performant data acquisition system.

The validation strategy detailed in Chapter 3 proves that the workflow successfully produces systems with the desired properties.

- **Validated Modularity and Portability:** the workflow's promise of hardware abstraction was definitively proven. Test "Component Isolation" (section 3.2.1) demonstrated that stopping individual containers had no effect on the operation of other system components, confirming the robustness of the microservice architecture. More critically, test "Portability" (section 3.2.2) validated another core thesis goal. The Ouster LiDAR container was successfully migrated from the NVIDIA Jetson to the Intel NUC by simply rebuilding it with the exact same Dockerfile. This is concrete evidence that the workflow achieves true portability.
- **Validated Reliability and Data Integrity:** the workflow's use of ROS2 and its QoS policies was proven to be highly effective. The test for data integrity under "Best Effort" QoS (section 3.3.1) showed a negligible message loss of less than 1%. The same test, but for safety-critical "Reliable" topics (section 3.3.2) demonstrated 0% loss, even across the multi-computer network. In general, this is essential for ensuring safe autonomous driving. Furthermore, test "System Stability" (section 3.3.3) ran for over 14 hours, obtaining a continuous and error-free operation.
- **Validated Performance and Scalability:** the architecture proved more than capable of handling the demanding data load. The test presented at section 3.4.1 demonstrated low-jitter predictable data streams from all sensors. Most significantly for future development, test "Bandwidth Utilization" (section 3.4.3) showed that the entire high-bandwidth sensor suite used less than 30% of the 1 Gbps Ethernet network's capacity. This result proves that the system is capable of supporting additional sensors without any significant modifications.

In summary, the validation results confirm that the workflow produced a high-quality data acquisition system to be used as a solid foundation for future research activities.

4.2. Future Developments

The most important outcome of this thesis is not the system itself, but what it now makes possible. This work provides the Department of Mechanical Engineering with a repeatable approach that can be applied to every vehicle of the Department, as was done for the Zhidou D1 testbed. This creates a fundamental shift in our research capabilities. It allows future researchers to bypass the time-consuming process of system integration. Instead of building a platform from scratch, researchers can now utilize this validated workflow and focus more on the higher-level software algorithms.

This platform is already enabling new research. For example, a fellow student of mine, Giovanni Prinzi, is leveraging this system to develop and test his algorithms for parking activities and time-to-collision prediction with moving obstacles. His work is made possible also by this thesis, which provides the high-fidelity, time-synchronized, and low-jitter data streams he requires. In particular, he uses the messages coming from the GNSS container that are then published via MQTT to an external computing node. This collaboration is a direct demonstration of the thesis's success as an enabling platform.

This project serves as a tool for the Department's ultimate goal, that is building a complete autonomous driving software stack. This thesis has effectively created a ROS2 API for the vehicle, providing a well-documented set of inputs (the sensor topics) and outputs (the vehicle control topics). The logical next steps are to build the "brain" that connects these inputs and outputs. This roadmap includes perception and sensor fusion, which involves developing algorithms that subscribe to the provided topics (`/ouster/points`, `/zed/.../image_rect_color`, `/imu/data`, etc.) to perform 3D object detection, semantic segmentation, and robust LiDAR-camera-IMU fusion. This colud be followed by SLAM algorithms, using the data from the GNSS (`/antenna_front/navsatfix`) , IMU (`/imu/data`) , and LiDAR. The final piece of the autonomy puzzle will be planning and control. New algorithms for path planning, motion planning, and obstacle avoidance will consume data from the perception and localization layers. Their decisions will be published as commands to the control topics established by this thesis, such as `/cmd/ref_steering_wheel_angle`, `/cmd/ref_throttle`, and `/cmd/ref_brake_pressure`. The can_commander node will then execute these commands, closing the loop and achieving full autonomy.

The possibilities are infinite. The integration of the MQTT bridge and the V2I latency test are the first steps toward V2X communication research. Finally, the proven modularity and network scalability imply that the department can easily prototype and add new sensors, such as radars or event-based cameras, simply using the established workflow.

Bibliography

[1] Autosar - automotive open system architecture, 2025. URL `https://www.autosar.org/`.

[2] N. Aliane. A survey of open-source autonomous driving systems and their impact on research. *Information*, 16(4):317, 2025.

[3] R. Ayala and T. Khan Mohd. Sensors in autonomous vehicles: A survey. *Journal of Autonomous Vehicles and Systems*, 2021.

[4] Baidu. Apollo - an open autonomous driving platform, 2025. URL `https://github.com/ApolloAuto/apollo`.

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

[6] P. Buddi, C. V. K. N. S. N. Moorthy, V. Nagari, and D. Myneni. Exploration of issues, challenges and latest developments in autonomous cars. *Journal of Big Data*, 10, May 2023. doi: 10.1186/s40537-023-00701-y.

