Chemnitz University of Technology
Faculty of Computer Science
Chair of Computer Engineering

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Master Thesis

Submitted to obtain the academic degree

Master of Science

submitted by

Dharam Mukeshbhai Dhameliya
(Matr. -Nr.: 701886)

Development of Real-Time and Historical Data Visualization and Configuration Tool for ROS2 Topics and ETAS Measurement Devices

Examiner: Dr.-Ing. Sebastian Heil (TU Chemnitz)
Advisors: Jan Ingo Haas M.Sc. (TU Chemnitz)
Prashantha Nettur Ramachandra Rao M.Sc. (ETAS GmbH)

Chemnitz, June 29, 2025

Abstract

This thesis designs and implements a suite of applications to visualize real-time and historical ROS2 topics. It features a web application utilizing Angular and Node.js, to facilitate cross-platform usage. These applications interface with a generic SQL database system, optimizing the storage and retrieval of voluminous ROS2 data. The integration of WebSocket ensures real-time data streaming with minimal latency, while advanced data visualization is achieved through Apache ECharts. In addition to data visualization, this project also includes the development of a graphical user interface (GUI) to configure ETAS measurement devices via IP communication, enabling users to easily modify and manage configuration settings. Security is bolstered by implementing Auth and secure communication protocols, and Docker is employed for consistent deployment across environments. This project enhances the development and debugging of robotics applications, providing scalable, secure, and effective tools for data analysis and hardware configuration.

Task Description

Development of Real-Time and Historical Data Visualization and Configuration Tool for ROS2 Topics and ETAS Measurement Devices

The rapid advancements in robotics, particularly with the introduction of Robot Operating System (ROS) 2, have revolutionized the development of sophisticated robotic applications. However, the ability to effectively visualize and configure the extensive real-time and historical data generated by these systems remains a critical challenge. Existing tools often lack the scalability, cross-platform compatibility, and feature integration required to manage high-volume, high-velocity robotic data streams in user-friendly ways. This thesis bridges the gap by designing a comprehensive application aimed at enhancing data visualization capabilities and providing intuitive device configuration for robotics developers and researchers.

Current visualization tools face significant limitations, including fragmented real-time and historical data analysis, lack of scalability for handling extensive robotic data, and insufficient security for deployments. Additionally, the need for efficient device configuration, especially for hardware like ETAS measurement devices, further complicates the development landscape. To address these issues, this thesis proposes a browser-based web application built with Angular and Node.js that enables flexible access and robust visualization. The application leverages a generic SQL database, WebSocket technology for real-time data streaming, Apache ECharts for dynamic, interactive data visualization, and a GUI for configuring ETAS measurement devices via IP communication. Security is reinforced with Auth for authentication and secure communication protocols, while Docker ensures consistent deployment across environments.

The objective of this thesis is to develop scalable, secure, and effective tools for visualizing ROS2 data and configuring ETAS measurement devices, combining real-time and historical analysis in an integrated manner. By enabling developers to efficiently analyze data streams, configure devices, and troubleshoot system performance across platforms, the project aims to enhance the development and debugging of robotics applications. A proof-of-concept prototype will demonstrate the feasibility of the proposed approach, with evaluations focused on usability, system scalability, and the impact of these tools on reducing development time and improving data-driven insights.

Contents

List of Figures

List of Tables

List of Abbreviations

ROS	Robot Operating System
LiDAR	Light Detection and Ranging
IMU	Inertial Measurement Unit
ADAS	Advanced Driver Assistance Systems
ETK	Emulator Tast Koft
ECU	Electronic Control Unit
GUI	Graphical User Interface
INCA	Integrated Calibration and Acquisition (ETAS Tool)
.db3	ROS2 SQLite3 Database Format
.mf4	Measurement Data Format v4
HIL	Hardware-in-the-Loop
DLL	Dynamic-Link Library
RCI2	Remote Command Interface 2
COM	Component Object Model
FFI	Foreign Function Interface
API	Application Programming Interface
JWT	JSON Web Token
CPU	Central Processing Unit
TCP	Transmission Control Protocol
UUID	Universally Unique Identifier
DDS	Data Distribution Service
CAN	Controller Area Network
XCP	Universal Measurement and Calibration Protocol
USB	Universal Serial Bus
TF	Transformation Frame
ASAM	Association for Standardisation of Automation and Measuring Systems
MDF	Measurement Data Format
HTTP	Hypertext Transfer Protocol
QoS	Quality of Service
OS	Operating System
SLAM	Simultaneous Localization and Mapping
MDA	Measurement Data Analyzer
BMS	Battery Management System
URDF	Unified Robot Description Format
LAN	Local Area Network

Chapter 1

Introduction

1.1 Current Situation

Modern robotics systems, especially those relying on Robot Operating System (ROS) 2, are becoming increasingly complex, modular, and data-intensive [38, 42, 27]. These systems typically involve a large variety of heterogeneous sensors such as Light Detection and Ranging (LiDAR), Inertial Measurement Unit (IMU), thermal cameras, encoders, and current sensors — all continuously publishing high-frequency data [33]. Additionally, these sensors are often coupled with automotive-grade measurement systems such as the Emulator Tast Koft (ETK) hardware family from ETAS GmbH — a Bosch subsidiary specializing in embedded systems development and automotive diagnostics — to validate, calibrate, and record external or Electronic Control Unit (ECU) based signals [19]. ETK devices are widely employed in Advanced Driver Assistance Systems (ADAS), autonomous driving, and robotics-based experimental setups that demand synchronized telemetry and high-fidelity measurement.

However, despite the richness of data and variety of tools, developers face several persistent challenges [3]. For instance, ROS2 data is typically monitored using command-line utilities such as `ros2 topic echo`, lightweight graphical interfaces like `rqt`, or specialized applications such as Foxglove Studio [38, 32]. Meanwhile, Integrated Calibration and Acquisition (INCA) — the industry-standard tool for working with ETAS devices — remains a standalone, Windows-only application with no native web-based interface or remote automation support [20]. These tools are domain-specific and were not designed to interoperate.

Consequently, development workflows are often fragmented. ROS2 telemetry is handled through its ecosystem and recorded in SQLite-based `.db3` files using rosbag [40], while ETAS measurements are recorded separately in Measurement Data Format Version 4 (`.mf4`) files via the INCA software [2]. There is no shared timeline, no cross-domain integration, and no common user interface to manage both streams of information. Engineers frequently need to switch between tools, platforms, and even operating systems to complete a single debugging or calibration session. This results in redundant effort, increased cognitive load,

and slower iteration during testing, particularly in field environments or collaborative projects involving distributed teams.

The application developed in this thesis addresses these workflow inefficiencies not by merging ROS2 and INCA data at the file or protocol level, but by providing a unified Angular based frontend through which both domains can be operated independently. This centralized user interface allows real-time visualization of ROS2 telemetry, as well as remote control and monitoring of ETAS measurement sessions — all without switching between different tools. By offering independent yet co-accessible subsystems under a single frontend, the platform enhances usability, accelerates development workflows, and improves overall engineering productivity, all while preserving the modular and domain-specific nature of the underlying technologies.

1.2 Motivation

The motivation for this project arose during the development and field testing of a self-balancing, autonomous e-bike equipped with multiple internal and external sensors. Internally, the robotic control system produced high-frequency telemetry using ROS2, including critical parameters such as IMU data, motor control states, system diagnostics, and environmental sensor readings [38, 47]. In parallel, engineers needed to acquire and validate external signals through measurement devices provided by ETAS GmbH, focusing especially on capturing ECU outputs and other high-fidelity automotive-grade measurements using the INCA software suite [20].

However, despite the technological maturity of both toolchains, development workflows were hindered by a significant degree of fragmentation. ROS2 telemetry was typically handled separately through its ecosystem of command-line tools, lightweight visualization plugins, or specialized software like Foxglove Studio. Conversely, ETAS INCA measurements were performed on dedicated Windows-based machines, utilizing standalone graphical user interfaces without remote automation capabilities [38, 20]. No integrated mechanism existed to concurrently access both systems within a unified operational framework. As a result, engineers frequently had to switch between platforms, operating systems, and data formats to analyze, monitor, or debug system behavior during critical phases of testing. This fragmented approach not only slowed development but also increased the cognitive load on engineers, especially when dealing with live experiments or time-sensitive calibration tasks.

In practice, a typical debugging session often involved two engineers: one responsible for monitoring ROS2 topics and robot behavior, and another operating the ETAS INCA interface to record or observe ECU parameters. This separation of concerns, while necessary due to tool limitations, created inefficiencies in workflow synchronization, delayed diagnosis of system faults, and made it difficult to correlate observations across robotic and measurement domains. Furthermore, remote collaboration was practically impossible because neither ROS2 native tools nor INCA provided accessible web-based interfaces suitable for distributed engineering teams.

Recognizing these limitations, the project aimed to simplify and streamline workflows by

creating a unified application that provides access to both domains through a centralized user interface. The goal was not to merge ROS2 and INCA data into a single timeline or database — since their purposes, formats, and semantics differ significantly — but rather to offer independent yet co-accessible entry points to each system from a single frontend. This approach respects the architectural and functional separation of both subsystems while dramatically improving accessibility, usability, and efficiency.

By introducing a modular platform that bridges these traditionally isolated toolchains, developers and testers can now initiate ROS2 recordings, monitor live telemetry, start or stop ETAS INCA measurements, and visualize results without leaving the application. Real-time and historical data handling are unified at the user interface level, providing consistent interaction patterns whether the user is monitoring a live experiment, reviewing recorded telemetry, or validating calibration parameters. Importantly, the cross-platform nature of the application, achieved through modern web technologies, ensures that remote collaboration and distributed debugging are feasible, opening new possibilities for agile hardware-in-the-loop (HIL) development workflows.

Overall, this project is motivated by a clear need: to reduce workflow complexity, accelerate development cycles, and enable more collaborative, scalable engineering processes in domains where robotic systems and automotive-grade measurement devices coexist but have historically been isolated by technical and platform barriers.

1.3 Background

The development of modern robotics and embedded systems increasingly requires seamless integration between data generation, processing, and measurement workflows. In systems based on ROS2, telemetry from sensors such as IMUs, LiDAR, encoders, and control modules is published at high frequencies using topic-based middleware communication [38, 42]. These messages enable real-time feedback and system state visualization, which are essential for validation, debugging, and autonomous behavior development [38].

Simultaneously, precise and time-synchronized signal acquisition remains essential for applications such as calibration, validation, and diagnostics. In automotive and embedded domains, this is typically handled using measurement systems such as those developed by ETAS GmbH. These devices interface with ECUs and analog or digital sensors to capture high-resolution physical data. The measurement process is orchestrated through dedicated desktop applications like INCA, which store results in structured formats such as `.mf4` [2, 20, 35].

In many practical deployments — including the one studied in this thesis — the ETAS measurement device is physically connected to the robotic platform. It captures relevant signals such as actuator feedback, voltage, or environmental inputs directly from the robot's hardware. These signals are not available through the robot's middleware but are essential for full-system monitoring and validation. Through INCA, engineers are able to configure, start, and stop measurement sessions that record this external data in parallel to the robot's internal telemetry.

Despite the functional overlap between robotic telemetry and hardware measurement, the toolchains for each domain remain siloed from a systems integration perspective.. ROS2 telemetry is typically visualized using native tools such as `rqt` or Foxglove Studio [38, 32, 43], while ETAS measurements are managed via desktop-based interfaces with limited extensibility. There is no built-in mechanism to jointly monitor, control, or visualize both types of data within a single workflow or interface.

To bridge this gap, the platform developed in this thesis introduces a unified user interface that enables users to interact with both telemetry and measurement systems in parallel. The system does not attempt to merge the underlying data structures or protocols. Instead, it offers a modular frontend through which both live ROS2 topics and ETAS measurement sessions can be independently accessed and visualized. A dedicated backend layer serves as a mediator between the user interface and the INCA application, enabling remote control of measurement operations through standardized commands while preserving compatibility with existing device configurations [3, 5].

ETAS GmbH Overview

ETAS GmbH, a wholly owned subsidiary of the Bosch Group, specializes in providing tools and solutions for embedded systems in the automotive industry and related domains. With a strong focus on measurement, calibration, and diagnostics, ETAS offers a suite of hardware and software products designed to support development workflows across ECUs and real-time embedded platforms. One of their flagship offerings, the INCA software, is widely used in industry for acquiring, visualizing, and analyzing automotive-grade measurement data [20].

This thesis makes extensive use of ETAS technologies, particularly the ETK hardware interface and the INCA software suite. These tools enable precise, synchronized measurement of signals that are not available through the robot's middleware stack but are essential for debugging and validating complete system behavior. The collaboration with ETAS GmbH not only provided access to industry-grade tools but also significantly informed the architectural direction and practical constraints of the platform developed in this work [19].

1.4 Problem Statement / Analysis

Problem Statement: Modern development workflows involving robotic telemetry and automotive-grade measurement systems are limited by the lack of a unified, platform-agnostic interface for real-time data visualization, system control, and historical analysis. Specifically, there is no integrated environment capable of handling ROS2 topic streams and ECU measurements concurrently within a single, cohesive interface. This fragmentation introduces significant inefficiencies across engineering tasks and hinders productivity in collaborative development settings.

In current development environments, engineers are required to operate multiple disjointed toolchains to monitor robot telemetry and collect hardware-level measurements. ROS2 offers tools for real-time topic inspection and data recording, while ETAS GmbH's INCA software provides access to external measurement devices connected to the robot. However,

these tools operate in completely separate environments — both in terms of platform (Linux vs. Windows) and interface (command-line vs. graphical). No native or standardized mechanism exists to utilize them in tandem through a shared interface.

This fragmentation leads to several operational inefficiencies:

1. Engineers must alternate between different software applications to complete a single testing session, often switching between machines or operating systems.
2. Collaboration is constrained by the local nature of INCA, which lacks web-based interfaces or remote access features.
3. Debugging processes are slowed due to the absence of synchronized visualization or control for telemetry and measurements.
4. Data collected from ROS2 and INCA cannot be observed in parallel within a unified user interface, complicating tasks like field diagnostics or validation.

These limitations hinder rapid prototyping, collaborative debugging, and system validation — especially in scenarios involving remote teams, hardware-in-the-loop testing, or real-time field trials. A modular solution that enables developers to independently but simultaneously access both data domains from a shared frontend would address these workflow inefficiencies without compromising the separation and purpose of each toolchain.

1.5 Objectives and Scopes

Project Goals

The primary objective of this thesis is to design and implement a cross-platform application that enables both ROS2 topic visualization and ETAS measurement device control through a common user interface. While the ROS2 and INCA subsystems remain logically and functionally separate, the platform provides integrated access to both from a single frontend [38, 20].

Key project goals include:

- Develop a modular platform that facilitates access to ROS2 data and ETAS measurement tooling from a shared user interface [3]
- Support real-time visualization of live ROS2 topics and playback of historical `.db3` files [40]
- Enable configuration and control of INCA experiments using a custom-developed C++ DLL, callable from Node.js via the Windows COM API and INCA Automation interface [20]
- Provide flexible and reusable charting components for various telemetry and measurement types [7, 10, 49]

- Implement authentication and session tracking using JSON Web Token (JWT) and Universally Unique Identifier (UUID) based mechanisms [30, 37]
- Package all backend services using Docker to ensure consistent deployment and portability [31]

Applications and Relevance

Although the platform was originally developed within the context of an autonomous e-bike project, its design principles and architectural flexibility enable broad applicability across various domains. In ADAS testing, the integration of perception data from ROS2 with ECU-level signal acquisition via INCA allows engineers to correlate environmental inputs with control logic in real time [38, 20]. Similarly, HIL configurations benefit from the system's ability to provide remote, parallel access to live telemetry and measurement streams, facilitating distributed debugging and system evaluation. Beyond automotive contexts, the platform proves highly relevant in robotics research and development environments where system-level validation, sensor fusion, and iterative debugging are essential to experimental workflows.

By separating the data domains but consolidating the control and visualization interface, the system enhances team workflows, improves accessibility, and reduces tool-switching overhead — particularly in environments where both ROS2 and INCA tools are needed.

Chapter 2

State of the Art

This chapter establishes the foundational context for designing a cross-domain telemetry and measurement platform that integrates robotics middleware (e.g., ROS2-based systems) with automotive-grade data acquisition environments (e.g., ETAS INCA). It highlights technological gaps, architectural limitations, and opportunities observed in current solutions, forming the basis for deriving functional and integration requirements.

Unlike the previous chapters, which introduced background and motivation, this chapter adopts a forward-looking perspective to investigate the state of middleware, data handling, and visualization tooling. The goal is not to present an implementation but to identify key capabilities and limitations in existing systems. The analysis will support the formulation of system-level requirements and inform the architectural decisions to be discussed in Chapter 3.

2.1 Technological Landscape of Telemetry and Measurement Systems

Modern robotics and automotive engineering environments operate within two parallel but historically isolated technological ecosystems: robotics middleware platforms and automotive calibration and measurement tools. This section provides an overview of these two domains to establish the foundational context required to understand their roles, limitations, and interrelation in system development and testing workflows.

2.1.1 Robotics Middleware: ROS2 Ecosystem

The ROS2 is a widely adopted middleware framework for distributed robotics applications [38]. ROS2 is based on the Data Distribution Service (DDS) [26], which provides decentralized, real-time publish-subscribe messaging for distributed nodes. Nodes in ROS2 are lightweight computational units that publish or subscribe to typed messages organized by named topics.

Key architectural features of ROS2 include:

- **Executor Model:** Nodes run inside executors that manage threading, timers, and callback scheduling.
- **Quality of Service (QoS):** ROS2 allows specification of reliability, durability, liveliness, and deadline constraints on a per-topic basis, enabling developers to balance latency, reliability, and bandwidth usage [38, 28, 26].
- **Discovery and Namespacing:** DDS-based automatic node discovery eliminates the need for a central master, and namespaces help organize large systems.

In addition to its message-passing infrastructure, ROS2 supports a layered middleware design that abstracts hardware access, sensor fusion, diagnostics, and parameter tuning. Its extensibility is enabled through modular interfaces such as launch files, transformation frames (TF) 2, and lifecycle nodes, which promote clean separation between runtime behavior and system configuration [38, 42]. The use of standardized message definitions and introspection mechanisms also allows developers to build reusable components and integrate with broader ecosystems including simulation environments, embedded microcontrollers, and cloud-based analytics [27, 3].

2.1.2 Automotive Calibration and Measurement Tools

In the automotive sector, measurement and calibration are handled through purpose-built environments. ETAS INCA is one of the most established platforms in this domain. It is used for acquiring signals from ECUs, sensors, and in-vehicle networks such as Ethernet, Universal Serial Bus (USB), Controller Area Network (CAN), and Universal Measurement and Calibration Protocol (XCP) [20, 35].

Key components of the ETAS measurement ecosystem include:

- **INCA Measurement and Calibration Environment:** A Windows-based desktop application used for configuring and executing experiments on ECUs. It enables live signal acquisition, parameter tuning, and calibration.
- **ETAS ETK Modules:** ETK devices interface with analog/digital Input/Output (I/O), CAN buses, and XCP-over-Ethernet to extract high-fidelity measurement signals from vehicle subsystems [19].
- **MDF Format:** The Association for Standardisation of Automation and Measuring Systems (ASAM) defined Measurement Data Format (MDF), typically stored as `.mf4` files, is used for time-aligned, high-resolution storage of multi-channel signals, supporting metadata layers such as units, annotations, and events [1, 17].
- **MDA (Measurement Data Analyzer):** A companion tool that allows users to visualize, annotate, and compare measurement data, often used for post-processing calibration workflows.
- **Automation Interfaces:** Scripting capabilities are provided through COM interfaces and RCI2, which allow external tools to configure experiments, access variable data, or control runtime behavior programmatically [20, 18].

ETAS tooling is engineered for precision, traceability, and standard compliance, making it particularly suitable for regulated industries such as automotive safety systems or powertrain development [41]. However, its design remains strongly oriented toward desktop-based workflows. Integration with open-source systems or web-native environments is limited by several factors.

Platform Restriction: INCA is designed to operate exclusively on Microsoft Windows environments, leveraging native Windows APIs for device drivers and user interface rendering.

License and Access Model: Full functionality often requires licensed hardware connections and elevated system permissions, which can restrict use in flexible or containerized deployment scenarios.

Data Flow Paradigm: Measurement workflows are typically predefined and sequential. Real-time streaming into external tools or mixed-domain orchestration is not a native feature.

While the ETAS ecosystem provides unparalleled support for ECU-level experimentation and standardized file output, its closed architecture and dependency on dedicated environments pose challenges for engineers seeking to integrate it with open, modular systems such as those common in robotics or cloud-based testing infrastructures.

2.1.3 Siloed Ecosystems and Cross-Domain Limitations

Despite maturity within their respective domains, ROS2 and ETAS INCA reflect two fundamentally different design philosophies and operational models.

ROS2: An open-source, modular middleware designed for distributed robotics applications. It supports Unix-like platforms (especially Linux) and leverages community-driven standards and tooling [38, 42].

INCA: A proprietary, monolithic platform focused on measurement and calibration in automotive environments. It is tailored for Microsoft Windows and built around closed-loop experiments and vendor-controlled interfaces [20].

They differ notably in the following aspects:

- **Platform Dependencies:** ROS2 favors Linux for native compilation and performance tuning, and it benefits from extensive community support on Ubuntu and other distributions. INCA, in contrast, is designed strictly for Windows due to its integration with ETAS-specific device drivers, COM-based automation layers, and GUI rendering stack.
- **Data Formats:** ROS2 stores telemetry using structured SQLite-based `.db3` files through the `rosbag2` utility [40]. These files support type-safe serialization and replay, offering a flexible and inspectable logging format. INCA utilizes the ASAM MDF4 format, a highly efficient binary layout optimized for large-scale signal logging with metadata layers [1, 35]. However, MDF files are often unreadable without specialized tooling like MDA or proprietary parsing libraries.

- **Access Models:** ROS2 encourages streaming, dynamic data introspection and real-time debugging through DDS and command-line or scriptable tools. In contrast, INCA is structured around pre-configured experiments and assumes offline post-processing workflows. Although automation is supported through RCI2 and COM APIs, it requires platform-specific configuration and access rights [20, 18].

These fundamental differences have resulted in siloed engineering workflows [3, 5]. Roboticists typically use ROS-native tools like `rqt`, Foxglove Studio, or terminal-based commands to inspect sensor data and debug control logic. Automotive engineers, on the other hand, depend on ETAS tools for acquiring and analyzing ECU-level signals using CAN, XCP, or analog interfaces.

There is currently no standard mechanism to unify these workflows under a shared interface or synchronized timeline. This gap introduces inefficiencies in development and testing workflows, particularly in cross-domain contexts such as robotic vehicles, cyber-physical test benches, or embedded HIL setups. Engineers are forced to switch between platforms, adapt data formats manually, and synchronize observations by approximation, increasing the cognitive load and risk of misalignment [45, 23].

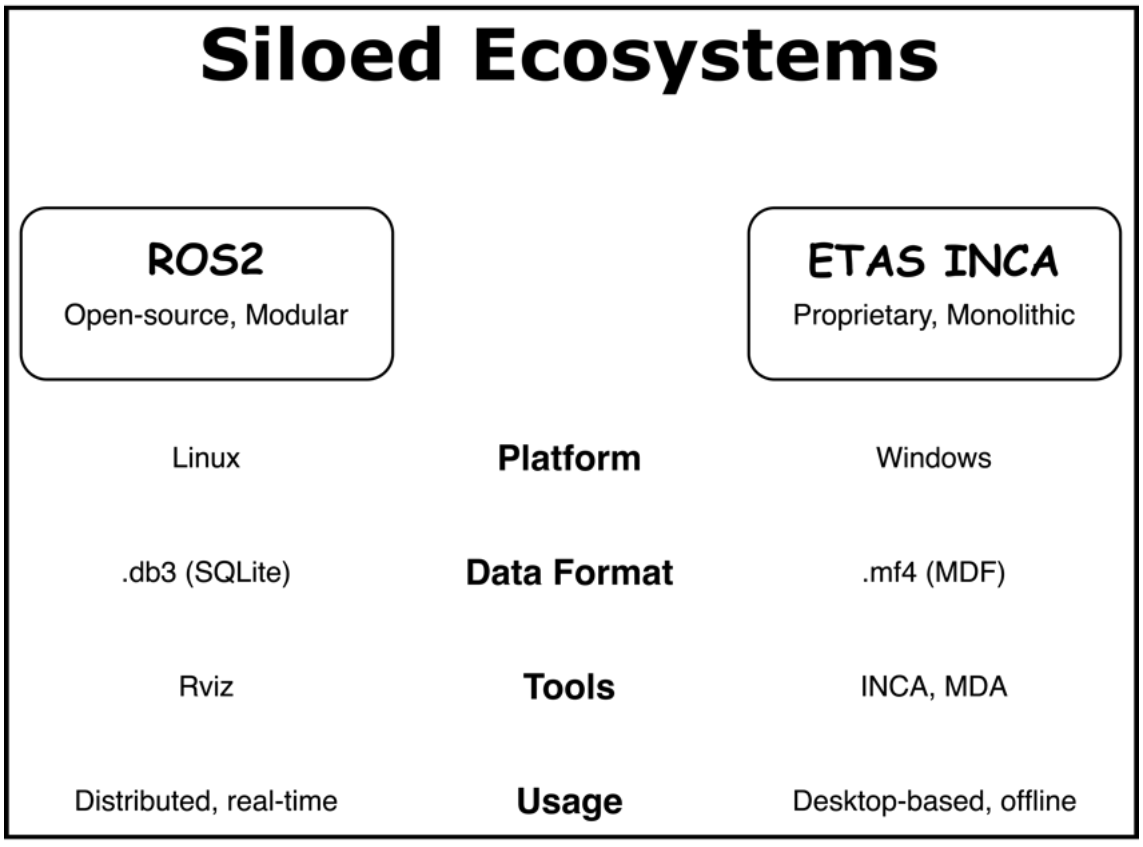

Figure 2.1: Comparison of the ROS2 and ETAS INCA ecosystems

2.2 Middleware Foundation and Communication Principles

Middleware technologies form a foundational layer in distributed software systems by enabling seamless communication and coordination between heterogeneous components [44, 42, 3]. In the context of robotics and embedded systems, middleware abstracts the complexity of networking, message serialization, and timing, allowing developers to design modular, scalable, and platform-independent applications [44].

