FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Container-Based Framework for Validation of Agricultural Robotic Software Components

Ana Luísa Campos e Sampaio Melo



Mestrado em Engenharia Eletrotécnica e de Computadores

Supervisor: Dr. Luís Carlos Feliz Santos

Second Supervisor: Dr. Rafael Lírio Arrais

October 27, 2025

© Ana Luísa Campos e Sampaio Melo, 2025

Resumo

O desenvolvimento de sistemas autónomos na agricultura tornou-se essencial para responder a desafios criticos, como a escassez de mão de obra, a sustentabilidade ambiental e a segurança das operações. Os robôs agrícolas, concebidos para executar tarefas fundamentais como a plantação ou a colheita, exigem software robusto e fiável para operar em abientes dinâmicos e imprevisíveis. Garantir que estes sistemas se comportam corretamente durante a execução é crucial, sobretudo quando falhas podem comprometer a produtividade ou a segurança.

Esta dissertação propõe um *framework* baseada em *containers* para a validação de aplicações ROS2 na robótica agrícola, que utiliza como base de execução a TRIBE Navigation Stack, um sistema ROS2 desenvolvido no INESC TEC para navegação autónoma em vinhas de declive acentuado e outros ambientes rurais. Ao invés de se focar na avaliação dos algorítmos específicos de navegação, o *framework* dedica-se à verificação, durante a execução de missões autónomas, de requisitos funcionais e temporais previamente definidos.

Com o propósito de alcançar este objetivo, a solução desenvolvida integra o Rigel, uma plataforma destinada ao desenvolvimento e execução de aplicações ROS em ambientes baseados em *containers*, através de um *plugin* dedicado à monitorização de requisitos em tempo de execução. Este *plugin* processa um ficheiro de especificação associado à missão e permite a avaliação, em tempo real, do comportamento do robô face aos requisitos especificados, tais como a travessia correta por pontos-chave ou o cumprimento de restrições espaciais. A validação decorre em ambientes de simulação, assegurando a repetibilidade dos testes e permitindo uma avalição sistemática sob diferentes condições.

Os resultados obtidos evidenciam a viabilidade e expressividade da abordagem, com potencial de integração em fluxos de execução automatizados de testes contínuos. Ao articular a verificação em tempo de execução com o uso de *containers*, o *framework* proposto promove práticas de validação reprodutíveis e automatizadas em sistemas robóticos desenvolvidos em ROS2 aplicados à agricultura.

Palavras-chave: avaliação em tempo de execução; ROS2; containerização; validação de aplicações robóticas; *framework* baseada em plugins; robótica agrícola.

Abstract

The development of autonomous systems in agriculture has become pivotal in addressing critical challenges, including labour shortages, sustainable practices and operational safety. Agricultural robots, designed to perform essential tasks such as planting or harvesting, require robust and reliable software to operate in dynamic and unpredictable environments. Ensuring that such systems behave correctly during execution is essential, particularly when failures may compromise crop yields or safety.

This dissertation proposes a container-based framework for validating ROS2 applications in agricultural robotics, which is built on top of the TRIBE Navigation Stack, a ROS2 system developed at INESC TEC for autonomous navigation in steep slope vineyards and other rural environments. Rather than evaluating individual navigation algorithms, the framework focuses on verifying whether robotic systems behave according to predefined functional and temporal requirements during mission execution.

To achieve this objective, the proposed solution extends Rigel, a platform for containerised development and deployment of ROS systems, with a dedicated plugin for runtime requirement monitoring. This plugin processes a mission-specific specification file and enables real-time assessment of the robot's behaviour against the specified requirements, including correct waypoint traversal and adherence to spatial constraints. Validation is performed within simulation-based workflows, ensuring repeatability and enabling systematic testing across diverse operational conditions.

The results demonstrate the feasibility and expressiveness of the validation approach, with support for integration into continuous testing pipelines. By bridging runtime verification with containerisation, the proposed framework enables reproducible and automated runtime validation of ROS2-based robotic systems deployed in agricultural contexts.