[7] comma.ai. openpilot - an open source driver assistance system, 2025. URL `https://github.com/commaai/openpilot`.

[8] *EZ10 Gen1 User Guide*. EasyMile, Toulouse, France, Jun 2018. Revision A1.

[9] S. M. et al. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 2022.

[10] T. A. Foundation. Autoware - the world's leading open-source software project for autonomous driving, 2025. URL `https://github.com/autowarefoundation/autoware`.

[11] K. Heineke, N. Laverty, T. Möller, and F. Ziegler. The future of mobility. *McKinsey Company*, Apr 2023.

[12] R. Hussain and S. Zeadally. Autonomous cars: Research results, issues, and future challenges. *IEEE Communications Surveys Tutorials*, 21(2):1275–1313, 2019. doi: 10.1109/COMST.2018.2869360.

[13] S. Jeyachandran. Introducing the 5th-generation waymo driver: Informed by experience, designed for scale, engineered to tackle more environments, Mar 2020. URL `https://waymo.com/blog/2020/03/introducing-5th-generation-waymo-driver`.

[14] H.-Y. Jung, D.-H. Paek, and S.-H. Kong. Open-source autonomous driving software platforms: Comparison of autoware and apollo. *arXiv preprint arXiv:2501.18942*, 2025.

[15] D. P. Klüner, M. Molz, A. Kampmann, S. Kowalewski, and B. Alrifaee. Modern middlewares for automated vehicles: A tutorial. *arXiv preprint arXiv:2412.07817*, 2024.

[16] L. Liu, S. Lu, R. Zhong, B. Wu, Y. Yao, Q. Zhang, and W. Shi. Computing systems for autonomous driving: State of the art and challenges. *IEEE Internet of Things Journal*, 8(8):6469–6486, 2020.

[17] M. Liu, E. Yurtsever, J. Fossaert, X. Zhou, W. Zimmer, Y. Cui, B. L. Zagar, and A. C. Knoll. A survey on autonomous driving datasets: Statistics, annotation quality, and a future outlook. *IEEE Transactions on Intelligent Vehicles*, 2024.

[18] Mobileye. Mobileye drive | self-driving system for autonomous maas, 2025. URL `https://www.mobileye.com/solutions/drive/`.

[19] P. Muzumdar, A. Bhosale, G. P. Basyal, and G. Kurian. Navigating the docker ecosystem: A comprehensive taxonomy and survey. *arXiv preprint arXiv:2403.17940*, 2024.

[20] NVIDIA. Autonomous vehicle self-driving car technology from nvidia, 2025. URL `https://www.nvidia.com/en-us/solutions/autonomous-vehicles/`.

[21] *J3016_202104: Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, Apr 2021. On-Road Automated Driving (ORAD) Committee, SAE International.

[22] B. Pal, S. Khaiyum, Y. Kumaraswamy, et al. Recent advances in software, sensors

and computation platforms used in autonomous vehicles, a survey. *Int. J. Res. Anal. Rev*, 6(1):383, 2019.

[23] C. Philpot. Infographic of the 5th-generation Waymo Drive, 2022. Published by BloombergNEF.

[24] M. Reke, D. Peter, J. Schulte-Tigges, S. Schiffer, A. Ferrein, T. Walter, and D. Matheis. A self-driving car architecture in ros2. In *2020 International SAUPEC/RobMech/PRASA Conference*, pages 1–6, 2020. doi: 10.1109/SAUPEC/RobMech/PRASA48453.2020.9041020.

[25] J. Z. Sasiadek. Sensor fusion. *Annual Reviews in Control*, 26(2):203–228, 2002.

[26] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[27] The Waymo Team. Waypoint: The official Waymo blog, 2025. URL `https://waymo.com/blog/`.

[28] S. J. Vaughan-Nichols. Docker: The software development revolution continued. Technical report, Docker, 2023.

[29] C. Wang, X. Yang, X. Xi, S. Nie, and P. Dong. *Introduction to LiDAR remote sensing*. CRC Press Boca Raton, FL, USA, 2024.

[30] H. Wu. A comprehensive of cpu and gpu performance and applications in autonomous vehicles. *Department of Unity Concord International School*, Oct 2024.

[31] Zoox. Know your ride | zoox, 2025. URL `https://zoox.com/know-your-ride/`.

[32] Óscar Silva, R. Cordera, E. González-González, and S. Nogués. Environmental impacts of autonomous vehicles: A review of the scientific literature. *Science of The Total Environment*, 830:154615, 2022. doi: https://doi.org/10.1016/j.scitotenv.2022.154615.

A | User Guide

This manual details the controlled launch method of the data acquisition system developed in this thesis. This procedure, while more time-consuming than a single docker-compose launch, is the preferred method for research and development. It provides the operator with the ability to launch, configure, and verify each component individually, which is essential for ensuring system stability and data integrity before beginning an experiment.