This section introduces the communication principles and middleware strategies commonly employed in robotic and measurement-intensive environments. It outlines communication paradigms, interface models, and quality-of-service configurations that support real-time interoperability across domain boundaries such as automotive and robotics.

2.2.1 Middleware in Distributed Embedded Systems

Modern robotics and automotive systems consist of numerous interacting modules, often distributed across different physical machines or operating systems. Middleware facilitates interaction among these modules by providing standardized APIs and protocols for data exchange, decoupling the application logic from the underlying transport mechanisms [26, 6, 27, 42].

Common benefits of middleware in such environments include:

- **Modularity:** Each functional unit (e.g., sensor driver, control algorithm, visualizer) can be developed and deployed independently.
- **Scalability:** Middleware supports horizontal scaling by enabling seamless addition of nodes or services.
- **Fault Isolation and Resilience:** Middleware manages communication loss, retries, and reconnections, enhancing system robustness.
- **Cross-platform Operation:** Abstraction of OS-specific communication details allows interoperability between Windows- and Linux-based subsystems.

Middleware solutions in robotics often leverage real-time publish-subscribe mechanisms, whereas those in automotive environments may prioritize deterministic control and secure point-to-point protocols [38, 20, 28].

2.2.2 Communication Paradigms: Topics, Services, and Events

Publish-Subscribe (Topics): Producers emit messages on named channels without knowledge of consumers. This model is widely used in robotics for high-frequency telemetry (e.g., sensor feeds, control signals). ROS2 implements this via DDS, which handles message discovery, serialization, and delivery across distributed nodes. The publish-subscribe model supports loose coupling, scalability, and parallel processing. However, it may introduce nondeterministic behavior in lossy networks unless carefully tuned using QoS parameters such as reliability, durability, and deadline. These parameters

allow developers to optimize for latency-sensitive control or bandwidth-constrained environments.

Request-Response (Services): Clients send structured requests and await replies from specific providers. This model is common for control commands or parameter adjustments, such as starting a logging session, setting configuration values, or querying internal states. It offers synchronous interaction with guaranteed feedback, but the blocking nature of service calls can limit their use in hard real-time systems. To mitigate this, many systems implement asynchronous service wrappers or offload such tasks to non-critical threads. ROS2 service interfaces are defined using '.srv' files that describe the request and response structure.

Event-Driven Messaging: Nodes can emit discrete events in response to system changes. These are often used for state transitions, faults, or triggers [5]. Events differ from topics in that they usually occur sporadically and signal specific occurrences rather than continuous streams. This paradigm enables reactive programming styles and is particularly useful in safety-critical scenarios, where alarms, completion flags, or watchdog timers must be propagated quickly [5, 43]. Proper event handling requires priority-based execution and logging mechanisms to ensure no critical event is missed or delayed.

Each paradigm provides specific trade-offs in terms of latency, reliability, and coupling. Hybrid systems often use a combination of these paradigms to balance reactivity and control fidelity across components. For example, a robotic control system might stream sensor data via topics, initiate calibration through a service call, and handle emergency stops using event notifications [5].

2.2.3 Quality of Service (QoS) in Middleware Configuration

To meet the performance demands of real-time systems, middleware frameworks typically support QoS parameters [26, 28]. These configurations define the behavior of message delivery under varying network and system conditions:

- **Reliability:** Guarantees message delivery via acknowledgments (reliable) or allows best-effort transmission. Reliable QoS ensures that no messages are lost during transport, which is essential for mission-critical data such as safety flags or actuator commands. In contrast, best-effort delivery is suitable for high-frequency sensor data (e.g., LiDAR or IMU), where occasional packet loss is tolerable and preferable to delayed transmission.
- **Durability:** Specifies whether messages should be stored and delivered to late-joining subscribers. In scenarios where a node may temporarily disconnect or launch late (e.g., visualization clients or data loggers), transient-local or persistent durability ensures they receive previously published data. This is especially important in calibration setups or parameter broadcasting where values must persist beyond initial publication.
- **Liveliness and Deadlines:** Monitors message publishing activity and detects time-outs in node communication. Liveliness settings help detect node failures or

unresponsiveness by expecting periodic updates, while deadline policies enforce time constraints between consecutive message publications. This is critical in closed-loop control or monitoring systems where data must arrive at regular intervals to maintain synchronization and operational safety [28].

- **Latency Budget and History Depth:** Allows configuration of message delivery timing and buffering behavior. Latency budget sets the acceptable delay between message publication and reception, enabling tuning for applications with tight response time requirements. History depth controls how many past messages are stored per topic, which can be tuned for consumers that process data at different rates. For instance, visualizers may use a deeper history, whereas real-time controllers require only the latest state.

These settings are critical in coordinating message flows across systems with differing temporal constraints, such as high-frequency sensor streams and low-frequency calibration commands. Proper tuning of QoS profiles helps balance data integrity, responsiveness, and resource usage in heterogeneous systems, especially when combining robotics middleware and automotive-grade instrumentation.

2.2.4 Cross-Platform Middleware Challenges and Synchronization Strategies

A key challenge in cross-domain systems lies in enabling real-time, bidirectional communication between components that operate on dissimilar operating systems and middleware stacks. For instance, robotic systems often rely on ROS2 running in Linux-based environments, while automotive-grade measurement tools such as ETAS INCA operate exclusively within Microsoft Windows [38, 20, 35]. These systems not only differ in OS platforms but also in communication protocols, data serialization formats, and runtime architectures, making interoperability non-trivial.

Bridging such heterogeneous environments requires a careful orchestration of translation, abstraction, and synchronization mechanisms across both system and network layers. One common approach is to use cross-platform proxies or middleware relays that forward structured data between operating system-specific nodes using TCP sockets, WebSockets, or RESTful APIs. These gateways act as intermediaries that abstract platform dependencies and expose a unified interface for communication, enabling components written for different runtime environments to exchange data seamlessly.

Another essential strategy involves embedding precise timestamp information within each message. Time-aligned data encoding ensures that sensor readings, actuator signals, or measurement values captured on one platform can be correctly correlated with events on another, despite differing sampling rates or execution schedules [45]. In practice, this requires synchronization of system clocks, often through protocols such as NTP or PTP, to ensure minimal temporal drift.

To enable parsing and exchange of proprietary data formats such as '.db3' (used in ROS2) or '.mf4' (used in INCA), encapsulation techniques are applied. These involve wrapping binary payloads into structured, platform-agnostic formats such as JSON or Protocol Buffers. This allows external middleware components to extract, convert, and process

measurement data without requiring deep integration into vendor-specific toolchains.

In addition to these bridging methods, middleware patterns play a key role in runtime synchronization. Timestamp-based message tagging remains central for aligning multiple streams during replay or post-processing. This technique is particularly valuable in systems where robotic telemetry, hardware measurements, and experimental triggers must be interpreted on a unified temporal axis [45, 23].

Operational health and resilience are managed using heartbeat and watchdog signals. Nodes emit periodic "alive" messages, and if these messages are not received within a specified interval, failure recovery procedures such as component relaunch, fallback execution, or operator alerts can be triggered. This pattern helps ensure robustness in distributed setups with potential fault points across physical or software boundaries.

To further unify cross-domain systems, topic-bridging interfaces are employed. These software adapters convert foreign or proprietary data schemas into standard middleware messages, enabling visualization, analysis, or control through common tools such as ROS2 visualizers or data logging frameworks. For example, measurement data originating from INCA can be exposed as ROS2-compatible messages using custom converters that retain signal semantics and timestamps.

Collectively, these techniques allow developers to construct loosely coupled yet tightly coordinated systems that span across robotics and automotive toolchains. Such architectural flexibility is essential in modern experimentation scenarios such as hardware-in-the-loop (HIL) testing, embedded diagnostics, and autonomous vehicle development, where diverse components must operate as part of a synchronized experimental platform.

2.3 Visualization Approaches in Robotics and Measurement Domains

Visualization plays a pivotal role in the development, debugging, and evaluation of robotic and automotive systems [49]. In both domains, engineers and researchers rely on visual representations of telemetry, sensor outputs, and control signals to gain insights into system behavior. However, the tools, paradigms, and workflows used for visualization vary considerably between domains, reflecting distinct priorities, data structures, and user expectations. This section surveys the state of the art in visualization approaches across these two fields, emphasizing current capabilities, architectural characteristics, and observed limitations.

2.3.1 Visualization in Robotics Middleware

Robotic systems, particularly those built using middleware frameworks such as ROS and ROS2, produce high-frequency, real-time telemetry that includes sensor data (e.g., LiDAR, IMU, cameras), actuator commands, odometry, and system health metrics. Visualizing this data is essential for validating perception algorithms, monitoring control loops, and diagnosing runtime issues.

Topic-Centric Visualization

Tools such as `rqt_plot`, `rqt_graph`, and Foxglove Studio support visualization of individual or aggregated message topics. These tools enable users to:

- Monitor variable changes over time using line plots or histograms.
- Visualize the structure of the communication graph between nodes.
- Explore hierarchical message contents via introspection.

Visualization of Spatial and Geometric Data

In robotic applications involving navigation, localization, or manipulation, the ability to visualize spatial and geometric data is crucial for development, debugging, and system validation. Visual representations such as trajectories, pose estimations, occupancy grids, 3D point clouds, and velocity vectors enable engineers to understand the physical context in which algorithms operate. RViz, a standard tool in the ROS ecosystem, plays a central role in this process by providing a modular interface for rendering spatial information in real time [13, 23].

RViz supports the visualization of 3D point clouds obtained from LiDAR, stereo cameras, or depth sensors. It enables the rendering of coordinate frames and transformation trees, which illustrate the spatial relationships between sensors, robot parts, and reference frames. Developers can interactively observe the robot's localization within a map, monitor the progression of planned trajectories, and verify whether transformation hierarchies are correctly configured. The tool also offers support for displaying costmaps, occupancy grids, camera feeds, and depth images, making it valuable for evaluating Simultaneous Localization and Mapping (SLAM) and navigation pipelines.

Interactive markers and plugins allow users to define goals, waypoints, or manipulation targets directly within the visualization environment. These inputs can be integrated with motion planners or inverse kinematics solvers, enabling real-time testing of navigation commands and grasping strategies. Additionally, RViz can overlay semantic labels, visualize sensor field-of-view cones, and display dynamic objects such as detected humans, obstacles, or robots in multi-agent scenarios.

However, RViz remains tightly coupled to the ROS environment and is implemented as a native desktop application [27, 43]. Its architecture relies heavily on OpenGL-based rendering and does not readily support deployment in headless or web-based environments. While third-party extensions exist, RViz lacks the flexibility to function as a lightweight visualization layer in distributed or browser-based systems. This limitation has motivated the development of alternative tools and custom web visualizers that bridge the gap between ROS-native and web-native workflows.

Limitations and Ongoing Developments

While existing robotic visualization tools offer strong support for real-time introspection, spatial understanding, and geometric rendering, they are not without limita-

tions—particularly in cross-domain or large-scale deployments. One significant drawback is the lack of cross-platform user interfaces. Many widely used tools, such as RViz and rqt, are native desktop applications that depend on system-specific libraries, graphical backends, or OpenGL support. As a result, deploying them across heterogeneous environments, such as cloud servers or embedded headless systems, often requires complex installation procedures, virtual displays, or remote desktop configurations. This limits their accessibility in modern distributed or web-based development workflows.

Another key limitation is the insufficient support for historical data analysis. While tools like `rosbag` and `rosbag2` allow recording and replay of telemetry, they are not tightly integrated with interactive visual analytics platforms[40, 45]. Developers must often switch between command-line tools, external plotting utilities, and static visualizations to extract meaningful trends or patterns from past sessions. This fragmented workflow hampers post-hoc debugging, especially in experiments involving time-aligned sensor fusion, failure diagnosis, or long-duration autonomous runs.

Moreover, the user interface architecture of many traditional visualization platforms is relatively rigid. Customizing visual layouts, integrating new data types, or adapting dashboards for different robotic roles (e.g., navigation vs. manipulation) typically requires direct code changes or low-level XML configuration. This reduces flexibility, particularly for multidisciplinary teams who may lack deep expertise in the underlying frameworks but still need to interact with the system through a tailored, intuitive interface.

To address these shortcomings, recent developments are focusing on modular and extensible visualization frameworks. Web-based frontends, such as Foxglove Studio, are emerging as promising alternatives by offering platform-independent deployment, real-time streaming over WebSockets, and native support for historical playback with synchronized timelines [32, 27]. Additionally, efforts are underway to decouple visualization logic from robotics middleware through standardized message schemas and plugin-driven architectures, which can significantly improve interoperability, reduce development overhead, and foster reusable design across domains.

2.3.2 Visualization in Automotive Measurement Environments

Automotive engineers rely heavily on measurement data for calibrating ECUs, evaluating system performance, and verifying compliance with regulatory standards. Visualization in this context is tightly coupled with standardized file formats such as ASAM MDF, as well as specialized tools designed for real-time acquisition and post-processing.

Desktop-Based Measurement Analysis

The ETAS INCA and Measurement Data Analyzer (MDA) toolchain provides robust capabilities for inspecting and interpreting signal data. These tools allow engineers to generate multi-channel, time-aligned plots for various signals such as voltage, speed, and pressure. In addition, built-in annotation tools make it possible to mark key events, identify outliers, or define threshold crossings directly on the visual timeline. Overlay functionality is also supported, enabling side-by-side comparison of different measurement sessions

or calibration runs. Measurement data is often acquired in real time and then visualized either concurrently during data capture or in offline post-processing sessions. INCA places particular emphasis on precision, synchronization, and strict alignment with the associated experiment configuration [20, 1, 35].

Format-Driven Visualization Paradigm

In automotive measurement workflows, the visualization approach is largely shaped by the underlying structure of the MDF file format. MDF includes rich metadata layers that define signal units, sampling rates, acquisition sources, and event markers. Visualization tools must be capable of parsing this structure accurately to produce meaningful and context-aware plots [2]. One of the key challenges in this space is handling very large measurement files, especially those generated from high-frequency logging over extended test durations. Visualization tools must also preserve sampling integrity when performing downsampling for performance optimization, ensuring that no critical data is lost. Another challenge involves managing signals that are non-uniformly sampled or recorded at varying rates. To address these issues, many tools incorporate pre-indexing of signals and region-based rendering caches to improve responsiveness during interactive analysis.

Remote Visualization and Automation Gaps

In contrast to the robotics domain, automotive measurement tools have been slower to adopt remote and web-based visualization technologies. Although INCA does support some degree of remote automation through RCI2 or COM-based scripting interfaces, the visualization itself remains tightly bound to local, GUI-centric workflows [35, 41]. Recent research and industrial initiatives are beginning to explore more flexible solutions, including web-based dashboards for live test bench monitoring and remote experiment control through browser-based clients. Additionally, some projects are working toward exposing MDF file contents via Python or MATLAB APIs, enabling engineers to build custom visualizations and integrate signal data into broader analytics pipelines.

2.3.3 Convergence Trends in Visualization Tooling

There is a growing interest in unifying visualization approaches across robotic and automotive domains, as both fields increasingly rely on rich telemetry and measurement data to drive experimentation and validation. Shared goals among these domains include the ability to monitor telemetry data in real time through interactive interfaces, as well as the capacity to support historical playback and comparative session analysis. These capabilities are essential for understanding dynamic system behavior, diagnosing faults, and refining algorithms across iterative experiments.

In addition, there is a clear demand for modular dashboards that can be adapted to various experimental setups without requiring deep technical expertise [3, 5]. Flexibility in dashboard composition allows users to tailor visual layouts to specific applications—whether for autonomous navigation, ECU calibration, or hybrid system testing. To meet the needs of modern engineering teams, visualization solutions are also expected to function across

a range of deployment environments, including desktop systems, browser-based clients, and mobile devices.

To address these diverse requirements, emerging frameworks are increasingly adopting architectural patterns that emphasize modularity and separation of concerns. One such approach involves the use of frontend-agnostic renderers, which decouple the user interface from the underlying data processing layer. This design makes it easier to integrate visualization components into different platforms without duplicating core logic. Another trend is the use of schema-driven visualization logic, where charts and widgets are dynamically rendered based on metadata descriptions embedded within the data stream [44]. This facilitates automated generation of appropriate visual components and simplifies integration with diverse data sources. Additionally, event-driven pipelines are gaining traction, where visual updates are triggered in response to telemetry changes or measurement events, ensuring responsiveness and efficient resource usage.

Examples of these modern architectures can be seen in cloud robotics dashboards, edge analytics platforms for connected vehicles, and multi-platform visualization services deployed in testing laboratories. These systems reflect a broader shift toward unified, scalable visualization infrastructures that transcend the limitations of domain-specific tooling.

2.3.4 Identified Gaps and Research Opportunities

Despite recent advancements, several key limitations continue to hinder the effectiveness and scalability of current visualization systems. One prominent issue is the persistent fragmentation between robotics and automotive domains, which makes it difficult to create integrated dashboards that span both types of data sources [42]. Engineers are often forced to work with separate tools, each tailored to a specific ecosystem, thereby duplicating effort and introducing unnecessary complexity.

Another significant gap lies in the lack of synchronization tools for aligning multi-source telemetry streams. In cross-domain experiments, it is common to encounter datasets with differing sampling rates, time formats, or triggering mechanisms, and current visualizers rarely provide built-in mechanisms to align or correlate these data streams effectively. This results in reduced clarity during analysis and increases the likelihood of misinterpretation.

Furthermore, many existing systems offer limited configurability from the end user's perspective. Setting up a new dashboard or adapting an existing one often requires manual code changes or deep familiarity with internal configuration files. User-friendly configuration interfaces, such as graphical layout editors or schema-based form generation, remain uncommon.

Finally, a recurring architectural limitation is the tight coupling between charting components and specific data models. This restricts reuse, reduces extensibility, and makes it harder to adapt visualizations to evolving data schemas or new experimental contexts.

These challenges highlight a pressing need for the development of middleware-agnostic, highly configurable visualization systems that are easy to deploy and adapt across a range

of platforms. The following chapter will introduce a proposed architecture and interface model designed specifically to address these challenges and enable seamless cross-domain telemetry visualization.

2.4 Measurement System Integration

Modern measurement systems in automotive and embedded engineering contexts are essential for capturing high-resolution, time-synchronized data from sensors, Electronic Control Units (ECUs), and in-vehicle networks. These systems are typically composed of hardware interfaces, desktop software stacks, and standardized storage formats that together support workflows such as calibration, validation, and signal analysis [35]. This section reviews the general structure, capabilities, and limitations of such systems, particularly in the context of their integration with broader data-processing and visualization environments.

2.4.1 Structure of a Typical Measurement Workflow

A standard automotive or embedded measurement workflow involves three primary components:

1. **Signal Acquisition Hardware:** Specialized devices capable of interfacing with ECUs, buses such as CAN or LIN, and analog/digital I/O channels. Common protocols include XCP, CCP, and FlexRay.
2. **Measurement Software Stack:** Vendor-specific desktop software (e.g., INCA, CANape) used to configure experiments, define sampling rates, and map signals to variables or memory addresses [20, 22].
3. **Output Format and Storage:** Acquired data is typically logged in standardized binary formats such as the Measurement Data Format (MDF), especially MDF4, which supports high-resolution, time-stamped storage with rich metadata layers [1, 2].

These components operate in tightly coupled workflows optimized for precise data acquisition and later analysis, often within regulated development environments.

2.4.2 Control Interfaces and Automation Layers

To support automation and repeatability, modern measurement systems often expose scripting or remote interfaces. These include COM APIs, RESTful services, or proprietary command protocols that allow external tools to perform tasks such as:

- Loading measurement configurations and initializing hardware
- Triggering acquisition sessions and stopping them based on defined conditions
- Exporting recorded datasets to accessible formats for offline analysis
- Querying real-time status, error logs, and hardware health metrics

While these capabilities vary across vendors, the overall trend is toward providing external control hooks that allow software agents or test orchestration frameworks to integrate with measurement stacks without direct manual input.

2.4.3 Cross-Platform Deployment Challenges

Most measurement environments are optimized for Microsoft Windows due to legacy dependencies and hardware driver availability. Conversely, many robotics or embedded development tools run natively on Linux. This divergence presents several challenges:

Platform Incompatibility: Middleware-based telemetry tools and measurement interfaces often lack shared communication protocols.

Hardware Access Constraints: Measurement devices may rely on Windows-only drivers, limiting their use in cross-platform deployments [35].

Segregated Workflows: Engineers may be required to split their toolchains, manually synchronize data, and switch between interfaces during multi-domain experiments.

Bridging these gaps typically requires custom relays, platform-specific wrappers, or remote procedure call layers that abstract underlying system dependencies [44, 5].

2.4.4 Data Preparation and Remote Accessibility in Modern Measurement Workflows

Modern measurement systems, particularly those used in automotive and robotics contexts, often output raw data in formats like MDF4. While these formats follow well-established standards [1], they are rarely suitable for direct use in web-based applications or lightweight visualization platforms. To bridge this gap, preprocessing routines are typically employed to convert the data into more accessible representations such as CSV, JSON, or streamable objects [45]. Beyond format conversion, this process often involves normalizing signal values over time and applying techniques like interpolation or downsampling to ensure that the data remains coherent and performant when visualized in real time. Structuring the data into logically grouped entities, such as topic- or signal-based partitions, further supports its consumption by visualization components and analytics pipelines.

These preparatory steps are not merely technical niceties—they form the foundation for enabling real-time diagnostics, browser-accessible dashboards, and seamless data fusion between sources. In distributed development scenarios, where engineering teams may be spread across locations or time zones, the ability to remotely interact with measurement infrastructure has become increasingly valuable. This includes the capability to trigger recording sessions from cloud-based orchestration environments, to stream selected signals securely to remote analysts, and to observe test bench behavior in real time without being physically present [16, 17]. The demand for remote control has further accelerated interest in open APIs, standard telemetry interfaces, and automated export procedures that support annotation, flagging of anomalies, and structured archival [35, 41].

Although such capabilities are not yet universally supported across all measurement platforms, the trajectory of tooling and middleware points toward increased interoperability

and reduced friction between local hardware systems and remote consumers. Together, data transformation and remote monitoring enable a more agile, scalable, and collaborative approach to experimentation—qualities that are essential in the context of modern engineering and research workflows.

2.4.5 Known Limitations and Ongoing Challenges

Despite their capabilities, current measurement systems face several limitations:

Closed Ecosystems: Many tools lack open documentation or public APIs, limiting integration and extensibility [19, 22].

Licensing and Privilege Requirements: Access to automation or scripting interfaces may require elevated privileges or licensed hardware.

Tooling Volatility: Frequent software updates or API changes may impact stability or require significant integration maintenance [20, 18].

To address these issues, engineers often use intermediate adapters or platform abstraction layers to isolate dependencies, enable fault recovery, and maintain reproducibility across experimental sessions.

2.5 Related Platforms and Approaches

In the domains of robotics, embedded systems, and automotive testing, a wide range of tools and platforms have been developed to support telemetry visualization, signal acquisition, simulation, and calibration workflows. However, these platforms are generally optimized for specific domains, such as robotic middleware or automotive-grade measurement tools, and tend to lack native interoperability across system boundaries. This section surveys representative categories of such platforms and tools, explaining their architecture, operational principles, and practical applications.