Keywords: runtime assessment; ROS2; containerisation; validation of robotic applications; plugin-based framework; agricultural robotics.

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) present a comprehensive blueprint for the promotion of global prosperity and peace, whilst ensuring environmental sustainability and social equity. The 17 goals address some of the world's most urgent challenges, including food security, climate change, innovation, and responsible resource usage.

This dissertation contributes to several SDGs through the development of a runtime validation framework for ROS2-based robotic systems. This work supports more sustainable and efficient agricultural practices.

The contribution of this work can be aligned with the following Sustainable Development Goals:

SDG 2: End hunger, achieve food security and improved nutrition and promote sustainable agriculture

Target 2.4: By 2030, ensure sustainable food production systems and implement resilient agricultural practices that increase productivity and production, that help maintain ecosystems, that strengthen capacity for adaptation to climate change, extreme weather, drought, flooding and other disasters and that progressively improve land and soil quality.

SDG 9: Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

Target 9.5: Enhance scientific research, upgrade the technological capabilities of industrial sectors in all countries, in particular developing countries, including, by 2030, encouraging innovation and substantially increasing the number of research and development workers per 1 million people and public and private research and development spending.

SGD 12: Ensure sustainable consumption and production patterns

Target 12.2: By 2030, achieve the sustainable management and efficient use of natural resources.

SGD 13: Take urgent action to combat climate change and its impacts

Target 13.3: Improve education, awareness-raising and human and institutional capacity on climate change mitigation, adaptation, impact reduction and early warning.

SGD	Target	Contribution	Performance Indicators and Metrics
2	2.4	The validation framework ensures autonomous robots operate reliably in dynamic agricultural environments, contributing to efficient and sustainable food production systems.	Success rate of validated missions, tool usage correctness, real-time detection of execution faults.
9	9.5	Development of containerised, modular assessment tools enhances scientific research capacity and promotes scalable deployment of robotics infrastructure.	Number of automated assessments, reusability of containerised frameworks, integration with CI/CD pipelines.
12	12.2	Runtime assessment reduces unintended robotic behaviour, helping avoid inefficient use such as energy, water or fertilizer during agricultural operations.	Tool activation only in valid zones, adherence to spatial constraints
13	13.3	Simulation-based validation reduces the need for repeated physical testing, lowering emissions and environmental impact during robotic system development.	Ratio of simulated to real-world tests, decrease in physical test emissions, efficiency of test iteration cycles

Agradecimentos

Agradeço ao meu orientador, Luís Santos, pela orientação contínua, disponibilidade e pela motivação ao longo de todo o trabalho, bem como pela oportunidade de o desenvolver.. Ao coorientador, Rafael Arrais, agradeço o acompanhamento técnico e os comentários que contribuíram para a consolidação da dissertação. Agradeço também ao Pedro Melo, pelo apoio prático e técnico durante todo o desenvolvimento.

À FEUP e à UNIGE por todo o conhecimento, exigência e oportunidades que me proporcionaram, e por contribuírem não só para o meu desenvolvimento técnico, mas também pessoal.

A dois grupos que me acompanharam ao longo destes dois anos de mestrado, e que marcaram este percurso de formas muito diferentes, mas igualmente importantes. Ao NEEEC, por ter sido mais do que um grupo académico, foi um espaço de crescimento, amizade e muitos desafios. Ao GASPorto, pela experiência de voluntariado, partilha e motivação - "Estamos Juntos".

À Bárbara, pela amizade e por me fazer companhia durante tantos dias de trabalho, tanto dentro como fora do laboratório.

Aos meus amigos, pelo apoio, pela distração nos momentos certos e por me lembrarem de aproveitar o percurso. Em especial à Catarina, que me ouviu demasiadas vezes dizer “ainda agora funcionava” e respondia sempre com um “eu não percebo, mas sei que tens razão”.