A.1. System Prerequisites

This section details mandatory setup procedures and pre-operation checks required to use the system.

A.1.1. First-Time Docker Image Build

Before the system can be run, the Docker images for all six modules must be built from their respective Dockerfiles. This is a one-time operation that has to be done on the computer that will be used.

For each of the six module folders (/docker-zed, /docker-ouster, /docker-gnss, /docker-imu, /docker-can, /docker-record):

1. Navigate to the module directory:

   ```
   cd /path/to/docker-<module>
   ```

2. Navigate to the Docker build folder:

   ```
   cd docker
   ```

3. Execute the build script:

   ```
   sudo ./build.sh
   ```

The "build.sh" script executes the Dockerfile, which installs all necessary dependencies (e.g., ROS2 Humble, sensor SDKs), clones the /ROS2_ws, /data, and /scripts folders

into the image, and builds the ROS2 workspace using `colcon build`.

A.1.2. Host System Configuration (Mandatory Pre-Launch)

The following commands must be executed in a terminal on both the NVIDIA Jetson and the Intel NUC before every system launch. These settings are not persistent across reboots.

These commands tune the host kernel's network parameters for CycloneDDS (the ROS2 middleware), enabling it to handle the high-bandwidth data streams from the sensors and ensuring reliable node discovery.

1. Set maximum socket receive buffer size for high-bandwidth data:

   ```
   sudo sysctl -w net.core.rmem_max=2147483647
   ```

2. Adjust IP fragmentation parameters for large data packets:

   ```
   sudo sysctl -w net.ipv4.ipfrag_time=3
   ```

   ```
   sudo sysctl -w net.ipv4.ipfrag_high_thresh=134217728
   ```

3. Enable multicast on the loopback interface for ROS2 discovery:

   ```
   sudo ip link set lo multicast on
   ```

A.1.3. Physical System Checks

Perform a brief "pre-flight" check before launching the system:

1. Verify all sensors (ZED X camera, Ouster LiDAR, GNSS Antennas, IMU) are powered on.
2. Confirm all physical connections (for reference look at Image 2.1):
   - ZED X: GMSL2 cable connected to the Jetson.
   - Ouster LiDAR: Ethernet cable connected to the network switch.
   - GNSS Boards: connected via USB to the Jetson (for RTK data) and Ethernet to the network switch.
   - IMU: connected via USB to the Jetson.
   - CAN Bus: PCAN-USB adapter connected between the Jetson and the vehicle's CAN port.

3. Confirm network connectivity: the Jetson and NUC are both connected via Ethernet to the same network switch.

A.2. Module Launch and Verification

This section provides the "controlled launch" procedure for each sensor module. The steps for each module are:

1. Host-Side Setup (if required): commands run on the host OS.
2. Launch Procedure: commands to start the container and the ROS2 nodes.
3. Verification: a table of key ROS2 topics to monitor (e.g., using `ros2 topic echo <topic_name>`) to confirm the node is operational.

A.2.1. ZED X Stereo Camera (/docker-zed)

- Target Host: NVIDIA Jetson AGX Orin (mandatory).
- Host-Side Setup:
  - Before launch, run `ZED_Explorer` on the host terminal to confirm the camera is detected.
  - Note: every time the connection with the zed is changed a reboot is necessary.
- Launch Procedure:
  1. Open a terminal on the Jetson: `cd /docker-zed`
  2. Launch the container start script: `./start.sh`
  3. Inside the zed_ros2_container, launch the appropriate node.
     - Case 1: single ZED X camera (this project's setup):

       ```
       ros2 launch zed_wrapper zed_camera.launch.py camera_model:=zedx
       ```

     - Case 2: future dual-camera setup:

       ```
       ros2 launch zed_wrapper zed_camera.launch.py camera_model:=zedx
       camera_name:=zed_front
       ```

       ```
       ros2 launch zed_wrapper zed_camera.launch.py camera_model:=zedx
       camera_name:=zed_rear
       ```

- Verification: check that topics with `/zed` nametag are being published.

A.2.2. Ouster LiDAR (/docker-ouster)

- Target Host: Intel NUC 13 Pro (not mandatory).
- Host-Side Setup: none (beyond A.1.2).
- Launch Procedure:
  1. Navigate to the repository: `cd /docker-ouster`
  2. Launch the container start script: `./start.sh`
  3. Inside the ouster_ros_container, execute the ROS2 launch file. This command specifies the sensor's IP and sets the field of view:

     ```
     ros2 launch ouster_ros sensor.independent.launch.xml \
     sensor_hostname:=192.168.1.50 \
     azimuth_window_start:=90000 \
     azimuth_window_end:=270000
     ```

     Note: this command will also open rviz2 for visual confirmation.
- Verification: check that topics with `/ouster` nametag are being published.