2.5.1 Robotics-Centric Visualization Tools

Robotics visualization platforms are typically developed to interface directly with middleware frameworks like ROS and ROS2. Tools such as `rqt`, RViz, and Foxglove Studio fall into this category [12, 13, 32]. These tools support real-time monitoring and debugging of robotic telemetry by subscribing to topic-based message streams.

RViz, for example, is a 3D visualization tool that allows users to render sensor data (e.g., point clouds from LiDAR, RGB or depth images from cameras, odometry vectors) within a virtual environment. RViz works by subscribing to ROS topics, extracting structured message data, and mapping it to interactive graphical elements using plug-in modules [13].

rqt provides a more GUI-oriented experience for telemetry monitoring. It offers pre-built and extensible plugins for plotting numerical topics, echoing message contents, tuning parameters, or visualizing diagnostics. Developers can load specific plugins based on the

topic types they are working with and view synchronized signals such as joint angles, IMU readings, or controller states [12, 43].

Foxglove Studio is a modern visualizer that supports ROS1, ROS2, and custom telemetry data streams via the WebSocket-based Foxglove protocol. It allows users to create dashboards with modular charting panels, 3D scenes, and map overlays. It offers integration with rosbag playback, topic introspection, and time-aligned data rendering, making it suitable for collaborative debugging or post-experiment analysis [32].

While these tools are highly modular and extensible within robotics development environments, they have limitations. Most are optimized for Linux platforms, and few offer built-in support for integrating non-ROS telemetry sources or automotive-grade measurement formats like MDF. Moreover, their workflows often assume a robotics-centric development lifecycle and may not align with the structured, hardware-calibrated workflows used in embedded system validation.

2.5.2 Measurement-Oriented Automation Suites

Measurement and calibration suites are widely used in automotive and embedded system domains for acquiring high-resolution signals from electronic control units (ECUs), buses, and sensors. Examples include **ETAS INCA**, **Vector CANape**, and National Instruments-based systems [20, 22, 34].

ETAS INCA is a Windows-based desktop application that allows engineers to define measurement tasks using configuration files, interface with devices over CAN, LIN, or XCP, and record time-aligned datasets in ASAM MDF (`.mf4`) format [20, 1]. INCA supports real-time visualization of scalar and vector signals, parameter calibration, and ECU flash programming. Post-processing is handled via its companion tool, MDA (Measurement Data Analyzer), which offers time-series plots, annotations, and export functions [20].

Vector CANape performs similar functions and adds enhanced bus decoding, CCP/XCP support, and integration with vehicle diagnostics. Its experiment editor allows the design of test sequences with conditional logic, timers, and parameter sweeps. CANape also supports automation via COM scripting or integration with third-party control systems [22, 41, 35].

National Instruments (NI) solutions, such as LabVIEW and VeriStand, use graphical programming and modular hardware interfaces to design custom test benches. These platforms support analog/digital I/O, synchronized acquisition, and stimulus-response analysis. Signal data is typically acquired using PXI-based chassis and can be exported in TDMS or MDF formats [34].

These suites are built around GUI-centric workflows and assume direct interaction with physical hardware. Automation interfaces such as RCI2 (for INCA) or COM bindings allow limited external control but require Windows environments and vendor-specific runtime libraries. While powerful for automotive applications, these tools are rarely used outside their specialized domains and lack interoperability with modern middleware or web-based systems.

2.5.3 Hybrid and Simulation-Based Development Environments

Simulation-driven platforms have gained significant traction in recent years for testing and validating embedded systems in virtual environments. Examples include **IPG CarMaker**, **dSPACE AutomationDesk**, and **NI VeriStand** [21, 15, 34]. These environments are particularly useful for hardware-in-the-loop (HIL) testing and model-based development.

IPG CarMaker allows engineers to simulate entire vehicle dynamics, traffic scenarios, and sensor inputs such as camera and radar data. The platform integrates with real-time hardware and provides configurable test cases for vehicle ECUs under controlled conditions. Telemetry is generated synthetically from the simulation engine and logged alongside model states and stimuli [21].

dSPACE AutomationDesk offers a graphical interface to build test sequences that interact with dSPACE hardware, such as MicroAutoBox or SCALEXIO systems. It enables automated stimulus generation, signal monitoring, and pass/fail evaluation based on test assertions. Results can be stored and exported in standardized formats for documentation [15].

NI VeriStand is a real-time test framework that manages stimulus profiles, signal acquisition, and closed-loop testing. It provides out-of-the-box support for I/O modules and integrates with Simulink, LabVIEW, and third-party software. Engineers use VeriStand to validate control algorithms, test diagnostics, and tune control loops in embedded software [34].

These environments are highly sophisticated but are primarily focused on simulation or HIL use cases. They often support integration with measurement hardware, but require extensive configuration and licensing. Moreover, their telemetry interfaces are typically not compatible with middleware-driven robotic systems like ROS2, and custom connectors are needed for integration. Their visualization components are also tightly coupled to the testbench or simulation scenario, rather than general-purpose telemetry inspection.

2.5.4 Summary of Observed Trends

The reviewed categories of tools reveal the following patterns:

Robotics tools prioritize real-time topic monitoring, message introspection, and modular GUI plugins. However, they are limited in hardware integration and cross-platform orchestration [12, 13, 32, 43].

Measurement suites provide high-fidelity data logging and automation within calibration workflows, but are constrained by proprietary formats, desktop-centric designs, and Windows dependency [20, 22, 35, 1].

Simulation environments support repeatable, scenario-based testing but lack native telemetry ingestion from external robotic or sensor platforms [21, 15, 34].

There remains a gap in tooling that supports seamless integration between robotic middleware telemetry and automotive-grade measurement or simulation workflows. As the boundaries between domains continue to blur—especially in applications such as

autonomous vehicles, intelligent test benches, and cyber-physical systems—there is a growing need for modular, interoperable, and visualization-capable systems that can unify telemetry from diverse origins [44, 5].

2.6 Requirement Summary and Justification

Based on the gaps observed in existing robotic and measurement toolchains, this section presents the core requirements that will shape the design of the proposed system. Unlike generic visualizers or proprietary measurement environments, the planned platform will focus on interoperability, modularity, and future extensibility. Rather than listing an exhaustive feature set, this requirement summary outlines six carefully selected capabilities that emerged as essential through comparative analysis.

2.6.1 Requirement R1: Real-Time Visualization of Robotic Telemetry

The platform will need to support the ingestion and real-time display of structured telemetry generated by robots or embedded systems. These data streams may include inertial measurements, actuator commands, or environmental sensor outputs, transmitted over middleware channels. The visualization must be responsive, time-aligned, and modular—capable of adapting to different data formats and topic structures.

Justification: Existing robotic dashboards support live data only in isolated workflows (see Section 2.5.1). A hybrid system must visualize such data alongside measurement values, with minimal latency and high configurability.

2.6.2 Requirement R2: Historical Playback of Structured Telemetry

The system will include capabilities for loading previously recorded data files and replaying them as if they were live. This functionality will be essential for debugging, validation, and performance comparisons across test sessions. The replay engine will be decoupled from the live stream architecture, but will feed the same frontend visualization modules for consistency.

Justification: Historical playback is supported in some middleware tools but lacks synchronization with external measurement data (see Section 2.5.1). The proposed system will enable aligned review across domains.

2.6.3 Requirement R3: Integration with Measurement Devices

Measurement hardware, such as embedded acquisition units, will continue to play a central role in calibration and validation workflows. The platform must provide mechanisms to access, control, and visualize signals from these devices, either in real time or through processed files. Integration will be facilitated via backend abstractions that wrap native interfaces and expose simplified controls.

Justification: As noted in Section 2.5.2 and Section 2.5.3, measurement systems like INCA and CANape provide reliable ECU signal acquisition but are tightly coupled to Windows-

based GUIs and vendor APIs. Robotic tools currently cannot access these measurement channels, creating a disconnect during hybrid testing. Integration via backend abstraction is essential to bridge the two ecosystems.

2.6.4 Requirement R4: Unified Web-Based User Interface

A single web-accessible frontend will provide user interaction across all system components. This interface will support live monitoring, signal replay, hardware control, and visualization configuration. Users should not require domain-specific tools or installations to interact with the platform, and the interface must remain responsive across devices.

Justification: Existing systems rely heavily on platform-specific GUIs. By contrast, a web-native interface ensures wider accessibility and enables collaborative workflows across remote teams (see Section 2.3.2 and Section 2.3.3).

2.6.5 Requirement R5: Modular Deployment Across Operating Systems

The system architecture must be deployable across all platforms, acknowledging that robotic middleware typically operates on Linux, while measurement tools are often restricted to Windows environments. Containerization, remote orchestration, and abstraction layers will be leveraged to ensure clean separation and coordination between services.

Justification: Multi-platform execution remains a key limitation of current tools (see Section 2.1.3 and Section 2.4.3). The proposed architecture will address this with backend modularization and frontend unification.

2.6.6 Requirement R6: Extensible Charting Framework with Domain-Agnostic Logic

Visualization logic must be decoupled from specific data domains or hardware types. The charting modules will be reusable across signal types and sources, accepting configurable schemas for axes, units, and labels. This will enable reuse across robotics, embedded systems, and future telemetry formats without reimplementation.

Justification: Many tools hard-code visualizations for particular signals or middleware types (see Section 2.3.3 and Section 2.3.4). A schema-driven approach will ensure flexibility and long-term sustainability.

2.6.7 Requirements Table Overview

Table 2.1 summarizes the six refined system requirements, along with their targeted domain(s) and the expected method of fulfillment within the proposed platform architecture.

Table 2.1: Refined system requirements and coverage

ID	Description	Target Domain	Fulfillment Strategy
R1	Visualize live robotic telemetry	Robotics	Topic-based data ingestion with live streaming chart modules
R2	Replay historical structured data	Robotics / Mixed	Backend parser for recorded files with timestamp-aligned chart output
R3	Integrate measurement device signals	Measurement	Hardware wrapper modules interfacing with acquisition APIs
R4	Web-based frontend for interaction	All	Browser-accessible dashboard with unified controls
R5	Modular multi-platform deployment	All	Containerized backend services and cross-platform orchestration
R6	Reusable charting framework	All	Schema-driven frontend logic with modular visual components

These requirements will serve as a foundation for the system design discussed in Chapter 3, where each architectural decision will directly address one or more of the functional goals listed above.

2.7 Comparative Analysis and Outlook

To contextualize existing solutions within the broader landscape of robotic and measurement tools, this section presents a focused comparison between a selected set of representative systems. These include one open-source middleware visualizer, one proprietary measurement suite, and one hybrid simulation environment. Each system is evaluated based on the core requirements defined in Section 2.6, highlighting their respective capabilities and limitations.

2.7.1 Selected Approaches for Comparison

The selected tools represent diverse technological paradigms and usage domains:

- **A1: Open-Source Middleware Dashboard** — Tools such as Foxglove Studio and RQT, designed to visualize real-time telemetry from robotics middleware frameworks like ROS or ROS2.
- **A2: Measurement System Interface** — Environments such as ETAS INCA or Vector CANape, focused on hardware-level signal acquisition and calibration in automotive

applications.

- **A3: Hybrid Simulation Suite** — Platforms like IPG CarMaker or dSPACE AutomationDesk, used to simulate test scenarios and visualize synthetic telemetry in HIL/SIL workflows.

2.7.2 Evaluation Matrix

Each platform is assessed across six requirements (R1–R6) on a 5-point ordinal scale. Higher scores indicate more comprehensive functionality in the context of each requirement:

- **1:** No feature support
- **2:** Basic or initial functionality
- **3:** Moderate support with partial capabilities
- **4:** Advanced support for most features
- **5:** Full-featured implementation

Table 2.2: Functional comparison of selected robotics, measurement, and simulation tools

Requirement	**A1**	**A2**	**A3**
R1: Real-time robotic telemetry	5	1	3
R2: Historical telemetry replay	3	1	4
R3: Measurement device integration	1	5	3
R4: Unified web-based user interface	2	1	2
R5: Modular cross-platform deployment	1	1	4
R6: Reusable charting framework	3	1	1

2.7.3 Analytical Insights

The comparison in Table 2.2 reveals how existing tools reflect the constraints and assumptions of the domains in which they were developed. Although each system demonstrates maturity within its specialized scope, the comparative analysis shows a distinct lack of convergence between robotic telemetry, embedded measurement, and simulation-based validation environments.

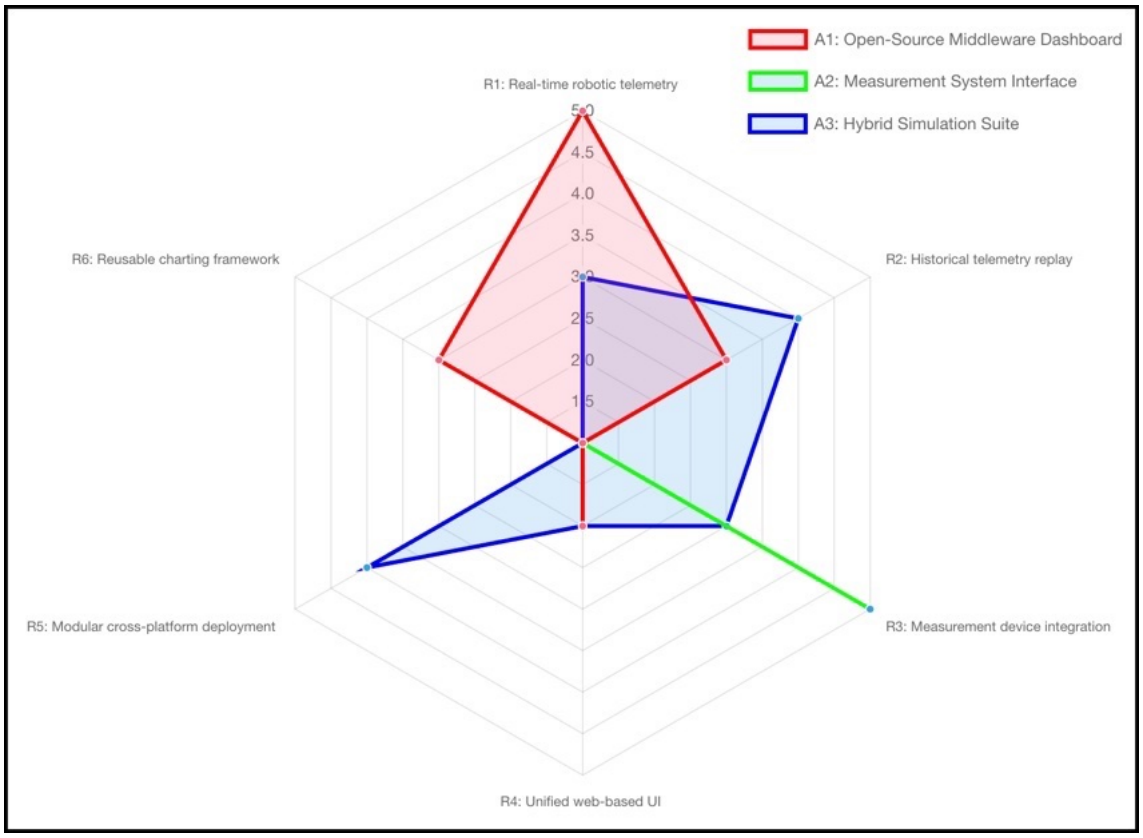

Figure 2.2: Radar chart comparing selected tools across the six core requirements

Observation 1: Open-source middleware dashboards (A1), such as RQT and Foxglove Studio, excel in streaming telemetry visualization (R1) due to their tight integration with robotic middleware. However, they are limited in their capacity to ingest hardware-based measurements (R3), operate across mixed operating systems (R5), or support session orchestration involving ECU calibration workflows. Their middleware-centric architecture assumes homogeneous environments and does not easily extend to closed-loop experimental settings.

Observation 2: Measurement interfaces (A2), such as ETAS INCA or Vector CANape, are optimized for high-precision signal acquisition and calibration (R3). Yet, their architectural rigidity and Windows-only deployment models severely limit cross-platform interoperability (R5) and user interface innovation (R4). These systems rely heavily on local desktop applications, static experiment definitions, and proprietary APIs, which hinders integration with distributed robotic systems or real-time collaborative workflows.

Observation 3: Simulation suites (A3), including platforms like IPG CarMaker or dSPACE AutomationDesk, support configurable scenario playback (R2) and provide modular deployment capabilities (R5) through HIL or SIL architectures. However, these tools often operate in isolation from live telemetry sources and typically lack reusable, domain-agnostic charting frameworks (R6). Visualization components are designed for specific

test cases and do not accommodate dynamically structured external telemetry, limiting their adaptability to broader use cases.

These insights emphasize that while each system brings domain-specific strengths, none currently offers seamless integration across robotic, measurement, and simulation workflows. The gap in tooling reflects not just architectural incompatibilities but also divergent assumptions about user interaction, data ownership, and runtime flexibility.

2.7.4 Outlook on Toolchain Convergence

With the increasing interdependence of robotics, embedded systems, and real-time telemetry in research and industrial settings, a strong trend toward toolchain convergence is emerging. Robotic platforms are now routinely embedded with ECUs, sensors, and real-world data acquisition components, while automotive systems increasingly leverage robotic middleware for autonomy and perception. This evolution demands toolchains that span traditional domain boundaries.

The following technical directions are projected to shape future platform development:

Cross-domain integration: Future systems must support seamless ingestion and correlation of diverse telemetry sources—including ROS2 topics, MDF logs, CAN bus messages, and synthetic simulation outputs—on a shared temporal axis. This will enable coherent validation of system behavior under real and simulated conditions [38, 1, 20].

Web-based, device-agnostic interfaces: A shift toward browser-native user interface is expected to democratize access to telemetry dashboards and calibration tools. Such interfaces should operate across devices and support responsive layouts, real-time updates, and user-driven customization without platform lock-in.

Modular backend orchestration: Middleware solutions will need to support distributed, decoupled architectures that allow dynamic integration of new data sources, visualization modules, and automation workflows. This includes runtime discovery of telemetry publishers, brokered communication, and service-based control layers [44, 5].

Temporal synchronization and traceability: As data from multiple systems is fused, synchronization becomes critical. Toolchains must support timestamp alignment, trigger signals, and causal correlation of events across telemetry, measurement, and simulation domains [45].

Beyond these core trends, emerging areas such as **cloud-native measurement pipelines** and **AI-assisted data fusion** are set to further accelerate toolchain convergence. Cloud-hosted environments will increasingly facilitate distributed telemetry processing, allowing developers to run large-scale simulations and real-time experiments with minimal local hardware requirements. This shift promises enhanced scalability, where computationally intensive tasks like signal correlation, anomaly detection, or predictive analysis can be offloaded to elastic infrastructure, thereby reducing system complexity at the edge.

Moreover, **semantic data enrichment** is gaining importance as a key enabler for unified telemetry and measurement frameworks. Instead of treating sensor signals, bus messages,

or calibration parameters as isolated values, future platforms will embed semantic metadata—such as unit definitions, physical relationships, and context tags—directly within the data streams. This will simplify cross-domain analytics, facilitate automated integrity checks, and improve interoperability between engineering tools.

The adoption of **open data exchange standards** is also poised to play a pivotal role in overcoming the fragmentation that currently hampers integrated workflows. By aligning on community-driven specifications for telemetry formats, service interfaces, and synchronization protocols, developers and researchers will be better positioned to compose toolchains from heterogeneous components without needing extensive bespoke adapters or fragile workarounds.

Finally, as hybrid systems grow in complexity, the importance of **user-centric design principles** in toolchain development cannot be overstated. Future solutions will need to balance technical sophistication with accessibility, providing engineers with intuitive, low-friction interfaces for configuring experiments, inspecting telemetry, and interpreting results. This human-centered approach is essential to ensure that powerful technical capabilities translate into practical engineering value across disciplines.

Despite incremental advances in some of these directions, current solutions remain siloed. The analysis confirms a pressing need for platform-agnostic solutions that unify engineering tasks across robotics, automotive, and cyber-physical systems. Such platforms must emphasize openness, extensibility, and interactivity, enabling collaborative experimentation without compromising precision [49].

The gaps and trends identified in this section motivate the design principles explored in Chapter 3, which will focus on modular system architecture, telemetry abstraction, and visualization unification. These design directions aim to address the limitations uncovered in current approaches and propose an adaptable foundation for future hybrid systems.

Chapter 3

Concept

This chapter outlines the conceptual foundation of the proposed telemetry and measurement platform. The focus is on defining the system's core objectives, high-level structure, communication patterns, and functional components. Unlike the implementation details discussed in Chapter 4, the content here presents a platform-agnostic view that describes how the system will address integration challenges across robotics middleware, automotive measurement tools, and visualization interfaces. Each section aims to translate functional requirements into modular design principles that guide the subsequent development process.

3.1 System Overview

This section introduces the foundational architecture of the proposed telemetry and measurement platform. The system is conceptualized to support seamless integration across heterogeneous domains, namely robotics middleware and automotive measurement environments. By abstracting hardware and protocol differences through layered middleware and modular interfaces, the platform aims to unify real-time telemetry visualization, session control, and data analysis into a single extensible framework [42]. The architecture outlined here will serve as a baseline for more detailed components in subsequent sections.

3.1.1 Objectives of the Architecture

The conceptual system architecture is guided by a set of high-level objectives derived from the requirements defined in Chapter 2. These objectives include:

Cross-Domain Compatibility: Enable integration of telemetry data from robotic frameworks such as ROS2 and measurement tools based on automotive standards [38, 20].

Separation of Concerns: Isolate ingestion, processing, and visualization logic across distinct system layers to promote maintainability and reusability [44].

Real-Time and Historical Support: Provide consistent interfaces for both live data streaming and playback of recorded sessions without requiring interface reconfiguration

[40, 45].

Platform-Agnostic Deployment: Support execution across all platforms, including hybrid lab environments with constrained deployment topologies [5].

Scalability and Modularity: Facilitate the addition of new telemetry sources, visualization modules, or backend automation logic without disrupting existing functionality [44].

These goals are intended to address gaps identified in current tooling, where domain-specific systems often operate in isolation, leading to fragmented workflows and limited interoperability.

3.1.2 Conceptual Platform Interaction

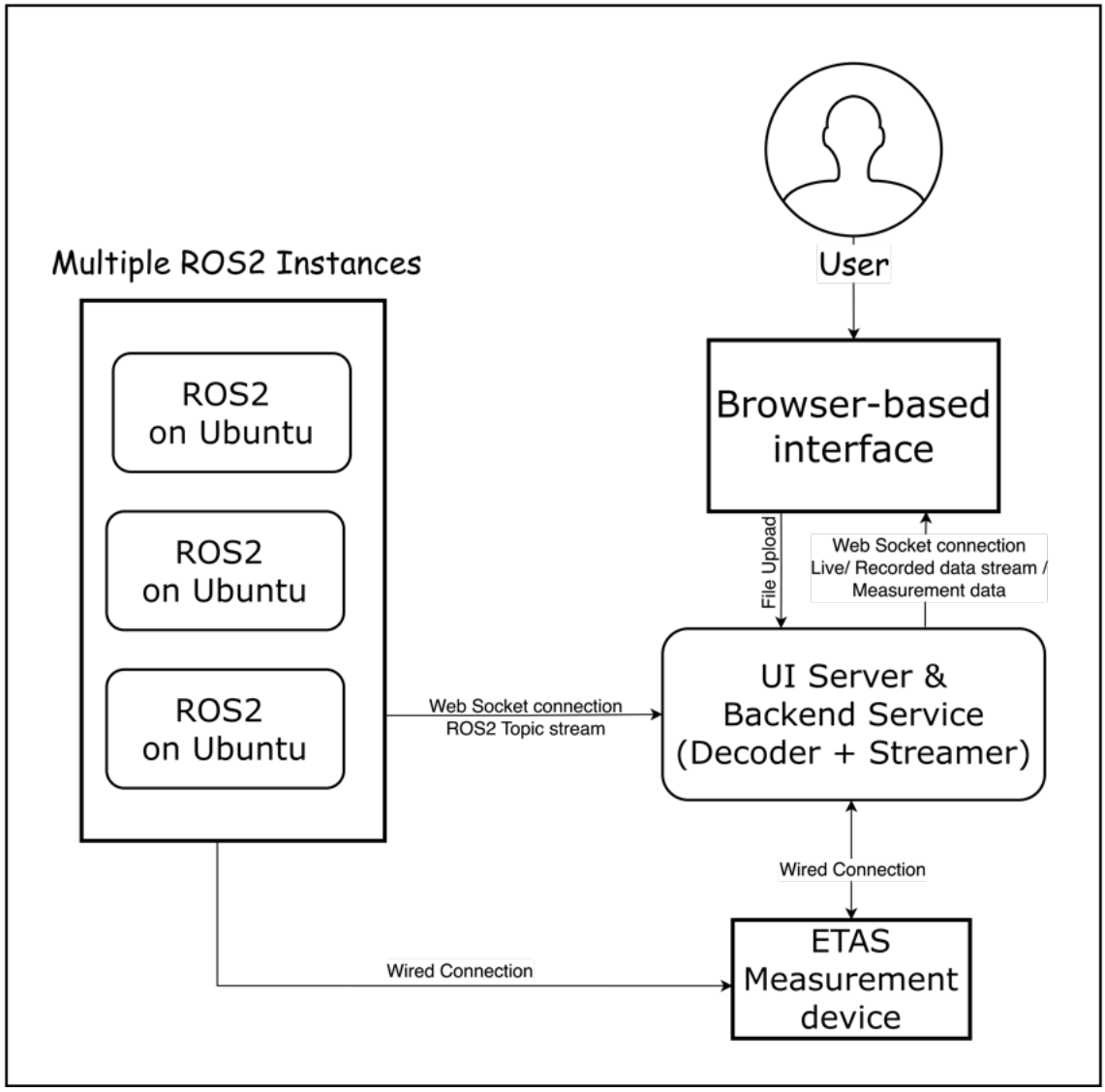

Figure 3.1: Conceptual Platform Interaction Diagram

Figure 3.1 illustrates the conceptual interaction model of the telemetry and measurement platform. The architecture centralizes user interaction through a unified dashboard interface that communicates with three major backend sources: ROS2 live telemetry, historical playback services, and automotive measurement controllers. All interactions occur over WebSocket or Hypertext Transfer Protocol (HTTP) interfaces, ensuring low-latency, event-driven connectivity across domain boundaries.