Por fim, à minha família, pelo apoio incondicional ao longo de todos estes cinco anos de FEUP, pelas palavras de incentivo, pela (muita) paciência e pelas 1001 boleias até à faculdade.

Ana Luísa Melo

"When something is broken, you can put it back together. It may not fit quite the same, or work like it did once before, but that doesn't mean it's no longer useful."

T.J. Klune, The House in the Cerulean Sea

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Motivation . . . 2
- 1.3 Research Objectives and Questions . . . 3
- 1.4 Dissertation Structure . . . 3

2 State of the Art **5**
- 2.1 Background . . . 5
  - 2.1.1 Agricultural Automation . . . 5
  - 2.1.2 Autonomous Navigation . . . 7
- 2.2 ROS and ROS2 . . . 8
  - 2.2.1 ROS2 for Agricultural Robotics . . . 9
- 2.3 Assessment of Robotic Software Components . . . 10
- 2.4 Containerisation Technologies in Robotics . . . 12
  - 2.4.1 Fundamentals . . . 12
  - 2.4.2 Applications in Robotics . . . 14
- 2.5 Rigel Framework . . . 15
- 2.6 TRIBE Navigation Stack . . . 16
- 2.7 Summary . . . 17

3 Runtime Assessment Mechanism **19**
- 3.1 Overview of the Runtime Assessment Strategy . . . 19
- 3.2 ROS1 to ROS2 Adaptation of the Runtime Assessment Tool . . . 20
- 3.3 System Architecture . . . 21
- 3.4 Specification Format and Assessment Logic . . . 22
  - 3.4.1 Example Specification File . . . 23
- 3.5 Runtime Assessment Use Case: Turtlesim . . . 25
- 3.6 Summary . . . 30

4 Rigel Plugin for Runtime Assessment **31**
- 4.1 Overview . . . 31
- 4.2 Runtime Assessment Plugin . . . 32
- 4.3 Mission Deployment Plugin . . . 33
- 4.4 Rigelfile Configuration and Execution Flow . . . 34
- 4.5 Execution of the Assessment Plugin . . . 38
- 4.6 Summary . . . 39

5 Runtime Assessment in Agricultural Mission **41**
5.1 Context and Environment . . . 41
5.2 Overview of the Assessment Scenarios . . . 42
5.3 Scenario 1: Basic Navigation Assessment . . . 43
5.3.1 Mission Description . . . 43
5.3.2 Assessment Configuration . . . 44
5.3.3 Results . . . 46
5.4 Scenario 2: Complex Navigation Assessment . . . 47
5.4.1 Mission Description . . . 47
5.4.2 Assessment Configuration . . . 50
5.4.3 Results . . . 54
5.5 Runtime Assessment Plugin Execution . . . 56
5.6 Summary . . . 57

6 Conclusions **59**
6.1 Outcomes . . . 59
6.2 Research Objectives and Questions Revisited . . . 59
6.3 Limitations and Future Work . . . 60

References **61**

A Full YAML Specification **65**
A.1 Turtlesim Scenario . . . 65
A.2 Simple Agricultural Scenario . . . 69

List of Figures

2.1 Comparison between ROS1 and ROS2 architectures 9

3.1 Conceptual architecture of the runtime assessment system, showing the interaction between core modules. 22
3.2 Example of a square trajectory followed by the turtle 27
3.3 Validation log output indicating successful compliance with all requirements for the pose topic 28
3.4 Example of a rectangle trajectory followed by the turtle 28
3.5 Validation log output for the pose topic after rectangle trajectory 29

4.1 Example of terminal output illustrating the lifecycle execution of the Runtime Assessment Plugin (`setup`, `start`, `process`, and `stop`). 33