A.2.3. Dual-Antenna GNSS (/docker-gnss)

- Target Host: NVIDIA Jetson AGX Orin (not mandatory).
- Host-Side Setup: none (beyond A.1.2).
- Launch Procedure: This module requires three separate processes to run in parallel. The "rtk_write.py" script is essential for feeding RTK correction data to the Piksi Multi boards via their USB-serial connection.
  1. In a new terminal on the Jetson, navigate to the repository: `cd /docker-gnss`.
  2. Launch the container start script: `./start.sh`
  3. Inside the gnss_ros2_container, open three separate terminals in the container (using `tmux` or by using `./connect.sh` in a new terminal).
  4. Terminal 1 (RTK Corrections):

     /root/scripts python3 rtk_write.py
  5. Terminal 2 (Front Antenna):

ros2 launch gnss_bringup antenna_front.launch.py

6. Terminal 3 (Rear Antenna):

   ros2 launch gnss_bringup antenna_rear.launch.py

- Verification: check for topics with these nametags: `/antenna_front, /antenna_rear`.

A.2.4. IMU (/docker-imu)

- Target Host: NVIDIA Jetson AGX Orin (not mandatory).
- Host-Side Setup: none (beyond A.1.2).
- Launch Procedure:
  1. In a new terminal on the Jetson, navigate to the repository: `cd /docker-imu`.
  2. Launch the container start script: `./start.sh`
  3. Inside the imu_ros2_container, execute the ROS2 launch command:

     ```
     ros2 launch microstrain_inertial_examples cv7_launch.py
     ```

- Verification: check that topics with `/imu` nametag are being published.

A.2.5. CAN Bus Interface (/docker-can)

- Target Host: NVIDIA Jetson AGX Orin (not mandatory).
- Host-Side Setup: these commands configure the "can2" interface on the host, linking the physical PCAN-USB adapter to the Linux networking stack at the correct bitrate (500000). This must be done before the container is started.
  1. Verify the CAN interface name (e.g., "can2") using `ip a`.
  2. Set the CAN interface type and bitrate:

     ```
     sudo ip link set can2 type can bitrate 500000
     ```

  3. Set the CAN interface up:

     ```
     sudo ip link set up can2
     ```

- Launch Procedure:
  1. In a new terminal on the Jetson, navigate to the repository: `cd /docker-can`.
  2. Launch the container start script: `./start.sh`

3. Inside the can_ros2_container, open two separate terminals in the container (using `tmux` or by using `./connect.sh` in a new terminal).
4. Terminal 1 (Decoder): `ros2 run can_decoder decoder_node`
5. Terminal 2 (Commander): `ros2 run can_commander commander_node`

   Note: The decoder_ node reads from the CAN bus and publishes ROS2 topics. The commander_ node subscribes to ROS2 topics and writes commands to the CAN bus.

- Verification: check for topics with these nametags: `/can_bus, /cmd, /imu/data_can, /vehicle/velocity`.

A.2.6. Data Recording (/docker-record)

The /docker-record container includes all dependencies needed to record and analyze ROS2 bag files from all sensors. It can be also use as a starting point for the creation of a custom container that uses the data from the sensors for driving algorithms.

Procedure :

1. Ensure all desired sensor modules (A.2.1-A.2.5) are running correctly.
2. Open a new terminal on the Jetson: `cd /docker-record`
3. Launch the container start script: `./start.sh`
4. In the host terminal (outside the container) inside /docker-record, execute the recording script: `./record.sh`

   Note: The ./record.sh script is pre-configured to record a specific set of topics and to save the bag into a specific directory (in the Jetson's 2TB SSD). To record all topics or a different custom set, either edit this script or use the `ros2 bag record` command manually inside the record_ container (accessed via `./connect.sh`).

List of Figures

List of Tables

Acknowledgements

I would first like to express my sincere gratitude to my supervisors, Prof. Stefano Arrigoni and Dr. Satyesh Shanker Awasthi, for their invaluable guidance throughout my master thesis research. Their expertise was fundamental to the completion of this work. I also extend my thanks to the entire Politecnico di Milano for the knowledge and opportunities provided during these years.

Becoming an engineer has been a long journey and it would not have been possible without the support of my family and friends, that I want to thank in Italian:

"Un grazie di cuore alla mia famiglia: a mamma e papà, per il loro amore infinito e per aver sempre creduto in me. A mio fratello e alle mie due sorelline, per starmi sempre vicino e volermi bene. Una gratitudine immensa va anche alle mie due splendide nonne, e a tutti i miei cugini e zii per il loro incoraggiamento.

E per finire, un grazie gigante a tutti i miei amici. I momenti passati insieme hanno reso questi anni indimenticabili e mi hanno aiutato a superare tutti gli ostacoli del percorso.

Grazie!"