Frontend Dashboard: The user interface, implemented as a browser-based application, serves as the central control and visualization hub. It fulfills two primary functions. First, it connects to all telemetry sources—including live ROS2 topics, historical replay services, and measurement outputs—via WebSocket channels. These data streams are rendered in real time through modular visual components, supporting both immediate diagnostics and post-run analysis within a unified dashboard environment.

Second, the frontend provides an interactive control interface that allows users to configure and manage measurement workflows. Through graphical elements such as buttons, selectors, and dropdown menus, users can issue commands to start or stop measurements, add or remove signals, and load INCA configurations. These commands are transmitted to the backend over HTTP, where they are handled by a control interface that invokes the INCA runtime through a custom-developed DLL using RCI2 scripting and COM APIs [20, 19, 5].

This dual role enables the frontend to act not only as a passive visualizer but also as an active orchestrator of the complete telemetry and measurement pipeline.

Live ROS2 Data: The live telemetry pipeline begins on the robot, which hosts a Linux-based CPU running the ROS2 middleware stack [38]. Sensor data—such as IMU readings, LiDAR scans, odometry, and actuator states—is continuously collected and published to ROS2 topics using the native DDS-based publish-subscribe mechanism. These topics are made accessible over WebSocket using `rosbridge_server`, sometimes referred to as `ros2_webbridge` in ROS2 contexts [9]. This service acts as a protocol adapter, converting DDS traffic into JSON-encoded messages suitable for web clients.

On the frontend, the browser-based application connects to the ROS2 system using `roslibjs` [46], enabling direct subscription to published topics. Messages are received in real time over WebSocket, parsed, and routed to rendering components for live visualization. This architecture eliminates the need for a backend intermediary in the live data path, offering a low-latency, fully browser-native telemetry experience.

Historical Replay Engine: The platform includes a backend service that supports offline telemetry visualization by decoding data files recorded via `rosbag2` [40]. This replay engine parses stored data using ROS2-compatible topic schemas and re-emits the decoded messages over WebSocket to the frontend [45]. Playback timing is simulated using synthetic timestamps that replicate the original message timing from the recorded session.

From the frontend's perspective, the messages received from the historical replay engine follow the same structure and topic layout as those from live ROS2 sources. This consistency allows all visualization components to be reused across both live and replay modes without modification. It ensures a seamless user experience for real-time diagnostics

and retrospective analysis alike.

Measurement Agent: The Windows-based measurement environment consists of ETAS INCA software interfacing with measurement hardware for real-time ECU-level data acquisition [20]. Users initiate measurement workflows via the frontend dashboard. These control commands are forwarded to a backend service, which internally invokes a custom-developed C++ `.dll` to interact with INCA. This dynamic link library leverages RCI2 scripting and COM APIs to automate measurement execution.

Once measurement begins, INCA receives streaming data from the measurement hardware using its native protocols. The backend, directly connected to both measurement hardware and INCA, retrieves selected signal values and streams them to the frontend over WebSocket. This enables real-time inspection of ECU measurements alongside robotic telemetry and historical playback data within the same unified interface.

The agent architecture abstracts away proprietary protocols, ensuring that measurement workflows remain accessible and automatable through standardized frontend interactions [5].

Unified Communication Backbone: All system components interact via JSON-encoded WebSocket messages. This design abstracts protocol differences (e.g., DDS, COM, SQLite) and enables the frontend to uniformly handle telemetry from diverse domains. Messages are enriched with metadata, including timestamps, source identifiers, and stream labels, to support modular rendering, structured logging, and robust traceability [5].

This interaction model facilitates concurrent access to all data sources, enabling users to perform live diagnostics, historical reviews, and measurement-driven analysis within a single session. The unified dashboard acts as a bridging layer, decoupling domain-specific backend implementations while preserving responsiveness and extensibility for cross-domain experimentation.

3.1.3 Domain Interoperability Goals

A key concept driving the system design is interoperability across heterogeneous domains. Robotics systems and automotive test environments differ significantly in runtime architectures, supported protocols, and data serialization methods. The conceptual architecture addresses these differences through:

Unified Abstraction Interfaces: Data from different domains is normalized into a shared telemetry model with consistent fields for timestamps, signal names, values, and units.

Cross-Platform Messaging Bridges: Middleware proxies facilitate bi-directional data exchange between Linux-based ROS2 systems and Windows-based measurement stacks, leveraging platform-neutral transports like WebSocket or TCP [44, 6].

Time Synchronization Mechanisms: Common timebases or reference markers are used to align incoming data streams, enabling coherent multi-domain visualization [5].

These interoperability strategies allow the platform to bridge the conceptual and technical gap between robotic telemetry and ECU-level measurement data, fostering integrated

experimentation, debugging, and analysis.

3.2 Data Ingestion and Middleware Coordination

This section outlines the internal communication and data coordination architecture of the system, focusing on how telemetry flows between robotic sources, measurement hardware, middleware services, and visualization components. It merges two previously separate concerns—(1) telemetry ingestion from live and historical sources, and (2) middleware mechanisms that decouple producers and consumers through structured communication models. The result is a unified framework supporting modular, scalable, and cross-platform telemetry management across robotics and automotive systems [38, 5, 42, 3].

3.2.1 Live Telemetry and Historical Replay

Live ROS2 Data Streams: The live telemetry ingestion path handles the continuous stream of data originating from a ROS2-enabled robot running on a Linux system [38]. The robot publishes sensor and control data through the DDS-based publish-subscribe model intrinsic to ROS2 [27, 28]. Topics may include IMU data, LiDAR scans, joint states, odometry, actuator commands, and custom debugging messages [33].

Unlike traditional systems with backend ingestion services, this platform connects the frontend directly to the ROS2 network using `rosbridge_server`, which exposes ROS2 topics over WebSocket. In the Angular application, a telemetry listener subsystem is implemented using `roslibjs` [46], allowing direct topic subscription from the browser. This approach eliminates the need for a backend intermediary in the live data path.

Subscriptions are managed using topic-level Quality of Service (QoS) profiles configured in ROS2 and 'rosbridge', including reliability, durability, and latency constraints [28, 27]. Messages are received in real time, buffered in browser memory, and normalized using schema definitions handled by `roslibjs`. Optional client-side processing includes topic filtering, metadata augmentation, and unit conversion before rendering.

The decoded messages are visualized using modular components in the Angular frontend, with charting handled by libraries such as Apache ECharts and Chart.js [10, 7]. Users can dynamically configure dashboards, and interact with real-time telemetry without modifying the underlying communication flow.

This browser-based architecture simplifies deployment and enhances modularity, while also enabling multiple users to observe the same ROS2 data stream in parallel through WebSocket connections. It decouples frontend development from ROS2 internal mechanics, enabling easier integration of robotics telemetry into web-native dashboards.

Historical ROS2 Playback: Complementing the live ingestion path, the system supports historical telemetry replay using '.db3' files recorded via `rosbag2` [40]. Instead of relying on native ROS2 playback tools, a custom Node.js backend service parses these files using topic schema definitions and emits the decoded messages to the frontend via WebSocket [45]. This replay module replicates the topic-based messaging structure of live telemetry while allowing for fine-grained control over data flow.

The Node.js application simulates the original message timing using a synthetic clock, ensuring that replayed data preserves the temporal characteristics of the original session. Key playback capabilities include seek functionality, playback speed control, and selective topic filtering. Unlike live ROS2 ingestion, which uses DDS and ROS-native tools, this historical pipeline is fully decoupled and facilitates standalone analysis of recorded datasets.

From the frontend's perspective, both live and historical telemetry arrive through a common interface and use identical visualization modules. This design choice ensures a unified user experience while allowing backend flexibility to handle the distinct technical requirements of real-time streaming and offline replay.

3.2.2 Measurement Control Integration

In addition to robotic telemetry, the system integrates with ETAS automotive-grade measurement devices, particularly the ETK series, managed through the INCA environment [19, 20, 35, 41]. These devices are accessed via command-and-control interfaces that differ from ROS2's topic-based model. Measurement sessions are orchestrated via COM APIs or RCI2 scripting interfaces, which enable programmatic configuration and control of signal acquisition workflows.

A dedicated Windows-based automation agent is deployed alongside INCA. This agent exposes a structured API, capable of receiving control commands such as:

- Load a predefined INCA project or measurement configuration
- Start or stop an active recording session
- Export captured measurement signals to disk
- Query hardware and session status (e.g., idle, measuring, error)

Commands are transmitted from the middleware to the INCA agent via WebSocket or HTTP interfaces. All exchanges include acknowledgments, retries, and audit logging for traceability. Internally, commands are translated into RCI2 scripts or COM method calls, which interact directly with the INCA runtime.

Measurement results are saved in ASAM-compliant MDF format (`.mf4`) [1], enabling downstream analysis in industry-standard tools. The frontend may optionally receive live status updates, duration metadata, or session identifiers, but ETAS data is not time-synchronized with ROS2 telemetry. Instead, a parallel-panel layout is used in the user interface to allow side-by-side inspection with approximate temporal alignment through semantic markers.

3.2.3 Messaging Models and QoS Configuration

The middleware layer supports both topic-based telemetry and service-based control through standardized communication models [5, 6, 3].

Topic-Based Messaging: ROS2 live and replay data are streamed using a publish-subscribe model. Messages are published to named topics with schema-bound message structures, and subscribers register interest without knowing the identity or number of publishers. Each message carries metadata, including timestamps, UUIDs, and source tags [30]. Topic-level QoS parameters include:

1. **Reliability:** Reliable vs. best-effort delivery
2. **Durability:** Transient-local or persistent for new subscribers
3. **Latency Budget:** Maximum tolerable delivery time
4. **Liveliness:** Heartbeats to detect publisher activity

These QoS settings are tunable per topic and are essential for balancing latency, bandwidth, and reliability across diverse telemetry sources.

Service-Based Control Interfaces: Measurement control, status polling, and session configuration follow a request-response model. These are implemented as REST APIs, gRPC services, or WebSocket handlers. Each service endpoint follows a schema-defined contract with validated parameters, predictable return types, and structured error codes. Asynchronous service calls are supported with acknowledgment and callback options.

Event Handling and QoS Observability: An internal observer module continuously monitors middleware behavior to detect message drops, latency spikes, schema mismatches, or unexpected load patterns. Alerts are raised for QoS violations, and telemetry channel performance can be logged for post-analysis. In some cases, auto-tuning mechanisms may adapt topic QoS settings based on network or system conditions. Users may also configure QoS preferences at runtime through the frontend.

3.2.4 Cross-Platform Middleware Relays

Due to the architectural separation between Linux-based ROS2 nodes and Windows-based INCA measurement systems, the platform includes cross-platform middleware relays that bridge protocols and transport layers. These relays enable bidirectional communication and ensure consistent message formats and delivery semantics across operating system boundaries.

Relay components consist of:

1. **Protocol Adapters:** Interfaces that translate between DDS, WebSocket, COM, and gRPC protocols
2. **Transport Buffers:** In-memory message queues that preserve delivery order and timestamps
3. **Error Isolation Modules:** Sandboxing features that prevent failures in one relay path from affecting other system components

In constrained environments (e.g., where DDS is unavailable on Windows or COM is restricted on Linux), fallbacks include SQLite-based IPC, shared memory buffers, or

periodic polling mechanisms. These ensure that baseline functionality is retained even under degraded conditions.

This relay infrastructure allows seamless integration between frontend dashboards, ROS2 live data, '.db3' playback, and measurement control backends. Future extensions may support additional middleware standards such as MQTT or OPC UA, expanding the platform's applicability to IoT, industrial, or cloud-native domains.

3.3 Visualization and Interaction Architecture

This section presents the conceptual design of the visualization and interaction layer, which serves as the user-facing entry point to the telemetry and measurement platform. The visualization subsystem is responsible for rendering structured telemetry data—whether ingested live from ROS2 nodes, replayed from '.db3' files, or reported by measurement devices—into human-readable, interactive dashboards. Furthermore, it provides configuration interfaces, playback controls, real-time alerts, and customization options necessary for debugging, experimentation, and validation workflows. The design emphasizes modularity, consistency, and extensibility across both robotics and automotive domains [10, 7, 24, 49, 3].

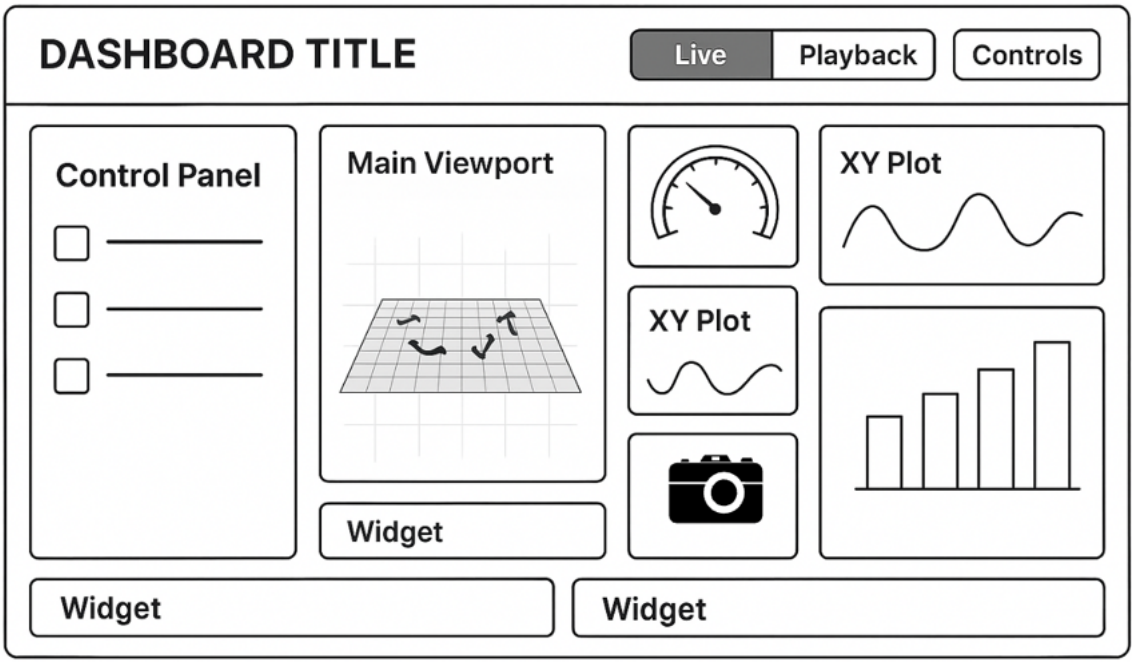

Figure 3.2: Schematic Layout of Visualization Dashboard Architecture

3.3.1 Dashboard Composition and Data Binding

At the core of the visualization layer lies the dashboard composition framework. Dashboards are composed of charting widgets, controls, and layout components arranged into

resizable and themable panels. Each panel may contain one or more charts that are linked to specific telemetry topics or measurement streams.

Data binding is achieved through a schema-driven interface. Every chart module declares a binding specification that includes:

Topic or Signal Source: Identifier for a ROS2 topic or measurement signal.

Data Path: JSON or binary field path for extracting the relevant value (e.g., `msg.pose.position.x`).

Axes Configuration: Time-axis window size, y-axis bounds, and value units.

Update Frequency: Desired refresh rate in frames per second (FPS) or samples per window.

This approach allows charts to remain agnostic to underlying transport protocols. Whether the data originates from a DDS message, WebSocket payload, or replay buffer, the frontend renders it using a uniform logic layer.

The dashboard manager maintains a centralized registry of active charts, bindings, and buffers. This enables runtime updates such as adding new charts, switching sources, or duplicating dashboards for comparative views. The architecture also supports layout persistence through session files (e.g., JSON configuration snapshots) that encode dashboard structure, chart bindings, and user interface preferences [3].

3.3.2 Live vs Playback Rendering

The visualization engine operates in two distinct modes: **Live Mode** and **Playback Mode**, both designed to reuse the same user interface components but with different backend coordination.

In **Live Mode**, the frontend subscribes to real-time data streams from ROS2 nodes using WebSocket connections provided by `rosbridge_server`. The Angular frontend, via `roslibjs`, decodes topic messages directly in the browser. Each data point arrives with a native timestamp and is appended to in-memory buffers within the dashboard. Charts operate in scroll mode or fixed time windows, depending on user configuration. High-frequency data, such as IMU or LiDAR, is rendered using downsampling strategies like min–max decimation to reduce rendering overhead [38, 46, 27].

Playback Mode handles historical telemetry visualization by ingesting '.db3' files recorded via `rosbag2`, which are processed through a dedicated Node.js backend replay engine [40]. The backend parses ROS2 message data for each topic, reconstructs timestamped message sequences, and emits them via WebSocket to the frontend using a structure identical to live topics. Synthetic timestamps simulate original message timing. The frontend renders this data using the same visualization components as in Live Mode [45].

Although interactive playback controls such as pausing, scrubbing, or seeking are not implemented, selective topic display and dashboard layout configuration allow users to focus on relevant telemetry streams. This enables differential debugging by comparing historical data to live telemetry in adjacent panels.

3.3.3 Modularity and Component Extensibility

The visualization architecture is implemented using a component-based frontend framework (Angular), and all visual elements conform to a standardized interface specification [24]. Each chart module is self-contained, subscribes to events via a message bus, and renders data using third-party libraries like ECharts or Chart.js [10, 7, 42].

New visualizations can be introduced by:

1. Implementing a base component class with required data binding and rendering hooks.
2. Registering the chart in the dashboard schema registry.
3. Providing optional controls (e.g., axis selectors, filters) via a plug-in configuration panel.

Examples of custom extensions planned:

3D Grid and Point Overlay: For displaying LiDAR scans or path traces within a ROS2 coordinate frame.

Bar Charts: Such as battery cell voltages, shown using standardized and color-coded groupings.

Ultrasonic Distance Panels: Displaying proximity measurements in a front-facing layout with dynamic scaling.

Dial Gauges: For variables like balance angle, temperature, or torque readings.

Time-Series Plots: Such as torque trends and error-angle evolution over time.

Replay-Overlay Panels: Panels visually structured to place historical and real-time data adjacent for differential analysis.

3.3.4 Architectural Support for Visual Extension

The visualization subsystem is designed with extensibility as a core principle. Its component-based architecture supports the seamless addition of new visualization features without requiring changes to the core application. This is enabled through clearly defined rendering interfaces, modular configuration logic, and a runtime schema registry that governs chart instantiation and interaction behavior [24, 10, 7, 3, 5].

All charting components conform to a standard contract that specifies required data bindings, rendering behavior, and optional user controls. This design allows developers to introduce new visualization types simply by implementing a base component interface, registering the module in the dashboard manager, and providing any domain-specific controls or preferences.

Key extensibility provisions currently supported include:

WebGL-Based 3D Rendering: Already used for LiDAR point cloud visualization, the rendering layer supports integration with WebGL-enabled components to handle real-time

3D spatial data within a ROS2 coordinate frame.

Flexible Grid Layout: The dashboard system uses a resizable and responsive layout grid that adapts to varying screen resolutions, making it suitable for both widescreen lab displays and smaller mobile devices.

Modular Panel Composition: Users can dynamically rearrange panels, add or remove chart modules, and configure topic bindings at runtime without modifying the source code. This is enabled through schema-driven configuration files that persist layout and binding state across sessions.

Visual Overlay Panels: The architecture supports comparative visualization by rendering multiple signals within the same panel or across adjacent charts. This is commonly used to overlay live and replayed telemetry, such as comparing real-time control output with a known-good reference trajectory.

Component Reuse: Existing visualization modules—such as time-series plots, bar charts, and dial gauges—can be reused across dashboards by cloning their configuration and rebinding to different telemetry topics or measurement streams.

Additionally, the platform supports frontend capability negotiation. Each visualization module declares its rendering requirements (e.g., WebGL support, data rate thresholds), allowing the system to gracefully degrade or disable components in environments with limited processing power or network bandwidth [49].

These architectural choices ensure that the visualization layer can evolve alongside telemetry demands and measurement complexity, while maintaining backward compatibility and user consistency.

3.4 Deployment and Integration Strategy

This section outlines the deployment architecture and integration mechanisms of the telemetry and measurement system. The platform is designed to operate reliably across heterogeneous execution environments—namely Linux-based robots for live telemetry, Windows-based hosts for automotive measurement, and browser-based clients for data visualization. A hybrid deployment strategy is adopted that combines native execution, containerized services, and cross-platform bridges, all coordinated via schema-driven communication. This strategy promotes modularity, maintainability, and platform-agnostic resilience [3].

3.4.1 Linux–Windows Deployment Separation

The system is architecturally divided across two primary execution domains:

Linux Domain: The ROS2-enabled robot operates natively on Ubuntu, publishing real-time telemetry via DDS. A Node.js backend for historical '.db3' replay runs in a separate container. These components are entirely decoupled and individually deployable.

Windows Domain: ETAS INCA and its ETK hardware modules run natively on a Windows system due to their reliance on proprietary COM APIs, USB drivers, RCI2 scripting, and GUI-bound execution flows [20, 19].

Communication between these domains is achieved as follows:

- The Angular-based frontend connects directly to the ROS2 robot using `roslibjs` and `rosbridge_server`, establishing WebSocket links for real-time topic subscriptions [38, 46, 27].
- Historical data playback is handled by a Node.js application that parses '.db3' files using ROS2-compatible schemas and emits topic-aligned messages to the frontend over WebSocket. This service is deployed in a containerized environment, independent of the robot.
- A Windows-resident INCA agent exposes REST and WebSocket APIs, allowing the backend to send control instructions for measurement workflows. The agent executes RCI2 scripts or COM commands and streams session feedback (e.g., status, timestamps) to the user interface.

All message formats follow JSON or Protocol Buffers, allowing versioned schemas and tool-independent contracts. This approach ensures that each subsystem can evolve without tightly coupling logic across operating systems or runtimes.

3.4.2 Containerization and Runtime Environment

Linux-side services are containerized using Docker [31, 3] to simplify deployment, enforce environmental consistency, and support rapid iteration. These include:

1. ROS2-to-frontend bridges (for debugging, optional)
2. The Node.js '.db3' replay server
3. Middleware services for schema validation and message logging
4. WebSocket and REST API wrappers for the frontend interface

Each container exposes a minimal set of open ports and uses mounted volumes for access to '.db3' files, log outputs, and shared configurations. Orchestration is handled via 'docker-compose' for local deployments or Kubernetes for test labs with high scalability requirements [5, 4].

Advantages:

1. Platform-independent deployments across development and production
2. Versioned, rollback-capable container images for reproducibility
3. Health-checks and auto-restart policies for service resilience

Windows Constraints: Due to licensing, GUI interaction, and hardware dependencies, INCA cannot be containerized. Instead, a thin relay agent runs as a persistent Windows

service. This service listens on specific API endpoints, marshals control commands to the native INCA application, and logs all activity for debugging and traceability.

3.4.3 Backend Integration and Fault Tolerance

Robust integration between telemetry, playback, and measurement domains is enabled by the following core mechanisms:

Asynchronous Messaging and Retry Logic: All backend subsystems communicate via stateless, retry-capable message protocols. Control messages sent to the INCA agent require acknowledgments, while telemetry and replay data are transmitted over buffered channels to protect against frontend rendering delays or temporary disconnections [5, 42].

Subsystem Health Monitoring: Each critical component (Node.js backend, INCA agent, frontend handler) exposes a heartbeat or liveness endpoint. These are periodically polled by a monitoring layer, which triggers alerts or recovery actions if any node becomes unresponsive.