5.1 Gazebo simulation environment used in the mission. 42
5.2 Visual confirmation of tool state during the mission. 43
5.3 Planned navigation trajectory used in the first assessment scenario 44
5.4 Runtime assessment log results for Scenario 1. 46
5.5 Trajectory of the second scenario 48
5.6 Segmented view of the trajectory of Scenario 2 49
5.7 Runtime assessment log results for Scenario 2. Requirements are colour-coded according to the corresponding mission segment: pink (initial point), blue (blue segment), green (green segment), orange (orange segment), red (unreachable segment), and yellow (global requirements). 55
5.8 Terminal output from a plugin execution 56

List of Tables

3.1 Example of changes during migration from ROS1 to ROS2 20
3.2 Summary of file-level contributions in the ROS1 to ROS2 migration 23
3.3 `Setup` configuration parameters 24
3.4 ROS topics monitored during the Turtlesim Assessment 26
3.5 Summary of validation results for the `/turtle1/pose` topic 29
3.6 Summary of validation results for the `/turtle1/pose` topic 29

4.1 Parameter defined in the `Rigerlfile` 36
4.2 Execution order of plugin lifecycle functions for the `assess` sequence 38

5.1 Requirements for the first Agricultural Scenario 45
5.2 Requirements for the second Agricultural Scenario 53

Listings

3.1 Simplified YAML specification 23
4.1 Complete `Rigelfile` used for launching the simulation and executing the runtime assessment. 35
4.2 Concurrent sequence used 36
A.1 Complete specifications.yaml file used for Turtlesim example 65
A.2 Complete specifications.yaml file used for the first agricultural scenario 69

List of Acronyms

AI	Artificial Intelligence
AgRobPP	Agricultural Robotic Path Planning
API	Application Programming Interface
CD	Continuous Delivery
CI	Continuous Integration
CPU	Central Processing Unit
DDS	Data Distribution Service
DEM	Digital Elevation Model
FRET	Formal Requirement Elicitation Tool
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HAROS	High-Assurance ROS
HPL	HAROS Property Specification Language
INESC TEC	Instituto de Engenharia de Sistemas e Computadores, Tecnologia e Ciência
LiDAR	Light Detection and Ranging
MTL	Metric Temporal Logic
PC2GD	Point Cloud to Grid Map and DEM
QA	Quality Assurance
QoS	Quality of Service
RFID	Radio-Frequency Identification
ROS	Robot Operating System
ROS2	Robot Operating System 2
RQ	Research Questions
SLAM	Simultaneous Localization and Mapping
VM	Virtual Machine
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

The present chapter provides a comprehensive introduction to the context, motivation, and objectives of this dissertation. It starts by outlining the current challenges and emerging trends in agricultural robotics, with a particular focus on the increasing need for reliable and adaptable software in dynamic field environments. As part of addressing these challenges, this work focuses on two core technological foundations, the runtime assessment of robotic systems, which allows for the continuous validation of system behaviour during execution, and containerisation, which enables reproducible environments. The motivation for this work is then presented, highlighting the gaps in existing validation methodologies for robotic systems operating in agriculture. The chapter concludes with a presentation of the dissertation's primary objectives and an overview of its structure.

1.1 Context

Agriculture remains a cornerstone of global economic and social stability. It is essential not only for food security, but also for sustaining rural livelihoods and contributing to sustainable development. As the global population grows and environmental pressures intensify, increasing the efficiency, resilience, and sustainability of agricultural systems has become a strategic priority.

In recent years, the agricultural sector has undergone a significant transformation, driven by the imperative need to increase productivity, overcome labour shortages, and improve sustainability. This need has led to the growing integration of automation and digital technologies across agricultural work. Robotics and Artificial Intelligence (AI), particularly, have become key enablers of this transformation, providing economically viable solutions for both standardised and non-standardised tasks, such as fruit picking, selective weeding, and crop monitoring [1].

Unlike industrial settings, agricultural environments are inherently dynamic and unstructured. They are characterised by fluctuating weather, uneven terrains, and biological variability - all of which impose significant challenges on the development of autonomous systems. Robotic operations must function safely and effectively in a variety of conditions, often requiring real-time adaptability and complex sensory integration. As highlighted in [1], agricultural robotics must go

beyond simple automation, embodying cognitive capabilities to handle variable crop characteristics, spatial uncertainties, and time-sensitive operations.