Session Persistence and Recovery: Experiment metadata—including selected topics, session state, and measurement history—is logged to a lightweight SQLite or PostgreSQL store. If a failure occurs, dashboards and automation scripts can resume without manual reconfiguration.

Fault Isolation: Each domain operates independently. An INCA failure on Windows does not affect ROS2 telemetry or replay services. Similarly, issues in the playback pipeline do not disrupt real-time data from the robot or block user interface interactions.

Centralized Logging and Debugging: Every backend service logs API usage, error codes, timing metrics, and topic activity with timestamps. These logs are optionally pushed to centralized aggregators such as Grafana Loki or ELK Stack for long-term monitoring and visualization.

3.5 Observability, Security, and Reproducibility

This section outlines the supporting mechanisms that ensure system integrity, secure communication, and the reproducibility of telemetry-driven experiments. As the telemetry and measurement platform operates in distributed, cross-domain environments, robust monitoring, access control, and traceability mechanisms are critical. This includes not only ensuring the system remains observable and diagnosable in real time but also securing communication channels and allowing experiments to be repeatable and verifiable by design [3, 5].

3.5.1 Monitoring, Logging, and Experiment Integrity

To provide full visibility into system behavior and runtime conditions, the platform includes a comprehensive observability stack composed of health monitoring, structured logging, and experiment trace metadata [5].

Health Monitoring and Heartbeats: Each service component—including the Node.js replay backend, INCA control agent, Angular frontend, and middleware proxy—exposes a lightweight heartbeat endpoint over HTTP or WebSocket. A central monitoring module regularly polls these endpoints and logs response latency and status. Failures to respond within the expected interval trigger alerts or automatic remediation routines, such as container restarts or fallback service activation.

Structured Logging and Traceability: All telemetry flow events, measurement commands, and service interactions are logged with contextual metadata. Logs include timestamps, session IDs, topic names, message types, and operation status. This is implemented using standard logging libraries on each backend (e.g., Winston for Node.js, and file-based loggers on the Windows INCA agent).

Logs are stored locally in structured formats (e.g., JSON) and optionally streamed to centralized platforms such as Grafana Loki, Promtail, or the ELK Stack. These tools enable developers and testers to visualize trends, trace anomalies, or audit command sequences in real time or post-mission [3].

Experiment Integrity and State Snapshots: To support reproducible testing workflows, the system captures snapshots of each telemetry session. This includes:

- Selected ROS2 topics or measurement signals
- Dashboard layout and chart bindings
- Configuration files or experiment parameters
- Measurement command history (e.g., INCA triggers)

These snapshots are stored in versioned files (e.g., as JSON descriptors), and can be reloaded to recreate prior visualization or measurement contexts. The session snapshot feature enables A/B testing, cross-run comparisons, and long-term regression analysis [42].

3.5.2 Secure Communication and Access Control

Security is enforced at multiple layers to ensure that telemetry data, control commands, and user credentials remain protected during operation.

TLS-Encrypted Channels: All WebSocket and HTTP endpoints are served over TLS, ensuring encryption in transit between frontend dashboards, Node.js replay services, and the Windows-based measurement agent. TLS certificates are either self-signed for lab deployments or issued by a trusted internal CA for production environments.

JWT-Based Authentication: User sessions and control commands are authenticated using JWT [30]. Tokens include claims for user identity, session validity, and access roles (e.g., admin, viewer). Middleware services verify the tokens before permitting access to replay control endpoints or measurement triggers.

CORS and Origin Restrictions: The Node.js backend and INCA control services enforce Cross-Origin Resource Sharing (CORS) policies to restrict which clients can initiate

connections. Only trusted origins (e.g., whitelisted Angular frontend domains) are permitted to interact with core services [29].

Rate Limiting and Abuse Prevention: To protect services from overload or misuse, rate limits are imposed per client IP and user session. Additionally, sensitive operations—such as experiment deletion or INCA reset—require multi-step confirmation and administrative tokens [3].

3.5.3 Session Isolation and Authorization

The system supports multi-user workflows where multiple test engineers or developers may interact with the platform concurrently. To ensure privacy, integrity, and predictable behavior, session isolation mechanisms are employed.

Session Scoping: Each session maintains an independent configuration context, including selected topics, dashboard layouts, and measurement control states. Sessions are uniquely identified by UUIDs and mapped to their user tokens.

Access Role Enforcement: Authorization policies are enforced based on roles defined in the JWT [29]. For instance:

1. **Viewers** can inspect dashboards but cannot initiate measurement commands.
2. **Test Engineers** can start/stop replay or INCA recordings.
3. **Administrators** can modify session configurations, manage users, and access full logs.

Frontend User Interface Isolation: User interface components are scoped per session to avoid shared dashboard state or race conditions. Each user sees only their active charts, layout preferences, and measurement data streams.

Audit and Revocation: All session activity is auditable. Logs include session creation, token usage, and command execution. Sessions can be forcefully revoked through an admin interface in case of expired access or suspicious behavior [3].

These safeguards ensure that the system remains reliable, secure, and compliant with common cybersecurity and experimental reproducibility standards, even when accessed by multiple stakeholders in parallel.

3.6 Concept Summary and Forward Linkage

This chapter presented the conceptual foundation and architectural blueprint of the telemetry and measurement system. It outlined the key components, communication pathways, and design principles that enable the platform to operate across robotic and automotive domains. This section summarizes the layered architecture and prepares the reader for Chapter 4, which details the implementation.

3.6.1 Recap of Conceptual Layers

The proposed system is structured into modular layers that support scalability, maintainability, and interoperability:

Telemetry Ingestion and Replay: Live telemetry from ROS2 robots streams to the Angular frontend via 'rosbridge_server', while '.db3' data is replayed using a Node.js backend, both through WebSocket [38, 40, 46, 27].

Measurement Control and Integration: Automotive measurements are managed using INCA and ETK devices through a Windows agent executing COM and RCI2 commands, with structured logs [1, 35, 20, 19].

Middleware Coordination: DDS, WebSocket, and HTTP enable pub-sub telemetry and service-based control. QoS settings ensure reliable and efficient message delivery [26, 5, 42].

Visualization and Interaction: The Angular frontend renders telemetry using ECharts and Chart.js. Users configure layouts, filter topics, and compare live and replayed data [10, 7, 24, 49].

Deployment Architecture: Linux services run in Docker containers, while INCA remains on a native Windows host. Components communicate over secure WebSocket and HTTP bridges with JWT-based authorization [31, 30, 3].

Observability and Reproducibility: Health checks, structured logging, and session snapshots support traceability and secure multi-user experimentation [5, 42].

Each layer is independently deployable and built on open standards, allowing reproducible setups in lab and field settings.

3.6.2 Link to Chapter 4 (Implementation)

The next chapter transitions from concept to realization. Chapter 4 will:

- Describe code structure, APIs, and configuration logic used to implement the system.
- Present screenshots and logs showing data flow, replay behavior, and INCA measurement control.
- Explain deployment scripts (e.g., 'docker-compose.yml') and runtime setup on Linux and Windows.
- Demonstrate how the modular frontend renders multiple data sources using unified logic.
- Discuss design trade-offs, such as browser-based ROS2 ingestion and COM-driven INCA control.

Together, Chapters 3 and 4 connect system design to implementation, illustrating how an integrated telemetry and measurement platform can be built with focus on usability and modularity.

Chapter 4

Implementation

This chapter presents the practical implementation of the telemetry visualization and automotive measurement platform described in Chapter 3. The goal is to demonstrate a modular, cross-platform system capable of acquiring, replaying, and visualizing structured telemetry data from robotic middleware and ETK-based measurement devices [3, 5].

Key aspects include integration of ROS2 topic streams using web-native technologies, a Node.js backend for historical replay of '.db3' files, and a C++ DLL for controlling ETAS INCA through COM and RCI2 interfaces. The Angular frontend renders both real-time and historical data through a configurable dashboard supporting a range of sensor visualizations including LiDAR, temperature gauges, and voltage plots.

To support usability, the implementation adopts a containerized deployment model for backend components, fault isolation mechanisms, and runtime configuration on the frontend. Each implementation detail aligns with the architectural principles of separation of concerns, platform interoperability, and end-user accessibility introduced in earlier chapters [42].

4.1 System Architecture Overview

This section provides a high-level overview of the telemetry visualization and measurement platform architecture. The system integrates robotic middleware, automotive measurement tools, and web-based visualization into a modular runtime environment. It is designed to bridge real-time robotics telemetry, historical data playback, and automotive calibration data through a unified frontend interface. Communication spans both Linux and Windows domains and leverages stateless messaging protocols for interoperability.

4.1.1 High-Level Integration Diagram

Figure 4.1 illustrates the major system components and their interactions. The platform integrates three primary data sources: (1) live ROS2 telemetry published by the robot, (2) historical telemetry logs stored in `.db3` format, and (3) ETK-based measurement sessions

controlled via INCA. All data sources feed into an Angular-based dashboard that renders structured telemetry through modular widgets.

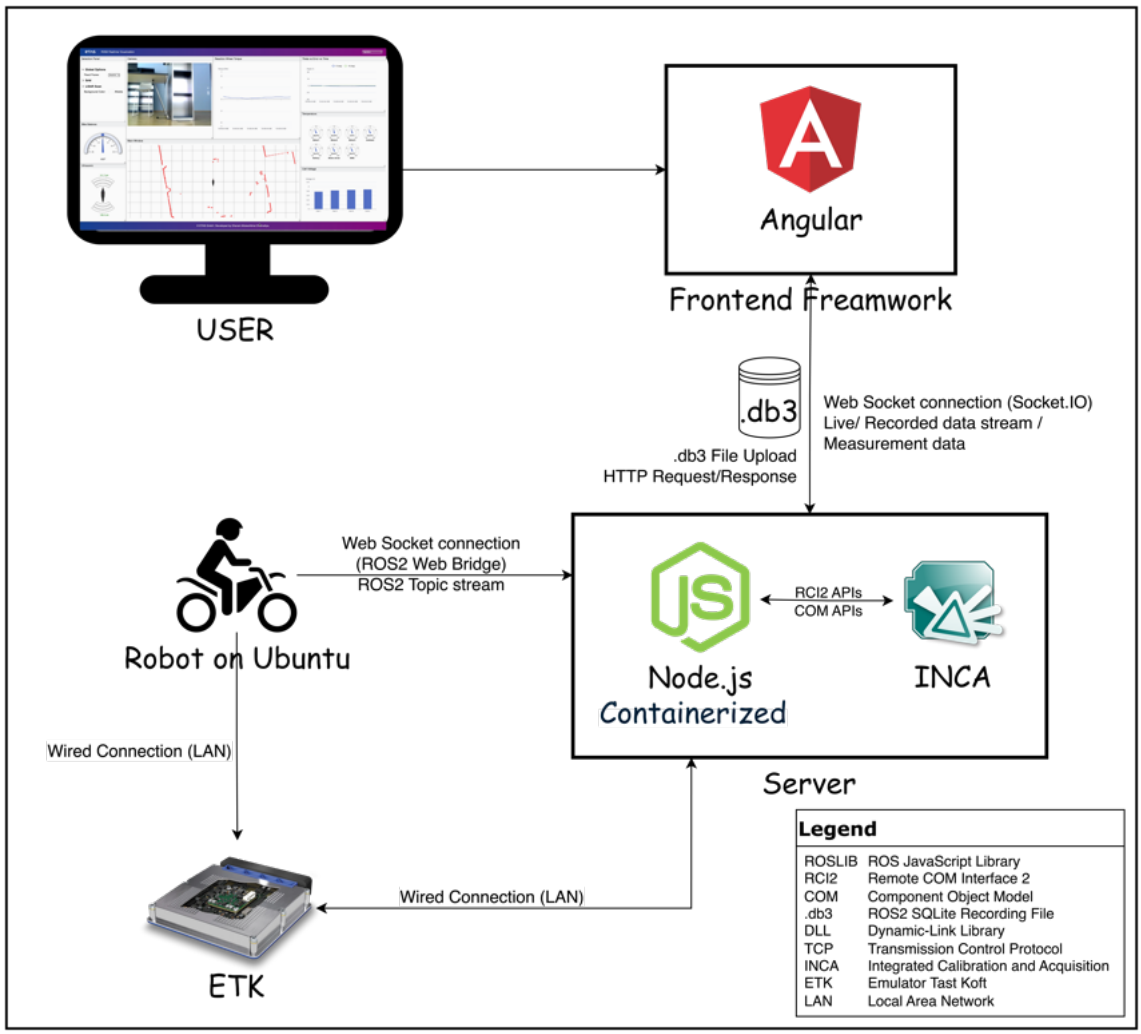

Figure 4.1: High-level system architecture

The network setup includes a wireless (Wi-Fi) connection for ROS2 live telemetry and wired Ethernet links for ETK and INCA-related communication. This hybrid configuration supports both mobile and stationary components while maintaining data integrity.

4.1.2 Technology Stack Summary

The system combines open-source and proprietary technologies to meet requirements such as low-latency rendering, cross-platform deployment, and hardware integration [3, 5]. Key technologies include:

ROS2 Humble (Ubuntu): Real-time robotics middleware using DDS [38, 26].

rosbridge_server + roslibjs: WebSocket bridge exposing ROS2 topics to web clients [9, 46].

Node.js + SQLite3: Backend service for historical data replay using `.db3` logs [40, 14].

Angular 18: Frontend framework for telemetry visualization [24].

Chart.js, Apache ECharts: Libraries for rendering interactive charts and gauges [7, 10].

C++ DLL + RCI2 + COM API: INCA automation and ETK measurement control on Windows [20, 35, 19].

4.1.3 Deployment Boundaries and Communication Links

The platform operates across Linux and Windows hosts. ROS2 telemetry is transmitted to the frontend via Wi-Fi, while ETK devices and the INCA host communicate over wired LAN for reliable measurement control. Protocols include WebSocket for telemetry streaming, HTTP for file uploads and configuration commands, and COM/RCI2 for native INCA operations [42]. Details of protocol handling and service logic are provided in Section 4.2.

4.2 Backend Architecture and Integration

This section describes the backend architecture responsible for telemetry ingestion, historical replay, measurement control, and frontend hosting. The design integrates Linux-based services with Windows-native tooling to create a modular and cross-platform platform. Communication between components leverages stateless protocols such as WebSocket, HTTP, and COM/RCI2 to ensure interoperability, scalability, and ease of maintenance [5, 42].

4.2.1 Node.js Backend Services

The Node.js backend fulfills several critical roles in the platform.

Historical Data Replay: The backend parses telemetry logs stored in `.db3` format (generated by `rosbag2`) and streams replayed data over WebSocket using `Socket.IO`. The service preserves original message timestamps and ordering, while supporting dynamic playback rate adjustments and topic filtering [40, 8, 45].

HTTP API Endpoints: The backend provides REST-style endpoints for uploading `.db3` files, configuring playback parameters, and monitoring service status. Uploaded files are stored in a shared volume accessible to the replay engine and indexed for session management.

Frontend Hosting: The Node.js server also serves the Angular frontend as static build artifacts. This eliminates the need for a separate web server and simplifies deployment. The build files are hosted via HTTP, enabling the user interface to be accessed through standard web browsers [14, 24].

4.2.2 Windows Host for INCA Integration

A dedicated Windows host runs the ETAS INCA environment alongside a lightweight Node.js service that acts as a control bridge. Key components include:

- **C++ DLL:** A custom dynamic library exposes functions that allow INCA to be controlled programmatically via COM API and RCI2 scripting interfaces [20, 19, 35].
- **Node.js Control Service:** This service loads the C++ DLL using foreign function interface bindings [11]. It receives commands (e.g., start or stop measurement) via HTTP or WebSocket from the Linux-based backend, invokes the corresponding DLL functions, and relays outcomes back to the platform.
- **Measurement Data Export:** INCA sessions controlled through this integration generate measurement files in `.mf4` format, which are stored on the Windows host for further processing or analysis [1].

4.2.3 Communication Protocols and Integration

The backend architecture employs a hybrid communication model

WebSocket: Used for streaming live telemetry from the robot (via `rosbridge_server`) and replayed data from the Node.js backend to the frontend interface [9, 46, 8].

HTTP: Used for uploading `.db3` files, issuing configuration commands, and serving the Angular frontend files.

COM + RCI2: Employed within the Windows host to enable native control of INCA and ETK measurement workflows [35, 20].

The design ensures that backend services operate independently, with clear boundaries between Linux and Windows components. This modularity facilitates maintenance, fault isolation, and future extension of the platform [5].

4.3 Frontend Data Handling and Visualization Framework

This section describes the frontend implementation that transforms incoming telemetry data—whether from live ROS2 topics, historical `.db3` replay, or ETK measurement sessions—into interactive visualizations rendered in a modular Angular-based dashboard. The frontend architecture remains agnostic to data source and supports real-time updates, fault-tolerant rendering, and user-driven layout customization. Specific attention is given to how each visualization widget processes data, integrates with shared services, and visually communicates telemetry states to the user.

4.3.1 Angular Services for Telemetry Streaming and Measurement Control

The frontend interfaces with three main sources using dedicated Angular services [24]:

1. Live ROS2 Data: The application connects to the robot via WebSocket using `rosbridge_server`. The `LiveService` wraps `roslibjs` [46], enabling direct subscrip-

tion to ROS2 topics. Subscriptions are initiated during component lifecycle events, with QoS-aligned reception and automatic reconnection logic.

2. Historical `.db3` Replay: The `HistoricalService` handles WebSocket messages emitted by the Node.js backend using `Socket.IO` [8]. Messages are timestamped and decoded using topic schemas before routing to the rendering pipeline.

3. ETK Measurement Control: The frontend provides a control panel for users to send commands (start/stop measurement, add signals) via HTTP to the backend. The backend relays these commands to INCA, where they are executed through the C++ DLL using COM and RCI2 [20, 35].

All services:

- Buffer messages in per-topic queues.
- Broadcast updates using RxJS observables [39].
- Apply minimal transformations for widget compatibility.

4.3.2 Widget Composition and Chart Integration

The dashboard is built from independent Angular components, each visualizing a signal type using Chart.js or ECharts [7, 10]. Components:

- Initialize charts via `ViewChild` [24].
- Subscribe to telemetry using unique topic IDs.
- Parse, normalize, and render data dynamically.
- Throttle redraws or batch updates for high-frequency data.

Styling uses SCSS modules, with responsive layouts and persistent user configurations.

4.3.3 Dashboard Layout and Visualization Modules

The dashboard integrates multiple modular widgets, each designed to represent specific telemetry signals in a form suited for human interpretation (Figure 4.2). The following widgets are currently implemented:

LiDAR Visualization: This widget renders 2D point cloud data using Chart.js. The points are plotted on a Cartesian grid based on their x-y coordinates extracted from ROS2 sensor messages. Users can switch between two fixed reference frames (`odom` or `base_link`) depending on the desired perspective for spatial alignment. The visualization auto-centers and rescales dynamically according to the frequency of incoming scan messages, ensuring that recent data remains in view.

Camera Feed: The camera feed is streamed as a base64-encoded image string, rendered in a native HTML `<img>` element. To minimize browser rendering overhead and visual artifacts (such as flicker during rapid updates), the image stream is throttled. This ensures that frame updates remain smooth while preserving real-time characteristics.

Cell Voltage Bar Chart: This ECharts-based widget visualizes the voltage levels of four battery cells. The operational range is between 3.0 V and 4.2 V, which corresponds to safe lithium-ion battery operation. A voltage below 3.0 V typically indicates a deeply discharged (dead) cell, which risks permanent damage and loss of capacity. Conversely, voltages above 4.2 V exceed the standard upper limit for lithium-ion cells, significantly increasing the risk of thermal runaway or explosion. The bar chart updates in real time, highlighting deviations from the safe operating range.

Ultrasonic Arc Visualization: This widget displays proximity data from two ultrasonic sensors, with one sensor mounted at the front and another at the rear of the bike. Each sensor's reading is represented as a layered arc. The arcs change color based on measured distance:

- Distances greater than 50 cm result in transparent arcs, indicating no immediate obstacle.
- When any single arc (either front or rear) detects an object within 31–50 cm, the corresponding arc displays yellow to signify caution.
- Objects within 15–30 cm trigger an orange color on the middle arc layer, signaling elevated risk of collision.
- Distances below 15 cm cause all three arc layers to turn red, indicating critical proximity and the need for immediate attention or action.

This layered approach provides both directional awareness and graded proximity feedback.

Temperature Gauges: Seven individual gauges, implemented using ECharts, display temperatures of key components: Motor 1, Motor 2, Motor 3, Ambient, Battery, Motor Driver, and Battery Management System (BMS). The gauges operate over a –20°C to +100°C range, which covers both typical and extreme operating conditions. Temperatures outside this range could indicate sensor faults, dangerous overheating, or environmental conditions beyond design specifications.

Bike Balance Gauge: This semi-circular dial shows the tilt angle of the bike platform. It measures from –30° (maximum left lean) to +30° (maximum right lean), with 0° indicating an upright position. The needle dynamically reflects the live tilt angle, aiding operators in balance monitoring during tests or live operation.

Reaction Wheel Torque Chart: This line chart plots torque applied by the reaction wheel over time, with a y-axis spanning –7 to +7 Nm. Torque values within this range represent the operational envelope of the reaction wheel system, where corrective forces are applied to maintain or adjust platform orientation. A moving average smoothing algorithm is applied to the data to help users discern trends and reduce the impact of transient noise.

Theta vs Error Chart: This time-series chart plots both `theta` (actual orientation) and `error` (deviation from desired orientation) on a shared timeline. The y-axis spans –20° to +20°, covering the expected bounds of orientation error in the system. The dual-series view allows users to observe how orientation errors evolve over time and correlate them with corrective actions taken by the control system.

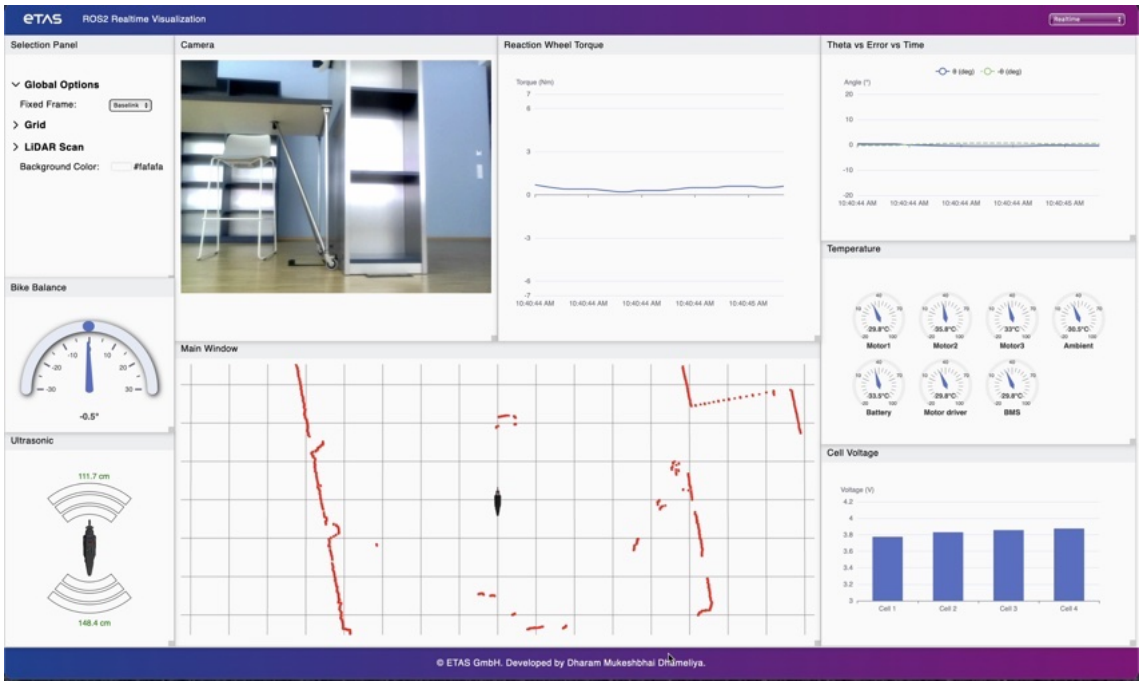

Figure 4.2: Dashboard layout with telemetry widgets

4.3.4 Measurement Control Interface

In addition to visualizing telemetry data, the frontend provides a dedicated interface for managing ETK-based measurement sessions. This interface is integrated into the dashboard and enables users to configure and initiate measurement workflows directly from the browser.