In this context, the reliability of robotic software becomes a critical factor. Ensuring that software behaves correctly and predictably in the face of uncertainty requires systematic validation mechanisms that are not only robust, but also modular and scalable. However, traditional validation methods often lack the adaptability required to address the complexities of field robotics in agriculture. In response to these challenges, runtime assessment has become a promising approach, enabling the continuous verification of system behaviour during mission execution.

Among the technological responses proposed in recent years, containerisation has emerged a promising solution. It enables the creation of isolated, reproducible, and portable environments that facilitate the testing and deployment of complex robotic software systems. In conjunction with the Robot Operating System 2 (ROS2) - a software development kit for robotics applications designed for modularity, distributed computing, and real-time performance - containerisation offers a strong framework for developing and validating robotic applications in agriculture.

This dissertation explores the integration of container-based validation strategies within ROS2-based agricultural robotic systems, with the aim of enhancing software reliability under realistic operating conditions.

1.2 Motivation

The increasing complexity of agricultural environments, coupled with the growing demand for automation, has made the reliability of robotic software a critical concern. While frameworks such as ROS2 and technologies like containerisation have shown promise in robotics, there remains a notable lack of standardised tools to validate robotic systems behaviour under real-world scenarios.

In particular, agricultural robotics faces unique challenges that are not adequately addressed by existing validation methods. Field conditions are highly dynamic, and the interactions between robotic components often have to be adapted in real time based on environmental triggers. Such scenarios require not only static testing during development but also runtime verification to ensure safety, efficiency, and functional correctness throughout operation.

Moreover, existing approaches to validation tend to rely on tightly coupled, scenario-specific configurations, which makes them difficult to scale or generalise across different robotic applications. This fragmentation hinders both reproducibility and wider adoption in agricultural contexts.

Despite the increasing use of ROS2 and containerised deployments, there is a lack of runtime validation tools that are both generic and easily integrable into agricultural navigation systems [2]. To address this gap, this dissertation aims to develop a container-based framework that integrates directly with ROS2 applications. The proposed work builds upon a previously developed runtime assessment tool implemented in ROS1, extending its architecture and adapting its components to the ROS2 ecosystem. The The motivation is to create a solution that supports real-time verification of functional and temporal requirements, while maintaining modularity, scalability, and ease of deployment. Ultimately, the goal is to contribute a practical and reusable validation tool that

improves the dependability of agricultural robotic systems, enabling safer and more consistent deployments across diverse and unpredictable environments.

1.3 Research Objectives and Questions

The primary objective of this dissertation is to contribute to the field of agricultural robotics by developing a container-based framework for validating ROS2 applications. This is achieved by creating a ROS2 plugin that facilitates the runtime validation process of robotic software components.

To support this goal, the work is guided by the following research questions:

- **RQ1 - How can runtime assessment of ROS2 applications be made effective and scalable in agricultural scenarios?**
- **RQ2 - Can containerisation improve the reproducibility and isolation of runtime validation workflows?**

To address these questions, the following objectives were defined:

- Research and develop a runtime mechanism capable of monitoring compliance with spatial and temporal requirements during execution;
- Integrate the assessment plugin within Rigel, a container-based framework, enabling automated orchestration and assessment workflows;
- Validate the proposed approach using realistic agricultural navigation scenarios, demonstrating its applicability and practical benefits.

These research questions and objectives structure the scope of the dissertation and are revisited in the conclusions to reflect on the outcomes achieved.

1.4 Dissertation Structure

This document is divided into six chapters:

- Chapter 2 provides a comprehensive review of the relevant literature and technologies, including ROS2, containerisation, runtime validation tools, and navigation systems in agriculture.
- Chapter 3 details the architecture, specification logic, and implementation of the runtime assessment system, including its migration from ROS1 to ROS2.
- Chapter 4 details the integration of the assessment mechanism into the Rigel framework via custom plugin, enabling automated deployment and assessment in containerised environments.