The interface includes the following interactive elements:

- **Database Selection:** A dropdown menu populated with available INCA project databases. Users select the target database corresponding to the measurement configuration required for the experiment.
- **Experiment Folder Name:** A text input field where users specify the folder name under which measurement data and logs will be stored. This folder helps organize outputs and session metadata.
- **Experiment Name:** A dedicated field for defining the specific name of the current measurement session. This name is applied to generated files (e.g., exported `.mf4` data) and used in session tracking.
- **Workspace Configuration:** Fields for specifying the workspace folder and workspace name, enabling organization of experiments within structured directories.
- **Signal Selection:** A dynamic form allowing users to add multiple signals for inclusion in the measurement. Each signal entry can be configured individually, and additional entries can be added as needed.

- **Start/Stop Controls:** Clearly labeled buttons enable users to initiate and terminate the measurement session. These controls send commands to the backend, which then invokes the INCA control routines via the Node.js agent and C++ DLL.

Upon measurement start, the interface displays a latest values table. This table lists active signals along with their associated device, group, signal name, current value, and any relevant metadata provided by the backend. The table updates dynamically during the measurement session, providing continuous feedback and supporting the user in monitoring measurement integrity and progress.

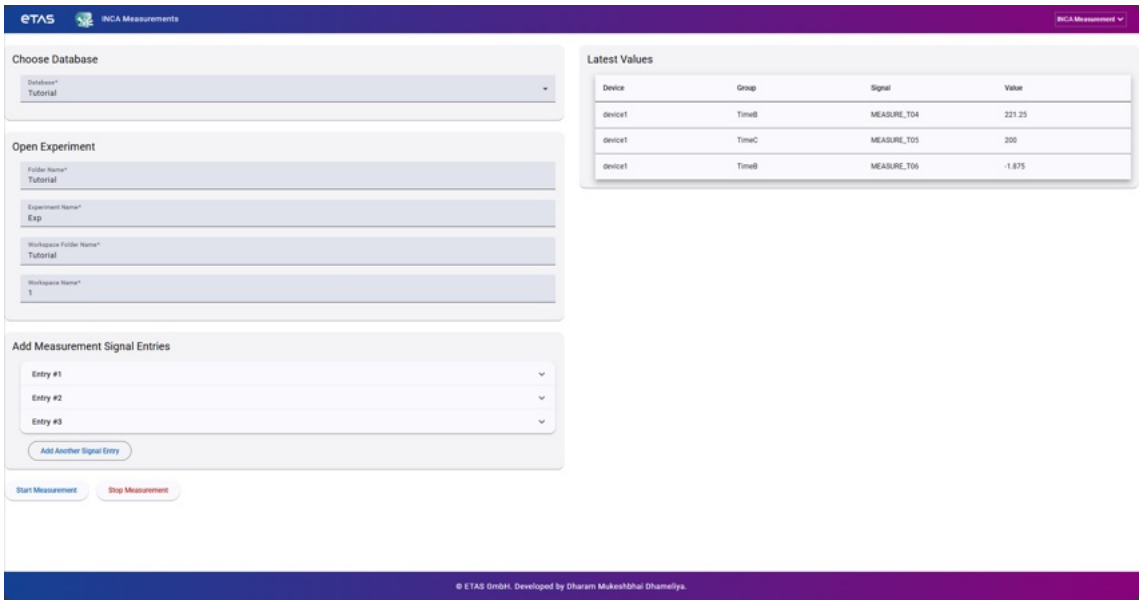

Figure 4.3: Measurement control interface

This measurement control module ensures that users can perform end-to-end session management without direct interaction with the native INCA interface, streamlining workflows and reducing dependency on manual configuration steps in the Windows environment.

4.3.5 Runtime Updates and Synchronization Logic

All messages—live, replayed, or from measurement sessions—are normalized (`topic`, `timestamp`, `value`, optional `frame`). Angular's `OnPush` change detection ensures efficient updates. User preferences for layout, widget visibility, and topic assignments persist locally [36]. Measurement session feedback (e.g., active signals, status) is shown alongside telemetry data for integrated monitoring.

4.4 Containerized Deployment Strategy

The system's backend architecture is designed for modular deployment using containerization technologies. This approach ensures reproducibility across development environments, eases integration with CI/CD pipelines, and provides a clean abstraction between services. While most Linux-based services are fully containerized, Windows-bound components such as ETAS INCA remain outside container orchestration due to their GUI requirements and licensing constraints [20, 31].

4.4.1 Docker Architecture for Node.js Replay and Middleware

The Node.js replay container is encapsulated within a Docker image that provides WebSocket-based emission of parsed `.db3` telemetry files to the frontend application. In addition to replay functionality, this container also serves the static Angular build files over HTTP, providing the frontend interface for telemetry visualization and measurement control.

The container setup includes:

- A Node.js environment with installed dependencies such as `sqlite3`, `Socket.IO`, and internal schema decoders [8, 14].
- Entrypoint scripts to load and decode `.db3` files and emit topic-aligned streams.
- Custom environment variables for setting replay rate, base topic paths, or WebSocket configurations.
- A healthcheck script to track uptime, log anomalies, and report container status.

This container mounts a shared volume for `.db3` uploads and metadata, with an exposed WebSocket interface used by Angular. Internally, telemetry messages are parsed per-topic and mapped with synthetic timestamps that preserve playback fidelity. These messages follow a consistent format aligned with live ROS2 telemetry, ensuring frontend rendering logic remains unchanged [40].

In addition to the replay engine, optional services such as schema validators, metadata enrichers, or log serializers are containerized. Each of these services is loosely coupled and deployable through `docker-compose` [31], promoting portability and testbench flexibility. For further details on backend roles, see Section 4.2.

4.4.2 Inter-Container Communication

Containers are deployed on a user-defined Docker bridge network, enabling name-based service resolution and internal-only routing. Inter-container communication occurs via the following mechanisms:

WebSocket: Used by the Node.js replay container to send real-time telemetry data to the frontend.

HTTP/REST: Used by coordination services (e.g., configuration agents, health monitors).

Volume Sharing: `.db3` files, session configs, and logs are shared via mounted volumes accessible across containers.

All containers declare clear port bindings, health policies, and restart logic in the deployment descriptor (`docker-compose.yml`). For fault tolerance, these containers may also include log rotation rules and memory limits to prevent resource starvation. Observability is improved via logging sidecars or remote monitoring endpoints [31].

This containerized design promotes modular updates. For instance, telemetry decoding logic can be updated or rolled back independently of the rest of the system. If integrated into CI pipelines, image versions are tagged and deployed per branch or test session.

4.4.3 Limitations with INCA and Host-Based Execution

While most backend components in the system are deployed using containers and benefit from the flexibility of orchestration on Linux, the ETAS INCA environment remains a notable exception. Due to its dependency on a graphical user interface, a proprietary licensing model, and direct hardware access requirements, INCA must run directly on a Windows host system rather than within a containerized setup [20].

Several technical constraints contribute to this limitation. For instance, INCA requires access to USB dongles for license validation, and the ETK drivers must interact with hardware interfaces at the system level. Moreover, its COM-based automation APIs are tightly coupled with an active Windows session that supports user interface operations. Additionally, the RCI2 scripts used to automate measurement control must run inside the live INCA runtime, making it unsuitable for headless or virtualized deployment without significant workarounds.

To bridge this gap, a lightweight automation layer was introduced. This component is implemented as a Node.js service running in the background on the Windows machine that hosts INCA. It acts as a communication relay, accepting control commands from the Linux-based middleware through local HTTP or WebSocket endpoints. These commands may request actions such as starting or stopping a measurement session, polling the current status of INCA, or loading specific project configurations.

Once a command is executed, the Windows service gathers session metadata and logs the outcome. This information is then forwarded to a shared storage location or monitoring tool for further use. Although INCA itself cannot be containerized, this hybrid strategy effectively isolates control logic while preserving compatibility with the required Windows environment.

Looking ahead, more advanced isolation methods—such as using virtual machines, containerized GUI virtualization solutions (e.g., Windows containers with GUI support or WSLg), or remote desktop-based automation agents—may offer improved modularity and portability. However, these options were not pursued in the current implementation due to added complexity and licensing implications.

4.5 Debugging, Logging, and Session Tracking

This section describes the diagnostic and session tracking mechanisms built into the system to support testing, troubleshooting, and reproducibility. Given the hybrid nature of the platform—combining live ROS2 telemetry, historical `.db3` playback, and INCA-controlled ETK measurements—it is essential to have reliable feedback mechanisms that allow developers to understand the system's behavior during operation. Rather than relying on third-party logging frameworks, the implementation favors lightweight, transparent tracing using native console outputs and manually written log files.

Console outputs provide real-time visibility into the system's internal state during operation, while log files serve as persistent records for post-session analysis and performance validation. This dual-layered approach ensures that developers can trace issues across different components without introducing unnecessary external dependencies. The logging system is designed to validate configurations, capture anomalies, and document interactions across the telemetry and measurement workflows.

4.5.1 Replay Logs and Console Tracing

The Node.js backend responsible for handling `.db3` files includes detailed inline logging that assists during development and debugging. Each telemetry session prints a summary of the parsed database to the console, including the number of discovered topics, message counts per topic, and schema resolution results. As the session progresses, emitted messages are logged with their associated topic identifiers and timestamps, helping verify alignment between replay timing and expected progression.

Alongside console output, a plain text log file is created for each session. This file captures metadata such as the session start time, the path to the loaded `.db3` file, playback speed, applied topic filters, and any notable anomalies detected during processing. Logs are written to a shared volume (see Section 4.4) to facilitate traceability across runs. Although no automated log rotation is currently implemented, manual cleanup scripts are used during development to prevent excessive disk usage. Future enhancements may include structured logging (e.g., JSON) or integration with centralized log aggregation tools.

4.5.2 ETK Session Metadata Output

On the Windows side, the Node.js control service managing ETK and INCA measurement sessions implements its own logging routines. Each received command—such as starting or stopping a measurement—is recorded in a timestamped `.log` file. Outcomes of RCI2 script executions, including success, failure, or timeout states, are logged with contextual details.

The service also records session-specific identifiers such as measurement names, exported file paths, and associated project configurations. All logs are stored in structured form within the active project directory, centralizing data for ease of access and correlation with telemetry records. Developers are advised to review logs for any confidential data (e.g., project names) before sharing for analysis or debugging.

4.5.3 Developer Tools and Diagnostic Hooks

To assist in debugging across multiple layers, the following diagnostic features are integrated:

Frontend Console Logging: Angular services print key lifecycle events (e.g., topic subscription, connection status changes, data throttling) to the browser console using debug flags that can be toggled during development.

Replay Progress Indicator: The frontend displays a progress marker that compares incoming message timestamps to the original session start time, providing soft feedback on playback progression.

Session Identifier Propagation: A unique session UUID is generated at startup and included in all log files and cross-component communication, supporting log correlation and multi-source trace stitching.

Manual Log Download: The Angular interface provides an option for users to download raw logs and chart snapshots for offline analysis.

These diagnostic features support development and testing without introducing overhead into production workflows. They form a foundation for future enhancements such as automated log analysis, cross-container log aggregation, or external monitoring integrations.

4.6 Challenges During Development

The development of the telemetry visualization and measurement platform presented a number of significant challenges across frontend rendering, backend data processing, and cross-platform integration. These obstacles were encountered at various stages and required a combination of custom tooling, debugging techniques, and iterative design adjustments to overcome. Below, the most notable challenges are described in detail along with the approaches used to resolve them.

Challenge 1: Integrating a 3D Bike Model into the LiDAR Scan View.

To improve spatial awareness during LiDAR visualization, a 3D representation of the bike was embedded within the scan chart using the Three.js library. This involved loading the bike's URDF description and rendering its mesh components with the help of the `urdfloader` utility. The primary difficulty lay in aligning the 3D model with live scan data in real time. Frame transformations between the ROS2 coordinate references and the rendering context required careful calibration. Achieving stable alignment involved tuning both positional offsets and orientation parameters through multiple iterations, particularly when switching between different fixed frames like `odom` and `base_link`. This solution significantly enhanced the usability of the LiDAR visualization module for end users.

Challenge 2: Real-Time Plotting of LiDAR and Odometry Data.

Rendering sensor data such as laser scans and odometry posed unique challenges due to the nested and variable-length structure of the messages. Parsing these arrays efficiently

on the frontend without causing user interface slowdowns proved difficult. Additionally, discrepancies between coordinate frames sometimes led to visual inconsistencies when combining data from different sources on the same chart; these were largely addressed through the frame calibration efforts described in Challenge 1. Further issues arose from noise in the scan readings and null values within data arrays. These were mitigated by introducing preprocessing steps to clean and filter the data, setting fixed axis constraints to stabilize chart rendering, and adding tooltips to assist with interpretation. These improvements contributed to a more stable and informative visualization experience.

Challenge 3: Decoding Historical ROS2 Data from `.db3` Files.

Parsing `.db3` files generated by `rosbag2` was one of the more technically complex aspects of the backend. Although the files use SQLite as their base format, the contents include binary blobs that encapsulate ROS2 messages with protocol-specific headers. Extracting meaningful data required bypassing these headers and reconstructing the original topic messages according to their schema. This process also involved grouping messages by topic and preserving the original timestamps for accurate replay. Additional care had to be taken to manage memory usage and avoid database locks, particularly when handling high-frequency topics. To support this, a custom Node.js parser was implemented using the `sqlite3` binding and integrated with logic for safely decoding each message block. Although this solution met immediate decoding needs, future work could explore more efficient decoding libraries or parallel processing to handle larger data sets.

Challenge 4: Creating and Testing a C++ DLL for INCA and ETK Integration.

Enabling automated measurement control through ETAS INCA required the development of a custom C++ DLL that could be invoked from the Node.js backend. The DLL was designed to expose key control functions—such as starting and stopping measurement sessions—via `extern "C"` interfaces, making them compatible with FFI bindings in JavaScript. However, several issues surfaced during implementation. Managing memory safely across the C++ and Node.js boundary was particularly delicate, and any misalignment in data types or pointers could lead to crashes or undefined behavior. Furthermore, reliably invoking INCA's COM and RCI2 interfaces within a multithreaded DLL context introduced timing and synchronization issues. Testing the DLL under real conditions was complicated by licensing constraints, as the INCA runtime requires an active Windows session, GUI access, and connected ETK hardware. These hurdles were addressed by adding structured error handling, clearly defined function signatures, and debug output routines that helped isolate issues during test runs [11]. This solution enabled reliable measurement automation and streamlined backend integration with INCA.

Overall, the platform's development required a combination of system-level insight, cross-language integration, and performance optimization. Each challenge introduced technical complexity, but the solutions developed during this process contributed significantly to the system's robustness and reliability.

4.7 Summary and Implementation Scope

This chapter has detailed the complete implementation of a cross-platform telemetry and measurement visualization platform. The system brings together multiple layers of functionality: real-time ROS2 topic streaming, historical playback of '.db3' telemetry files, and ETK-based measurement control through ETAS INCA. All of these data sources are integrated into a unified frontend built with Angular, offering a configurable dashboard that renders structured sensor data through a range of domain-specific visualization modules. The implementation aligns with the architectural principles of modularity, interoperability, and observability outlined in Chapter 3.

Among the key milestones achieved during implementation is the successful integration of live telemetry using `roslibjs` and `rosbridge_server`, which allows the frontend to directly subscribe to sensor topics with full support for quality-of-service constraints. The system also includes a standalone, containerized Node.js replay engine that parses '.db3' files and emits timestamp-aligned messages over WebSocket using `Socket.IO`, enabling historical sessions to be visualized in the same interface as live data.

Another notable accomplishment is the creation of a custom C++ DLL that enables automated control of INCA-based ETK measurements. This DLL interfaces with the Node.js backend and communicates with INCA through both the COM API and RCI2 scripting interface. On the frontend, the application delivers a responsive and user-friendly visualization experience, featuring modular components for LiDAR scans, torque readings, ultrasonic sensor feedback, and temperature and voltage gauges.

Realized vs Planned Features

The current version of the platform includes all foundational features necessary for telemetry ingestion, measurement control, and real-time visualization. Live ROS2 topics and `.db3` logs can both be rendered interchangeably, while a series of custom-developed widgets cover key signal types such as battery voltages, balance indicators, ultrasonic sensor proximity, temperature gauges, and reaction wheel torque. The dashboard layout supports modular components that can display both live and historical data streams in a consistent and coherent manner. Communication across components is handled through a layered protocol setup involving WebSocket for live streaming, HTTP for configuration exchanges, and COM bridges for INCA control operations. The system further provides basic diagnostic logging and persistent dashboard layout configuration, enabling users to preserve visualization preferences between sessions.

In terms of backend integration, the platform successfully implements the ingestion and decoding of ROS2 `.db3` files using a Node.js service, and exposes measurement control functions via a C++ DLL to interface with the ETK/INCA stack. These capabilities ensure that both robotics telemetry and automotive measurement data can be processed and visualized in a unified environment.

However, not all envisioned features were implemented within the current scope. Several enhancements remain as future work. For example, advanced playback controls in the frontend — such as pause, seek, adjustable playback speed, and timeline scrubbing — were

planned but not realized. The current system plays historical data in real-time sequence only, without interactive timeline manipulation. Additionally, interactive frontend features like linked chart brushing, synchronized highlighting across multiple graphs, and user-configurable chart thresholds are not yet implemented. These would significantly enhance exploratory data analysis and correlation between different signals.

From a measurement control perspective, the planned frontend configuration interface — designed to allow users to define INCA measurement parameters, signal groupings, and recording conditions directly from the dashboard — was not completed. As a result, the current implementation requires predefined configurations or manual entry of parameters before session start. Furthermore, features such as automated session metadata export, integrated reporting tools, and API hooks for external toolchain integration (e.g., for automated post-processing or data warehousing) were deferred due to time and scope constraints.

Finally, while the system achieves its goal of providing a modular, cost-effective telemetry and measurement solution, additional refinements in security (such as more granular access control) and observability (including more sophisticated monitoring dashboards) are recommended for future iterations. These improvements would help ensure the platform's robustness and scalability in production or field deployment scenarios.

Scope of Live Demonstration

Although the system does not incorporate an automated test suite or scripted validation pipeline, a comprehensive live demonstration will be conducted as part of the thesis defense. This demonstration will cover key capabilities, including real-time ROS2 telemetry visualization, historical replay of `.db3` files, and dynamic dashboard interaction. The goal is to showcase the system's modularity, real-time observability, and seamless integration between robotics middleware and automotive measurement workflows.

During the live session, the candidate will present how the visualization dashboard connects to a robot (specifically, an E-Bike platform) and receives live ROS2 topic data over the network. The dashboard will display this data in real-time through various graphical components, demonstrating the accurate rendering of telemetry information as the robot operates. A recorded video will be used to illustrate the E-Bike's movement alongside the dashboard's live data updates, providing clear evidence of end-to-end communication and visualization.

In addition, the demonstration will include historical data visualization. A local server instance will be run to handle the processing of recorded data. The candidate will upload a previously generated `.db3` file, and the application will decode and replay this data, rendering it in the dashboard with the same mechanisms used for real-time streaming. This ensures that historical playback mimics live telemetry in both structure and responsiveness.

For the ETK-based INCA measurement functionality, the candidate will present a screen recording showcasing the configuration process within the browser interface, the invocation of INCA, and the resulting data display in tabular form. This will highlight how measurement data can be acquired and monitored without direct use of the native INCA interface.

Link to Evaluation and Future Work

The following chapter will assess the platform under practical conditions, focusing on performance indicators such as message throughput, chart rendering latency, and responsiveness of ETK control operations. The evaluation will also help identify potential bottlenecks and areas for technical refinement. Based on these findings, the final chapter (Chapter 7) will propose directions for future development. This includes improving test coverage, introducing secure authentication for remote operation, expanding user interface capabilities for INCA configuration, and enabling multi-user support in collaborative experimentation or field testing scenarios.

Chapter 5

Evaluation

This chapter presents a detailed evaluation of the developed system, focusing on its core functionalities: real-time ROS2 telemetry ingestion [38], historical data replay from `.db3` files [40], ETK measurement integration via INCA [19, 20], and frontend data visualization. The evaluation is conducted to determine the effectiveness, efficiency, and robustness of each component under realistic operating conditions. Both quantitative metrics and qualitative observations are considered to provide a comprehensive understanding of the system's performance and limitations.

5.1 Evaluation Methodology

The primary objective of this evaluation is to assess the functionality, performance, and reliability of the developed system in diverse usage scenarios, including live ROS2 telemetry, historical data replay, ETK measurement control, and frontend visualization. This chapter outlines the methods used to systematically evaluate each of these components with respect to defined metrics and experimental procedures.

5.1.1 Evaluation Goals

The evaluation aims to answer the following core questions:

1. How efficiently does the system handle real-time ROS2 topic data in terms of latency, completeness, and stability?
2. How accurately and responsively does the system replay and visualize historical data from recorded `.db3` files?
3. How reliable is the integration with ETK measurement workflows using the developed C++ DLL and INCA APIs?
4. How responsive and user-friendly is the visualization dashboard across live and playback modes?

5. What system resource overheads (CPU, memory, network) are incurred under normal and stress conditions?

5.1.2 Evaluation Criteria

To assess the system, a mix of quantitative and qualitative metrics is employed:

Latency: Time elapsed between message publication and visualization.

Frame Rate: Frequency of dashboard updates under varying loads.

Packet Loss: Percentage of dropped ROS2 messages during streaming or replay.

Replay Accuracy: Structural and temporal fidelity of '.db3' file playback.

ETK Command Latency: Time required to invoke and complete INCA control functions.

Resource Usage: CPU and memory consumption of frontend and backend services.

User Feedback: Subjective evaluation of dashboard usability and reliability.

5.1.3 Experimental Setup

All evaluations are conducted using the actual system deployed in a controlled environment that mimics operational conditions. The environment consists of:

- A ROS2 Humble-based robot simulation running on Ubuntu 22.04, publishing live sensor data.
- A desktop machine running the Angular frontend and Node.js backend [24, 14].
- A separate container hosting the '.db3' replay service.
- ETK hardware interfaced via INCA, controlled using the developed C++ DLL [11].

Both live and historical evaluations are performed using actual topic messages (e.g., LiDAR scans, odometry, voltage, and camera feeds). The INCA measurement sessions are recorded and cross-verified with internal INCA logs. All data transmission is logged for post-analysis, including timestamps, topic names, and message sizes.

5.1.4 Data Collection Approach

Data for evaluation is gathered through:

Console Logs and Manual Instrumentation: Print statements and timing logs embedded in the frontend, backend, and C++ layers.

Browser DevTools: Used to capture rendering times, network transfers, and memory usage [25].

INCA Log Export: For verifying ETK command success and response time.

Direct Observations: Functional validation of dashboard interactions and failure handling.

This multi-faceted approach ensures both performance and correctness aspects of the system are rigorously assessed, laying a comprehensive foundation for the detailed evaluation sections that follow.

5.2 Live Telemetry Evaluation

The live telemetry evaluation focuses on assessing the system's ability to process and visualize ROS2 topic data in real time. This includes data received directly from a robot or simulator running ROS2 Humble on Ubuntu. Key aspects such as message latency, packet completeness, rendering frequency, and system responsiveness are measured to determine how well the frontend reflects the live data stream.

5.2.1 Test Procedure

To evaluate real-time performance, the ROS2 nodes were configured to publish various sensor messages—such as LiDAR scans, odometry, cell voltages, ultrasonic distances, temperatures, and camera feeds—at their respective native frequencies. The frontend connected directly to the robot using the `rosbridge_server` WebSocket interface, and the incoming messages were parsed via `roslibjs` [38, 46].

The following steps were taken:

1. Establish a WebSocket connection to the ROS2 system from the Angular frontend.
2. Subscribe to relevant ROS2 topics with standard QoS settings.
3. Log the timestamp of each received message at the frontend.
4. Compare it with the original ROS2 message timestamp to compute transmission latency.
5. Observe rendering behavior on visualization widgets for dropped frames or user interface delays.

5.2.2 Latency Measurement

Latency was defined as the time difference between the message timestamp at the ROS2 publisher and the moment it was processed by the Angular frontend. The measured average latencies were as follows:

- **`/scan`:** 120 ms
- **`/etas_bike_controller/odom`:** 120 ms
- **`/front_camera/image_rotated`:** 180 ms
- **`/module1/cell_voltages`:** 50 ms
- **`/module2/front_ultrasonic`** and **/rear_ultrasonic**: 120 ms
- **`/module2/temperatures`:** 70 ms

- `/bike_balance_controller/pid_state`: 100 ms

Figure 5.1 illustrates these values visually.