- Chapter 5 illustrates the implementation of the system in simulated agricultural scenarios utilising the TRIBE Navigation Stack, highlighting results and performance analysis.
- Chapter 6 summarises the key outcomes, revisits the research questions, and discusses limitations and potential directions for future work.

Chapter 2

State of the Art

This chapter provides an overview of the state of the art in agricultural robotics, with a focus on challenges encountered in this field. Particular emphasis is placed on the validation of robotic software components, a critical aspect of ensuring reliability and efficiency in dynamic agricultural environments.

The chapter also explores how containerisation technologies are transforming the development, testing and deployment of robotic applications, providing scalable and reproducible solutions to the complex challenges of integration and run-time validation.

2.1 Background

2.1.1 Agricultural Automation

Agricultural automation has emerged as a key enabler of modern farming, driven by the growing demands to increase productivity, improve sustainability, and reduce dependence on human labour. With a growing population and rising pressures on food systems, the agricultural sector faces significant challenges in maintaining efficiency while minimising environmental impact.

In response, autonomous technologies have evolved to address a wide range of tasks that were traditionally dependent on manual labour, including planting, harvesting, crop health monitoring and material handling [3]. Such technologies not only reduce the burden on the human workforce, a pressing concern in modern agriculture, but also allow for continuous operation, including during periods when labour availability is limited [3] [1].

By automating these repetitive and labour-intensive tasks, robotic systems improve operational precision and promote greater consistency in agricultural workflows, leading to overall productivity of the operations. They also support the implementation of precision agriculture techniques, enabling more targeted interventions such as site-specific fertilisation or selective weeding. Furthermore, these systems have the potential to revolutionise agriculture, enhancing the sustainability of farming practices by promoting more efficient use of inputs like water, fertilisers, and energy, ultimately reducing the sector's environmental footprint [4].

Despite these benefits, the lack of robust software validation mechanisms in agricultural robotics can lead to significant consequences. Faulty or unvalidated software can pose operational risks, resulting in inefficiencies, errors in agricultural processes, and consequent resource wastage [4]. Variations in weather, terrain, and crop structure require robots to adapt in real time, a challenge that is difficult to meet without continuous runtime validation [5]. For example, sudden changes in soil consistency or unexpected obstacles, such as a branch fallen on the ground, can disrupt operations if the software lacks mechanisms to adapt dynamically.

These challenges highlight the importance of developing validation frameworks to assess software behaviour under realistic and unpredictable conditions. In particular, while automation presents significant opportunities for the agricultural sector, it also introduces substantial technical challenges that must be carefully addressed. Unlike industrial settings, which are typically structured and predictable, agricultural environments are highly dynamic, unstructured and constantly changing. This variability makes it particularly difficult to guarantee consistent performance from robotic systems in real-world farming conditions.

To ensure that agricultural robotics are scalable, adaptable, and useful to farmers, these systems must be designed to operate reliably under a broad spectrum of environmental and operational conditions. The following are some of the most pressing challenges faced in agricultural robotics [5], [3], [6]:

- **Environmental Challenges:** Agricultural fields often present uneven terrain, natural obstacles and varying soil conditions. For example, in environments such as steeply sloping vineyards, the risk of instability becomes even more pronounced, requiring the development of navigation systems that take into account the robot's centre of gravity and the terrain's slope.

- **Limitations of Sensor Technology:** Sensors play a crucial role in robotic navigation and decision-making. However, technologies like Global Navigation Satellite Systems (GNSS) are prone to signal interference, especially in areas with dense vegetation or steep terrain. Alternative approaches, such as Simultaneous Localisation and Mapping (SLAM) and PointCloud-based navigation, can help overcome these limitations, but often require significant computational resources and remain sensitive to environmental noise.