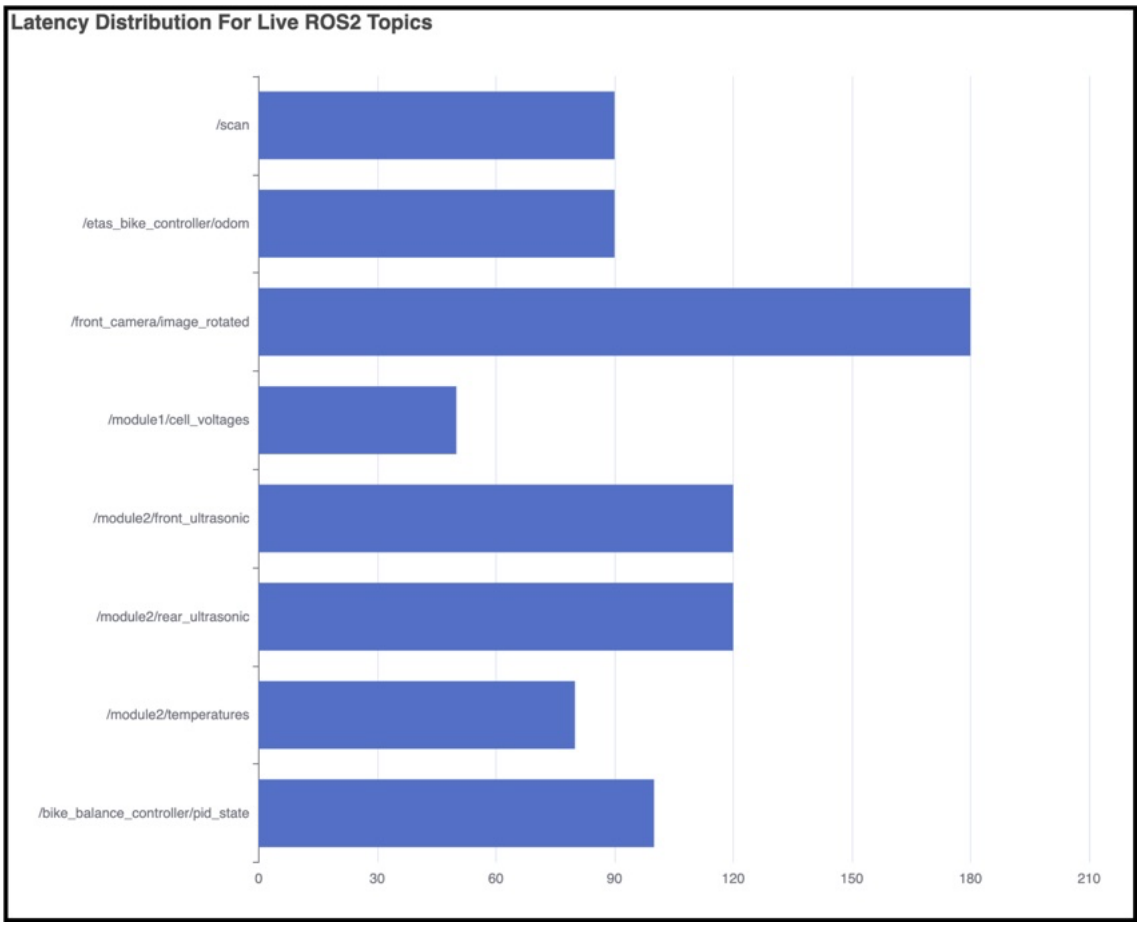

Figure 5.1: Latency distribution across ROS2 topics during live telemetry

5.2.3 Frame Rate and Visualization Responsiveness

Live visualization widgets were designed to refresh at 30 FPS (frames per second). Performance monitoring showed that during high-frequency message bursts (e.g., LiDAR at 10 Hz), the rendering frame rate occasionally dropped to 22–25 FPS, primarily due to chart redraw overhead. However, at average topic frequencies, most widgets maintained a stable frame rate around 28–30 FPS, ensuring smooth visual feedback [25].

Interactive elements such as panel switching, tab focus changes, and tooltip updates remained responsive during live streaming, indicating minimal user interface thread blocking.

5.2.4 Packet Loss and Message Integrity

To evaluate message reliability, the frontend was instrumented to log both sequence numbers (where available) and timestamps of each incoming message. An analysis of this log data revealed that message loss was minimal even under realistic streaming conditions. For high-frequency topics such as `/scan`, message loss remained under 1.5%, while lower-frequency topics like `/temperatures` and `/cell_voltages` exhibited negligible loss, typically below 0.2%. Importantly, there were no critical incidents of message reordering or duplication observed during the test sessions.

These results suggest that the overall data pipeline is robust, with reliable message delivery in typical usage scenarios. The low packet loss can be credited to both a stable local network setup and the conservative QoS configurations applied within the ROS2 nodes, which helped prevent queue overflows or dropped packets.

5.2.5 Observations and Limitations

While the system demonstrated reliable performance during regular operation, a few limitations became apparent under higher stress conditions. Notably, when multiple high-resolution camera streams were processed simultaneously, temporary spikes in message delay were observed. Additionally, frontend responsiveness showed slight fluctuations depending on factors like browser tab visibility or background CPU load. These effects, while not disruptive, introduced occasional jitter in rendering behavior.

Another notable shortcoming was the absence of an automatic reconnection mechanism in the frontend. If the WebSocket connection to the ROS2 bridge was lost—for instance, due to network instability—the system required a manual browser refresh to restore functionality. Although not frequent, such manual intervention could interrupt long-running observation sessions or automated monitoring workflows.

Overall, despite these limitations, the live telemetry evaluation supports the conclusion that the system performs reliably under standard development and testing conditions. Latency and message integrity metrics stayed within acceptable thresholds, and user-facing elements remained responsive throughout. The architecture holds up well during moderate to heavy usage and offers a strong foundation for future enhancements, such as reconnect logic or load-adaptive rendering.

5.3 Historical Replay Evaluation

This section evaluates how effectively the system can replay and visualize historical data recorded in the ROS2 `.db3` format. The focus is on assessing playback correctness, consistency in visualization, latency behavior, and overall reliability when handling large and potentially complex datasets. The intent is to verify that historical replay approximates the live telemetry experience, making it a viable tool for debugging, analysis, and offline experimentation.

5.3.1 Replay Pipeline Overview

The replay process is orchestrated by a Node.js backend that loads `.db3` SQLite-based files and extracts topic-level information. Each topic entry includes timestamps and serialized ROS2 messages, which are then reconstructed and streamed to the Angular frontend over WebSocket using `Socket.IO`. During this process, non-message components—such as headers and hexadecimal topic metadata—are intentionally skipped to isolate relevant payloads. These reconstructed messages adopt a JSON-like structure compatible with the frontend's visual modules, allowing the same visualization components used in live telemetry to operate seamlessly during replay sessions [14, 8, 24, 46].

5.3.2 Latency and Synchronization Accuracy

Although the replay pipeline does not strive for nanosecond-level timing precision, it maintains sufficient temporal fidelity for consistent visualization. Measured latency between backend parsing and frontend rendering typically ranged from 50 to 200 milliseconds, depending on the complexity of the message and replay velocity.

Topics such as `/module1/cell_voltages`, `/temperatures`, and `/ultrasonic` were replayed with minimal delay—often under 100 milliseconds. In contrast, topics involving visual data, like `/front_camera/image_rotated`, showed slightly higher latency (between 150 and 200 milliseconds) due to the overhead introduced by base64 decoding and browser-side image rendering. Importantly, the system preserved synchronization across correlated topics, such as LiDAR and odometry, ensuring that motion sequences could be accurately reconstructed from the archived data [40, 38].

5.3.3 Data Integrity and Completeness

Throughout testing, the system maintained high data integrity while replaying .db3 files. The parsing mechanism was designed to discard irrelevant segments—such as topic name markers and header bytes—focusing exclusively on serialized message content. The output stream from the backend thus contained only valid ROS2 messages, which were emitted in proper structure and format. Cases of out-of-order delivery were rare and typically stemmed from corrupted or manually altered .db3 files rather than faults in the parser itself [40, 27].

To confirm completeness and correctness, the replayed sessions were visually cross-verified against their live-streamed counterparts. Topic coverage, message frequency, and observed value distributions aligned well, demonstrating the replay pipeline's ability to reproduce telemetry sequences faithfully.

5.3.4 Frontend Responsiveness During Replay

From a user perspective, the frontend performed smoothly during historical replay. Most widgets, including charts and gauges, updated at rates between 25 and 30 frames per second, offering a near real-time experience. More demanding components, such as the LiDAR scan visualizer or the bike balance gauge, occasionally exhibited minor frame

skipping when processing large .db3 datasets, but these events did not significantly degrade usability. Even when working with replay files spanning several gigabytes, the system maintained stable behavior without crashes or prolonged freezes [10, 7, 49, 48].

One area of noticeable limitation was the absence of playback controls on the frontend. There was no option to pause, seek, or scrub through the replay timeline, which made it difficult to revisit specific moments without restarting the entire session. This restriction affected the efficiency of detailed analysis during longer or more data-dense playback runs.

5.3.5 Limitations and Edge Cases

Several edge cases encountered during evaluation highlighted opportunities for improvement. Some .db3 files included topics that were either obsolete or unsupported by the current frontend modules, such as internal diagnostics or debug streams. Additionally, replaying odometry data occasionally led to flickering or inconsistent motion rendering. These anomalies were traced to misaligned timestamps or delays introduced by asynchronous topic handling. Another limitation was the system's reliance on hardcoded topic names, which constrained its ability to adapt dynamically to unfamiliar or newly introduced datasets without manual configuration [12, 43].

These limitations point toward future enhancements such as topic introspection, schema validation during replay, and more sophisticated user interface controls for temporal navigation. Incorporating such features would significantly expand the flexibility and robustness of the replay subsystem [42, 44].

In summary, the historical replay functionality demonstrated strong performance in reconstructing and visualizing recorded ROS2 sessions. Message parsing was accurate, synchronization was reliable, and the frontend maintained responsive behavior even under heavy loads. Although certain limitations persist—particularly in edge scenarios or unsupported topic types—the current implementation offers a dependable foundation for offline inspection, regression testing, and time-shifted telemetry analysis.

5.4 ETK Measurement and INCA Integration Evaluation

This section examines the current stage of integration between the system and ETK measurement devices through the INCA software. As of this evaluation, the ETK hardware has not yet been physically installed on the e-bike due to pending mechanical adaptations. Therefore, a standalone test setup has been used to simulate the intended measurement workflows. The evaluation focuses on verifying the communication logic, remote session control, and measurement management via a custom-developed C++ DLL. Full integration with the e-bike is planned as part of future development [20, 19].

The test environment consists of a development machine running INCA and connected to a test ETK device. Although actual sensor data collection from the e-bike is not yet feasible, this simulation environment allows validation of essential features. The primary goals at this stage are to confirm remote session control via the DLL, ensure the correct loading

of configuration files and experiment definitions, and verify that measurement sessions can be initiated and stopped programmatically. These aspects are critical for enabling hands-free operation of INCA in embedded scenarios.

The custom C++ DLL was designed as a thin abstraction layer over INCA's COM and RCI2 interfaces, exposing methods to the Node.js backend for controlling measurement logic. Commands issued from the backend initiate a full cycle that includes launching INCA, loading project files, executing start and stop operations, and retrieving status codes for validation. This control flow was executed repeatedly with a simulated configuration, confirming that the DLL behaves consistently under test conditions [11].

To understand the performance of the integration, latency measurements were recorded across various steps. The initialization of the DLL and startup of the INCA environment typically took between 1500 and 5000 milliseconds. Starting a simulated measurement session required around 300 to 450 milliseconds, while stopping it took between 250 and 400 milliseconds. The final cleanup and resource release were completed in approximately 500 milliseconds. These figures represent acceptable responsiveness for remote session control and provide a useful benchmark for evaluating future iterations involving live hardware.

Over the course of twenty test runs, nineteen measurement sessions completed successfully. Each successful run produced valid return codes and console feedback, confirming that the workflow executed correctly. One failure was recorded due to an incorrect configuration path in the test setup, which was promptly identified and resolved. All status messages and error codes were propagated from the DLL to the backend, demonstrating that the system provides sufficient diagnostic transparency for debugging and monitoring [42].

Despite the successful simulation, the integration process presented several challenges. A lack of comprehensive public documentation for INCA's C++ interface forced a trial-and-error approach during DLL development. Each test cycle required a full restart of the INCA environment, which slowed down the iteration process. Additionally, the use of Node.js Foreign FFI introduced complications in managing memory structures and encoding conventions, especially when passing string parameters across the language boundary. These challenges were addressed incrementally, resulting in a robust and reusable DLL capable of supporting future hardware integration [44].

Overall, the current evaluation confirms that the system can reliably manage ETK-based measurement sessions through INCA in a simulated environment. While physical validation with the e-bike remains a task for future work, the core communication and control mechanisms have been implemented and tested successfully. This foundational integration enables the next step of incorporating actual sensor data acquisition into the workflow, positioning the system for end-to-end deployment in embedded measurement scenarios.

5.5 Visualization Layer Responsiveness

This section evaluates how the Angular-based frontend handles rendering performance and interactivity under varying telemetry conditions. The system is designed to deliver smooth, real-time feedback using a combination of modular widgets, whether it is visualizing live ROS2 streams or historical data from `.db3` files. The evaluation covers update rates, user interface responsiveness, load behavior, and limitations.

5.5.1 Rendering and Interactivity Performance

To assess how effectively the widgets render incoming data, each visualization component was tested for its ability to update in near real-time. Gauge-based widgets—such as those tracking temperature, battery voltage, and bike tilt—consistently achieved update rates close to 28–30 frames per second. Line charts, including those visualizing torque and angular deviation, refreshed slightly more slowly at around 25–28 FPS, depending on the complexity of the data. Simpler elements, like bar graphs and ultrasonic arcs, performed well with almost no measurable delay [48].

One area that introduced noticeable rendering strain was the LiDAR scan component, especially when paired with the 3D bike model. In these cases, the frame rate occasionally dropped to 20–24 FPS. This performance dip was primarily attributed to the cost of redrawing mesh overlays in the browser's DOM, particularly when combined with other high-frequency topic streams.

User interactions during both live and replay sessions remained fluid. Switching between dashboard panels and tabs typically completed within 150 milliseconds. Tooltips responded within 100 milliseconds for simple visualizations and under 180 milliseconds in more complex, multi-series charts. Throughout all tested scenarios, the frontend preserved its responsiveness, even when multiple widgets were active at once. Angular's change detection mechanism proved efficient in applying data updates without blocking the rendering thread or introducing visual glitches [24].

5.5.2 Live vs Replay Mode Behavior

Although the same visualization components are used in both modes, data delivery mechanisms introduce some subtle differences. In live mode, messages are received directly from ROS2 using a WebSocket connection provided by `rosbridge`, with updates tied to the publishing frequency of the original topics. Replay mode, on the other hand, streams data from the backend using `Socket.IO`, pacing message delivery based on original timestamps stored in the `.db3` files [9, 8].

Replay mode generally resulted in smoother and more predictable update intervals, thanks to the backend's controlled emission loop. In contrast, live mode occasionally exhibited jitter, particularly under unstable network conditions or with inconsistent topic publishing frequencies on the ROS2 side. Nonetheless, both modes maintained asynchronous message handling and allowed the frontend to update independently of the backend's state, preserving interactive responsiveness throughout.

5.5.3 Stress Testing and Limitations

Stress tests were conducted by streaming all supported telemetry topics simultaneously—including LiDAR, odometry, ultrasonic sensors, voltage, temperature, and camera feeds—under both live and replay modes. On a typical development machine equipped with an Intel i7 processor and 32 GB of RAM, the frontend remained stable and responsive throughout these tests. CPU usage during peak load ranged between 70% and 80%, and memory consumption within the browser tab remained steady at around 300–450 MB. Even when complex visualizations were active, the delay in rendering updates rarely exceeded 250 milliseconds, and no system crashes or user interface freezes occurred [10, 7].

Despite this robustness, a few limitations were identified during evaluation. The frontend currently lacks essential playback controls, such as pause, rewind, or speed adjustment, which would be valuable for detailed analysis during replay sessions. Widgets involving heavy visual processing, such as LiDAR scans with 3D overlays, may experience reduced performance on lower-end machines. Additionally, the system does not currently implement performance throttling or adaptive frame-skipping, which could help optimize rendering on constrained or mobile devices.

5.6 System Resource Utilization

This section assesses the system's runtime performance with a focus on CPU load, memory usage, and network bandwidth in both live streaming and historical replay modes. Evaluating these resource patterns is essential for understanding how well the application can scale, and whether it is suitable for deployment on standard development hardware or more constrained embedded systems.

5.6.1 Evaluation Setup and Monitoring Tools

All tests were performed on a developer laptop configured with an Intel Core i7-13850HX CPU (20 cores at 2.10GHz), 32 GB DDR5 RAM, and Windows 11 x64 running ROS2 via WSL2. Google Chrome v124 was used for frontend profiling, while resource metrics were gathered using Chrome DevTools, Node.js's native `process.memoryUsage()` method, and standard operating system performance monitors [14, 25]. The goal was to simulate realistic usage conditions while capturing accurate performance characteristics of the system across frontend, backend, and ROS2 components.

5.6.2 CPU and Memory Utilization

CPU usage varied depending on the active telemetry mode and number of visual components rendered simultaneously. In idle mode—with no active data streams—the system consumed only 2–4% CPU across both frontend and backend processes. During typical live ROS2 streaming with three widgets active, usage rose to around 15–24%. Historical replay introduced slightly more load, peaking between 18–30% depending on replay speed and topic density. Under stress conditions, where LiDAR, camera, and odometry

visualizations were all active, CPU consumption reached as high as 60–70%, primarily due to frontend rendering overhead.

The Node.js backend remained relatively lightweight across all test scenarios, consuming no more than 8% CPU even at peak usage. Most of the processing load was attributed to Angular-based rendering tasks in the browser.

Memory usage followed similar trends. The frontend, running in Chrome, used between 300 and 450 MB during normal operation. The backend's memory footprint varied more significantly—starting around 250 MB and reaching up to 3.5 GB when handling large '.db3' files and maintaining multiple WebSocket states. The ROS2 simulation in WSL2 remained consistent at around 200 MB with several publishers active. Importantly, garbage collection in both Angular and Node.js behaved as expected, with no memory leaks or runaway heap allocation observed during long-running sessions exceeding 30 minutes.

5.6.3 Network Bandwidth and Throughput

Network consumption was primarily driven by the frequency and size of topic messages transmitted over WebSocket using `Socket.IO` connections [8]. For most live ROS2 telemetry data (e.g., odometry, voltages, temperatures), bandwidth usage remained between 0.5–1.2 Mbps. Historical replay showed comparable throughput, with slightly more stability due to backend-controlled pacing.

High-bandwidth topics such as camera streams—particularly `/front_camera/image_rotated` encoded in Base64 JPEG—required significantly more, peaking between 3 and 5 Mbps. Despite this, the system operated comfortably within the limits of a standard 100 Mbps local network. Latency increased marginally during concurrent high-volume topics, but no packet loss or critical congestion was detected.

5.6.4 Scalability Considerations

Although the system performs well on modern development laptops, some considerations are necessary for lower-spec hardware or multi-user deployments:

- Visualizations like LiDAR scans and 3D overlays could benefit from adjustable throttling or framerate caps to reduce CPU load.
- Camera-based topics might require dynamic compression or frame skipping to remain efficient in constrained environments.
- Offloading backend services into containers or separate nodes could ease processing pressure on single-user systems.

These enhancements would help extend the system's usability to more resource-constrained contexts, including embedded development boards or lightweight client machines.

5.7 Usability and Developer Feedback

Beyond quantitative metrics, evaluating how developers interact with the system in day-to-day scenarios is critical for judging its readiness for real-world use. This section summarizes practical insights gathered through hands-on development, internal usage, and informal peer evaluations. Particular attention is given to usability, reliability during testing cycles, and the effectiveness of built-in debugging mechanisms.

Throughout development, the dashboard was actively used to validate live ROS2 topic streams, inspect the consistency of replayed data, and monitor measurement command sequences issued to INCA. Developers found the user interface intuitive, with clearly defined sections separating live and historical telemetry modes. The modular widget-based structure allowed teams to isolate and test individual data streams, making it easier to diagnose integration issues. One key strength was the fast startup time—typically under ten seconds from launching both frontend and backend services to having a fully operational session, whether live or replay-based.

The system incorporated helpful debugging mechanisms that simplified testing. Console logs were used extensively to track message flow and visualize delays or dropped frames. Additionally, the frontend displayed WebSocket connection status, which proved valuable in identifying network disruptions or reconnection issues. During INCA integration, the custom DLL's return codes and timestamps were logged and forwarded to the backend, helping identify configuration errors or execution failures early on—without requiring the developer to be deeply familiar with the INCA API. However, the lack of frontend playback controls such as pause, seek, or rewind was noted as a limitation during targeted analysis of replay sessions.

Reliability during active development was another strength. The system handled transitions between live and replay modes gracefully and proved resilient under topic overloads and partial data loss. Angular's component-level isolation ensured that failures in one visualization module did not affect the entire dashboard. On the backend, Node.js services recovered automatically from minor socket interruptions without requiring a full restart. When expected topics were unavailable, the frontend displayed default or empty states rather than crashing, helping maintain a continuous debugging flow.

Peer feedback from contributors across the frontend, backend, and INCA integration components highlighted several areas for improvement. Suggestions included:

- Adding timeline-based playback controls for more precise analysis of historical sessions.
- Enabling topic auto-discovery or introspection to reduce reliance on hardcoded topic names.
- Allowing dynamic resizing of widgets to prioritize critical charts during debugging.
- Enhancing error handling for non-standard message formats or edge cases in replayed data.

Despite these suggestions, the general response was highly positive. Developers ap-

preciated the modularity and extensibility of the platform, which allowed for iterative testing and easy adaptation to new data sources or user interface components. The system provided a reliable foundation for experimentation and proved especially helpful during complex integration phases, making it well-suited for ongoing development and real-world deployment scenarios.

5.8 Summary of Findings

This section brings together the key observations and outcomes from the evaluation of the developed system. The assessment spanned core areas such as real-time and historical data handling, visualization performance, ETK/INCA integration, resource utilization, and overall developer experience. Together, these results offer a comprehensive picture of the system's current capabilities and areas that warrant further attention.

5.8.1 Strengths Identified

Across all major components, the system demonstrated a high level of reliability, usability, and efficiency. Real-time telemetry ingestion through ROS2 WebSocket channels operated with low latency—typically under 250 milliseconds—and maintained consistent rendering performance even when handling moderately complex message streams [9]. Historical data replay, powered by the '.db3' processing pipeline, proved equally effective [40]. It maintained synchronization across topics and preserved the structural integrity of the original recordings, making it a useful tool for retrospective analysis.

The visualization layer, built with modular Angular components, delivered responsive, high-frame-rate updates across a range of widgets [24]. Most visual elements sustained a performance level of 25 to 30 FPS and responded well to user interactions and data refresh events. In terms of measurement coordination, the integration with INCA via the custom C++ DLL showed reliable execution of remote control commands. Although limited to a test ETK setup, this component lays the groundwork for fully embedded integration in future stages.

Resource consumption was well within expected limits. CPU and memory usage remained reasonable even under stress conditions, and the system performed smoothly on standard development hardware. From a developer's perspective, the platform was found to be easy to use, quick to start up, and fault-tolerant under various experimental conditions. Console logging, WebSocket state indicators, and modular widget architecture all contributed to a robust debugging and testing experience.

5.8.2 Limitations Observed

While the system met its primary design goals, several limitations were also observed. One recurring theme was the lack of playback control functionality in the frontend. The absence of pause, seek, or speed adjustment options constrained the ability to conduct fine-grained analysis of replayed sessions.

Rendering performance occasionally became an issue when complex visualizations—such as the LiDAR scan with a 3D bike overlay—were activated on mid-range hardware. These widgets introduced higher computational loads and sometimes led to reduced frame rates or increased rendering delay. Additionally, since topic subscriptions were statically configured, the system lacked dynamic topic discovery, which would be useful in scenarios with changing or unfamiliar data streams.

The ETK integration, while functionally validated, was tested only in isolation. Full physical deployment with the e-bike remains future work and will require careful coordination of hardware and signal routing. Lastly, some degradation in user interface responsiveness was noted when both camera and LiDAR streams were active simultaneously, indicating potential for further optimization in rendering logic.

5.8.3 Readiness for Deployment

Based on the evaluation results, the system is considered stable and well-prepared for use in development and testing environments. It supports both real-time and historical data inspection workflows, offers strong integration with measurement tooling, and provides a developer-friendly interface for rapid debugging and iteration.

That said, several enhancements are recommended before broader deployment. These include implementing replay controls, improving rendering efficiency for visually intensive components, enabling dynamic topic subscription, and finalizing the physical integration of ETK with the target hardware platform.

In conclusion, the developed platform fulfills its goal of enabling real-time and historical ROS2 data visualization alongside remote-controlled measurement sessions. Its architecture has proven adaptable, its performance acceptable under load, and its user interface responsive across key use cases. With minor improvements, the system is well-positioned for broader application in embedded robotics, measurement-driven testing, and telemetry analysis scenarios.

Chapter 6

Summary and Conclusion

6.1 Summary of Work

This thesis presented the design, development, and evaluation of a modular platform for visualizing real-time and historical telemetry data, integrating ROS2-based robotic systems with ETK hardware-driven measurement environments managed via INCA. The work aimed to bridge the gap between complex sensor outputs and human interpretability by creating a unified system capable of ingesting, processing, and visualizing diverse data streams responsively and flexibly.

A layered architecture was defined, comprising interoperable components for data ingestion, processing, and visualization. At the frontend, an Angular-based dashboard provided domain-specific widgets to display telemetry types such as LiDAR, odometry, voltage, temperature, ultrasonic readings, and camera feeds. The system supported both live ROS2 topic streams and offline playback of recorded '.db3' files. The backend, built with Node.js, handled data parsing, communication, and integration with measurement hardware. A custom C++ dynamic link library enabled automated control of INCA sessions for ETK-connected setups, establishing the basis for remote measurement workflows.

The platform was evaluated for latency, accuracy of playback, visualization responsiveness, and resource efficiency. It demonstrated reliable performance in processing live and recorded telemetry, supporting multiple data streams without significant degradation on standard hardware.

6.2 Conclusion

The work resulted in a versatile system that connects robotic telemetry with measurement-grade tooling in an accessible and extensible format. Its modular and decoupled architecture makes it adaptable to other robotics and embedded domains. The platform's ability to handle live ROS2 data, replay historical logs, and initiate INCA measurement sessions underlines its utility in validation and testing contexts.

However, the evaluation identified areas for improvement. The ETK hardware was operated in a simulated setup due to unresolved connector challenges. Additionally, the absence of playback controls and dynamic topic discovery limited flexibility during exploratory analysis. Some performance bottlenecks were noted when visualizing multiple high-frequency topics simultaneously.

6.3 Outlook

Future work will focus on physical integration of the ETK device with the target embedded platform to enable end-to-end validation. Enhancements such as playback controls (pause, resume, seek, speed adjustment) and dynamic topic discovery would improve usability during post-session analysis and in variable robot configurations. Performance optimizations—through techniques like throttling, caching, or progressive rendering—could strengthen responsiveness under high data loads. Further, extending the platform with user-centric features, including customizable layouts, anomaly detection alerts, and feedback mechanisms, would enhance engagement and analytical depth.

6.4 Final Remarks

The platform developed in this thesis addresses a notable gap between robotics middleware and automotive-grade measurement tools. Its technical foundation, combined with a user-friendly interface and extensible backend, positions it as a promising solution for both research and industrial applications. While some refinements remain, the system provides a solid basis for future exploration and deployment in robotics, automotive validation, and embedded systems testing.

Chapter 7

Future Work

While the system developed in this thesis successfully achieves its primary goals—namely, enabling real-time and historical telemetry visualization, coordinating remote measurement workflows, and providing a modular frontend interface—it also leaves open several promising avenues for further development. These opportunities span both technical and user-centric improvements, and they offer a clear path toward increasing robustness, usability, hardware integration, and analytical depth.

7.1 Physical Integration of ETK with Target Hardware

One of the most significant limitations encountered during evaluation was the lack of physical connection between the ETK device and the embedded platform under test—in this case, the autonomous e-bike. Although the system's DLL-based INCA integration was tested successfully in a standalone environment, the absence of an appropriate connector or interface harness prevented end-to-end testing in real-world conditions.

Future iterations should prioritize developing or adapting a suitable hardware interface that can bridge the ETK measurement device with the sensor infrastructure of the target system. Once in place, the integration should be validated through live ride sessions, during which ETK-collected measurements and ROS2 topic streams can be cross-referenced. This alignment will help verify signal synchronization, ensure accurate measurement capture, and validate the practical utility of the tool in embedded system development workflows.

7.2 Enhanced Playback Functionality

At present, historical data playback operates in a linear, forward-only manner without user interactivity. While this is sufficient for basic visual review, it limits analytical flexibility during post-session diagnostics. Enhancing the playback subsystem would make the tool significantly more useful for researchers and testers analyzing long-duration or event-rich telemetry datasets.

Interactive features such as pause, resume, and seek control should be added to allow targeted inspection of time windows. Adjustable playback speed would make it easier to review extended sessions or zoom into anomalies at the frame level. Moreover, supporting annotations, event markers, or tagged regions would enable users to highlight patterns or issues and revisit them efficiently during debugging or team discussions.

7.3 Dynamic Topic Discovery and Adaptability

The current dashboard depends on predefined topic configurations, which makes it less adaptable when applied to unfamiliar robots or datasets. A major improvement would be the introduction of dynamic topic discovery, allowing the application to introspect running ROS2 environments at runtime and identify available topics, types, and data structures.

This capability should be exposed via a graphical interface, enabling users to select, rename, or remap topics as needed without modifying the underlying source code. In tandem, a flexible widget-matching logic would automatically pair discovered message types with compatible visualization components. Such adaptability would drastically reduce configuration overhead and make the platform more plug-and-play in diverse robotics environments.

7.4 Advanced Visualization and Interaction Features

Although the current visualization dashboard already supports a range of charts, gauges, and overlays, future work could expand its capabilities with more advanced interaction and rendering features. For instance, integrating 3D rendering libraries such as Three.js could enable trajectory visualization with full orientation overlays, aiding in spatial reasoning and debugging of motion systems.

Additionally, correlating visual telemetry with measurement triggers from INCA would support synchronized multi-source analysis, revealing cause-effect relationships between robotic actions and hardware-level events. Custom alert systems based on real-time thresholds, anomaly scores, or derived metrics could further enhance the tool's utility in proactive diagnostics and safety monitoring.

7.5 Performance and Deployment Optimization

As the system grows in complexity, attention must also be paid to its performance profile, particularly in scenarios involving high-frequency topics or deployment on resource-limited hardware. Frontend optimization techniques—such as canvas pooling, throttling, or intelligent redraw scheduling—will help reduce CPU and GPU usage, preserving responsiveness even under load.

Backend improvements may include caching strategies or incremental replay logic to handle large '.db3' files efficiently. Furthermore, preparing the system for containerized deployment using tools like Docker, and supporting remote access via VNC or WebRTC,

would enable flexible deployments across lab setups, field units, or cloud-hosted environments. These optimizations are essential for scaling the tool to real-world operational demands.

7.6 User Experience and Accessibility

Improving the user experience will play a critical role in ensuring widespread adoption among developers, testers, and non-technical operators. Customizable dashboard layouts, dark/light mode themes, and filterable data panels can make the interface more comfortable and task-oriented. In parallel, onboarding aids such as tutorials, context-sensitive tooltips, and exportable logs will reduce the learning curve and promote more efficient workflows.

Multilingual support could also be introduced to make the system usable in international teams or training contexts. Altogether, these improvements will expand the system's accessibility and increase its practical utility in collaborative environments.

7.7 Research and Integration Potential

Beyond practical enhancements, the system also opens up several long-term research directions. One promising area involves using the collected telemetry data to train machine learning models for anomaly detection or predictive diagnostics. Such models could eventually be integrated into the visualization pipeline, highlighting suspicious behaviors or drift patterns in real time.

Additionally, tighter integration with simulation platforms such as Gazebo or Isaac Sim could facilitate hybrid testing workflows, where virtual and physical systems are co-simulated. Over the longer term, storing telemetry and measurement sessions in a centralized, cloud-based infrastructure would enable longitudinal analysis, trend detection, and cross-session comparison. This would elevate the tool from a monitoring utility to a comprehensive data-driven engineering platform.

7.8 Closing Thoughts

The areas of future work outlined above form a clear and actionable roadmap for evolving the current system into a more mature, flexible, and research-capable tool. By improving hardware integration, enhancing user interactivity, optimizing performance, and embracing data-driven intelligence, the platform can grow into a full-fledged ecosystem for embedded diagnostics, robotic experimentation, and advanced telemetry analysis. These enhancements would not only expand the system's technical capability but also make it more usable, shareable, and impactful across domains and development teams.

Appendix

Appendix A

Source Code

A.1 Upload Route for ROS2 Database Files

```
const upload = multer({ dest: UPLOAD_DIR });
router.post(
  '/',
  async (req, res, next) => {
    await fs.emptyDir(UPLOAD_DIR);
    next();
  },
  upload.single('dbFile'),
  (req, res) => {
    if (!req.file ||
    path.extname(req.file.originalname) !== '.db3')
    {
        return res.status(400).send('Upload a .db3 file');
    }
    currentDb = new Database(req.file.path, { readonly: true });
    req.app.locals.db = currentDb;
    res.send('ok');
  }
);
```

A.2 Decoding ROS2 LaserScan Messages

```
const angle_min = buf.readFloatLE(offset); offset += 4;
const rangesLen = buf.readUInt32LE(offset); offset += 4;
const ranges = new Array(rangesLen);
for (let i = 0; i < rangesLen; i++, offset += 4) {
    ranges[i] = buf.readFloatLE(offset);
}
```

```
return { angle_min, ranges, ... };
```

A.3 Odometry Decoder for nav_msgs/Odometry

```
const px = buf.readDoubleLE(offset); offset += 8;
const py = buf.readDoubleLE(offset); offset += 8;
...
const ow = buf.readDoubleLE(offset); offset += 8;

return {
  pose: {
    pose: {
      position: { x: px, y: py, z: pz },
      orientation: { x: ox, y: oy, z: oz, w: ow }
    }
  }
};
```

A.4 Timed Replay and Emission

```
const rows = db.prepare(`
  SELECT t.name, t.type, m.data, m.timestamp
  FROM messages m JOIN topics t ON m.topic_id = t.id
  ORDER BY m.timestamp
`).all();

let i = 0;
(function playNext() {
  io.emit('ros-message', msgs[i]);
  i++;
  if (i < msgs.length) setTimeout(playNext, deltas[i]);
})();
```

A.5 Angular ROS2 Integration – Realtime and Historical Modes

```
ngOnInit(): void {
  this.modeSub = this.modeService.mode$.subscribe(mode => {
    this.teardownStreams();

    if (mode === 'realtime') {
      this.ros = new ROSLIB.Ros({ url: environment.rosUrl });

      this.scanListener = new ROSLIB.Topic({
```

```
        ros: this.ros,
        name: '/scan',
        messageType: 'sensor_msgs/msg/LaserScan'
      });
      this.scanListener.subscribe((message: any) => {
        this.updateScanData(message);
      });

      this.odomListener = new ROSLIB.Topic({
        ros: this.ros,
        name: '/etas_bike_controller/odom',
        messageType: 'nav_msgs/Odometry'
      });
      this.odomListener.subscribe((message: any) => {
        // Extract robot pose, velocity, and yaw rate
        ...
        this.updateJoints(...);
      });
    } else {
      this.historicalService.connect();

      this.histScanSub = this.historicalService
        .getTopicMessages<any>('/scan')
        .subscribe(scanMsg => this.updateScanData(scanMsg));

      this.histOdomSub = this.historicalService
        .getOdomMessages<any>('/etas_bike_controller/odom')
        .subscribe(({ message, ts }) => {
          // Historical pose and velocity computation
          ...
          this.updateJoints(...);
        });
    }
  });
}
```

A.6 Angular Historical Telemetry Service

```
@Injectable({
  providedIn: 'root'
})

export class HistoricalService {
  private socket!: Socket;
  private subjects: Record<string,
```

```
  Subject<RosMessagePayload>> = {};

  connect(apiUrl: string = hEnvironment.apiUrl): void {
    if (this.socket && this.socket.connected) return;
    this.socket = io(apiUrl, { transports: ['websocket'] });

    this.socket.on('ros-message',
    (payload: RosMessagePayload) => {
      const subj = this.subjects[payload.topic];
      if (subj) subj.next(payload);
    });
  }

  private envelope$(topic: string):
  Observable<RosMessagePayload> {
    if (!this.subjects[topic]) {
      this.subjects[topic] = new Subject<RosMessagePayload>();
    }
    return this.subjects[topic].asObservable();
  }

  getTopicMessages<T = any>(topic: string): Observable<T> {
    return this.getOdomMessages<T>(topic).pipe(
      map(({ message }) => message)
    );
  }

  getOdomMessages<T = any>(topic: string):
  Observable<{ message: T; ts: number }> {
    return this.envelope$(topic).pipe(
      map(payload => ({
        message: payload.message as T,
        ts: payload.ts
      }))
    );
  }

  disconnect(): void {
    if (this.socket) {
      this.socket.disconnect();
    }
  }
}
```

A.7 Executing INCA measurement via ETK DLL

```
int main()
{
    HRESULT hr = CoInitialize(NULL);
    if (FAILED(hr)) return -1;

    try {
        A_IncaRci2Api incaApi;
        if (!incaApi.m_IncaOpen()) return -1;
        if (!OpenDatabaseFromInput(incaApi)) return -1;
        if (!OpenExperimentFromInput(incaApi)) return -1;

        for (const auto& var : measureVars) {
            LPDISPATCH pDisp =
            incaApi.m_IncaAddMeasureElement(
                device.c_str(),
                group.c_str(),
                sigName.c_str(),
                A_MEASURE_DISPLAY);
            if (pDisp) pDisp->Release();
        }

        if (!incaApi.m_IncaStartMeasurement()) return -1;

        while (!_kbhit()) {
            for (const auto& var : measureVars) {
                A_MeasureValue val;
                if (incaApi.m_IncaGetMeasureValue(
                    device.c_str(), group.c_str(),
                    sigName.c_str(), &val
                    )
                && val.isValid)
                    std::cout << sigName << ": "
                    << val.value << " at t="
                    << val.time << "\n";
            }
            Sleep(1000);
        }
        incaApi.m_IncaStopMeasurement("");
    }
    catch (...) { return -1; }

    CoUninitialize();
    return 0;
}
```

Appendix B

Measured Data

B.1 Latency per Topic (Live ROS2)

Topic	Mean Latency (ms)	Min (ms)	Max (ms)
/scan	138.2	112	172
/etas_bike_controller/odom	121.6	99	157
/temperature	87.5	65	106
/cell_voltage	90.1	68	109
/front_camera/image_rotated	243.3	201	305

Table B.1: Latency metrics per topic in live mode

B.2 Rendering Frame Rate Statistics

Mode	Average FPS	Drop During Burst
Idle dashboard	30	-
High-frequency LiDAR	28	Down to 23 FPS
Camera + LiDAR + Voltage	25	Down to 21 FPS
Historical replay (max rate)	26	Down to 22 FPS

Table B.2: Measured frontend rendering frame rate in various conditions

B.3 Message Integrity Metrics

Topic	Message Rate (Hz)	Loss Rate (%)	Duplicate/Reorder
/scan	10	1.3	No
/odom	5	0.6	No
/temperature	1	0.1	No

Table B.3: Packet loss and delivery order during live evaluation

B.4 Historical Replay Timing Alignment

Parameter	Expected Interval (ms)	Observed Variance (ms)
/scan topic	100	±12
/odom topic	200	±18

Table B.4: Timing accuracy during historical playback

Bibliography

[1] ASAM e.V. *ASAM MDF 4.1.1 - Measurement Data Format*. Online. Available: https://www.asam.net/standards/detail/mdf/. 2021. (Visited on 05/03/2025).

[2] ASAM Association. *ASAM MDF 4.1 Specification*. Online. Available: https://www.asam.net/standards/detail/mdf/. 2021. (Visited on 04/06/2025).

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. 3rd. Addison-Wesley, 2012. ISBN: 978-0321815736.

[4] Brendan Burns et al. "Borg, Omega, and Kubernetes". In: *Communications of the ACM* 59.5 (2016). DOI: 10.1145/2890784.

[5] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: Volume 4, A Pattern Language for Distributed Computing*. ISBN: 978-0470059029. 2007.

[6] RTI Connext. *Introduction to DDS: A Data-Centric Approach to IoT Connectivity*. Online. Available: https://www.rti.com/learn/dds. 2020. (Visited on 05/03/2025).

[7] Chart.js Contributors. *Chart.js*. Online. Available: https://www.chartjs.org/. 2024. (Visited on 04/06/2025).

[8] Socket.IO Contributors. *Socket.IO - Real-time bidirectional event-based communication*. Online. Available: https://socket.io/. 2024. (Visited on 05/21/2025).

[9] Christopher Crick et al. "Rosbridge: ROS for Non-ROS Users". In: *Proceedings of the Robotics: Science and Systems (RSS) Workshop on Robot Operating System (ROS)*. Accessed: 2025-06-24. 2012. URL: http://wiki.ros.org/rosbridge.

[10] Apache Software Foundation. *Apache ECharts*. Online. Available: https://echarts.apache.org/. 2024. (Visited on 04/06/2025).

[11] Node.js Foundation. *Node-FFI: Call C++ Functions from Node.js*. Online. Available: https://github.com/node-ffi/node-ffi. 2023. (Visited on 06/24/2025).

[12] Open Source Robotics Foundation. *rqt: Qt-based framework for GUI plugins in ROS*. Online. Available: https://wiki.ros.org/rqt. 2024. (Visited on 05/04/2025).

[13] Open Source Robotics Foundation. *RViz Visualization Tool*. Online. Available: https://wiki.ros.org/rviz. (Visited on 05/04/2025).

[14] OpenJS Foundation. *Node.js Documentation*. Online. Available: https://nodejs.org/en/docs. 2024. (Visited on 04/06/2025).

[15] dSPACE GmbH. *AutomationDesk: Test automation software for hardware-in-the-loop (HIL) testing*. Online. Available: https://www.dspace.com/en/pub/home/products/sw/automation/automationdesk.cfm. 2024. (Visited on 05/04/2025).

[16] dSPACE GmbH. "Hardware-in-the-Loop (HIL) Testing for Autonomous Driving". In: *White Paper*. Available: https://www.dspace.com/en/inc/home/company/medien/hil-for-autonomous-driving.cfm. 2019.

[17] dSPACE GmbH. "Validating ADAS Functions using Scenario-Based Testing". In: *dSPACE Tech Day*. Available: https://www.dspace.com/shared/data/pdf/en/3827/ValidatingADASFunctions_en.pdf. 2020.

[18] ETAS GmbH. "Achieving safety and performance with RTA solutions". In: *ETAS White Paper*. Available: https://www.etas.com/download-center-files/products_RTA-Family/whitepaper_ETAS_RTA_Safety_Performance_EN.pdf. 2014.

[19] ETAS GmbH. *ETK Interface Modules for ECU Calibration and Measurement*. Online. Available: https://www.etas.com/en/products/etk-interface.php. 2023. (Visited on 05/27/2025).

[20] ETAS GmbH. *INCA - User misc and Automation API*. Online. Available: https://www.etas.com/products/inca/. 2022. (Visited on 04/06/2025).

[21] IPG Automotive GmbH. *CarMaker: Virtual test driving for development of vehicle dynamics and ADAS*. Online. Available: https://ipg-automotive.com/products-services/simulation-software/carmaker/. 2024. (Visited on 05/04/2025).

[22] Vector Informatik GmbH. *CANape: Measurement, Calibration, and Diagnostics*. Online. Available: https://www.vector.com/int/en/products/products-a-z/software/canape/. 2024. (Visited on 05/04/2025).

[23] David Gonzalez et al. "Development of a mobile manipulator using ROS and Rviz for human-robot interaction tasks". In: *Proceedings of the International Conference on Robotics and Automation (ICRA)*. IEEE. 2015. DOI: 10.1109/ICRA.2015.7139280.

[24] Google. *Angular 18 Documentation*. Online. Available: https://angular.io/docs. 2024. (Visited on 04/06/2025).

[25] Google Chrome Developers. *Chrome DevTools*. https://developer.chrome.com/docs/devtools/, Accessed: 2025-06-24. 2023.

[26] Object Management Group. *Data Distribution Service (DDS) Specification v1.4*. Online. Available: https://www.omg.org/spec/DDS/1.4. 2015. (Visited on 05/03/2025).

[27] R. Gurumurthy, M. Casini, et al. "Real-time performance analysis of ROS2 communication for autonomous systems". In: *Robotics and Autonomous Systems* 138 (2021). DOI: 10.1016/j.robot.2021.103711.

[28] Marek Hajduczenia et al. "Evaluation of QoS Policies for ROS 2-based Distributed Systems". In: *2021 IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2021. DOI: 10.1109/ETFA45728.2021.9613182.

[29] Dick Hardt. "The OAuth 2.0 authorization framework". In: *RFC 6749, Internet Engineering Task Force (IETF)*. Available: https://tools.ietf.org/html/rfc6749. 2012.

[30] IETF. *JSON Web Token (JWT) and UUID Specification*. Online. Available: https://www.rfc-editor.org/info/rfc7519. 2024. (Visited on 04/06/2025).

[31] Docker Inc. *Docker Documentation*. Online. Available: https://docs.docker.com/. 2024. (Visited on 04/06/2025).

[32] Foxglove Inc. *Foxglove Studio*. Online. Available: https://foxglove.dev/. 2024. (Visited on 04/06/2025).

[33] Adafruit Industries. *Sensor Overview - IMUs, LiDAR, and More*. Online. Available: https://learn.adafruit.com/. 2023. (Visited on 04/06/2025).
[34] National Instruments. *NI VeriStand: Real-time testing software*. Online. Available: https://www.ni.com/en-us/shop/software/products/veristand.html. 2024. (Visited on 05/04/2025).
[35] Klaus Koch and Hans Dreier. "Efficient calibration of automotive ECUs using a virtual measurement and calibration layer". In: *2013 IEEE International Electric Vehicle Conference*. IEEE. 2013. DOI: 10.1109/IEVC.2013.6681180.
[36] Steve Krug. *Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability*. 3rd. New Riders, 2014. ISBN: 978-0321965516.
[37] Paul Leach, Michael Mealling, and Rich Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122, Internet Engineering Task Force (IETF). 2005. URL: https://www.rfc-editor.org/info/rfc4122 (visited on 06/26/2025).
[38] Open Robotics. *Robot Operating System 2 (ROS2)*. Online. Available: https://docs.ros.org/en/humble/index.html. 2023. (Visited on 04/06/2025).
[39] ReactiveX. *RxJS: Reactive Extensions Library for JavaScript*. Online. Available: https://rxjs.dev/. 2024. (Visited on 06/24/2025).
[40] Open Robotics. *rosbag2: Data Logging in ROS2*. Online. Available: https://github.com/ros2/rosbag2. 2023. (Visited on 04/06/2025).
[41] H. Schneider and B. Muellender. "Standardized calibration protocols in AUTOSAR ECUs". In: *SAE Technical Paper Series*. SAE International. 2010. DOI: 10.4271/2010-01-0591.
[42] Gerald Steinbauer. "A survey about middleware for robotic systems". In: *Autonomous Systems: Developments and Trends* (2010). DOI: 10.1007/978-3-642-11688-9_32.
[43] Mikael Stenmark and Jacek Malec. "Developing modular graphical user interfaces for ROS using rqt". In: *Procedia Computer Science* 76 (2015). DOI: 10.1016/j.procs.2015.12.286.
[44] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. ISBN: 978-0132392273. 2007.
[45] Feng Tao, Yu Li, and Xin Wang. "Data replay and visualization for simulation-based testing in autonomous vehicle development". In: *Simulation Modelling Practice and Theory* 56 (2015). DOI: 10.1016/j.simpat.2015.03.006.
[46] Robot Web Tools. *roslibjs: The Standard ROS JavaScript Library*. Online. Available: https://github.com/RobotWebTools/roslibjs. 2023. (Visited on 05/21/2025).
[47] Zhenyu Wang, Donghoon Lee, and Jae-Bok Song. "A real-time data logging and visualization system for mobile robotics using ROS". In: *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*. IEEE. 2017. DOI: 10.1109/ROBIO.2017.8324511.
[48] Claus O Wilke. *Fundamentals of Data Visualization: A Primer on Making Informative and Compelling Figures*. O'Reilly Media, 2019. ISBN: 9781492031086.
[49] Claus O. Wilke. *Handbook of Data Visualization*. CRC Press, 2021. ISBN: 9780367331568.

Acknowledgment

First and foremost, I would like to express my deepest gratitude to my supervisors, Dr. Sebastian Heil and Jan Ingo Haas, for their continuous guidance, invaluable feedback, and unwavering support throughout the development of this thesis.

I extend my sincere thanks to the team at ETAS GmbH, a Robert Bosch Company, for providing technical resources and mentoring during my thesis. Special thanks go to Prashantha Nettur Ramachandra Rao for their insightful suggestions on INCA integration and data measurement frameworks.

I am also grateful to my friends and colleagues at ETAS GmbH for their collaboration and encouragement, especially during the intense debugging and visualization phases of this project.

Finally, I wish to thank my family for their patience, motivation, and constant belief in my abilities, without which this journey would not have been possible.

Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this Master thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This thesis has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.

.....*D..M..Dhameliya*.....
Dharam Mukeshbhai Dhameliya

Selbstständigkeitserklärung

Ich erkläre gegenüber der Technischen Universität Chemnitz,dass ich die vorliegende Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

.....*D..M..Dhameliya*.....
Dharam Mukeshbhai Dhameliya