- **Integration with existing systems:** Many farms rely on legacy equipment and systems that were not designed to communicate with modern digital or robotic systems. Integrating robotic solutions into these settings poses significant compatibility challenges. To achieve seamless operation, new robotic solutions must be economically viable and interoperable with existing agricultural machinery, data formats and software ecosystems, a requirement that adds complexity to system design and deployment.

- **Adaptability to Task-Specific Requirements:** Agricultural activities such as planting, weeding and harvesting vary greatly in terms of technical and environmental requirements. A robot that exhibits high performance in one setting may exhibit substandard performance

in another. Therefore, robots must be designed to perform these tasks efficiently while adapting to specific field conditions, such as soil type, crop structure, and weather.

The challenges outlined above emphasise the complexity of deploying reliable and scalable solutions in agriculture. Overcoming them requires robust software systems that can be validated under dynamic and unpredictable environments.

2.1.2 Autonomous Navigation

Autonomous navigation is a fundamental aspect of agricultural robotics. It allows machines to operate independently across expansive and often complex fields, without the need for human control. These systems rely on a combination of advanced sensors, planning algorithms, and computational techniques to navigate environments, avoid obstacles, and perform tasks with high precision. The unique characteristics of agricultural terrains, such as irregular topography, dense vegetation, and presence of dynamic obstacles, place specific demands on navigation systems.

Agricultural fields present a wide range of terrain challenges, from flat and uniform surfaces to steep and irregular slopes. In hilly environments, like vineyards on steep terrain, ensuring the stability of robots is critical, particularly when navigating slopes where factors such as centre of gravity and ground conditions must be taken into account to avoid accidents or environmental damages. Navigation algorithms such as AgRobPP, presented in [6], which extend traditional A* planning methods, have been developed to address these challenges by incorporating terrain-specific parameters to generate safe and efficient paths. Other tools, such as Digital Elevation Models (DEMs) and segmented 3D maps, have also been used to represent and analyse agricultural terrain for navigation.

Effective obstacle detection and avoidance are also essential for autonomous navigation. In agricultural environments, obstacles can appear suddenly, such as moving equipment, debris or workers. To detect and avoid such hazards, agricultural robots use advanced sensors like LiDAR (Light Detection and Ranging) and cameras. These devices scan the surroundings and spatial awareness and decision-making capabilities are enhanced, allowing robots to navigate safely and efficiently [5]. These capabilities are critical for agricultural tasks such as spraying, weeding, and harvesting, where precision and adaptability are important to not damage crops or reduce efficiency.

Given these complexities, validating the software components responsible for autonomous navigation becomes essential to ensure reliable operation in diverse agricultural conditions. This means testing whether the robot can correctly interpret sensor data, follows the path planned, and execute movements without failure. It is vital to recognise that field conditions vary significantly, thus necessitating the employment of validation frameworks that are capable of simulating real-world environments and evaluating the software's performance under diverse scenarios. This ensures the reliable performance of the navigation system, thereby enhancing the safety and robustness of agricultural robotic solutions.

2.2 ROS and ROS2

The Robot Operating System (ROS) is a widely adopted open-source middleware framework designed to support the development of complex robotic applications through a modular and distributed architecture. It provides essential tools and libraries for building robotic applications, including communication middleware, device drivers, simulation environments, package management, and debugging utilities.

In ROS, applications are structured as a network of *nodes*. Each node is responsible for a specific task and communicates with other nodes through structured channels: *topics*, for streaming data; *services*, for request-response calls; and *actions*, for long-duration tasks. This modularity enables teams to develop, test, and reuse components independently, which is especially important in large-scale or collaborative robotic projects.

Despite its success, ROS1, the first version of ROS, presents several architectural and functional limitations that affect its scalability, reliability, and suitability for production environments. These limitations include the lack for real-time communication, limited support for multi-robot systems, poor cross-platform compatibility, and weak security features. Moreover, it relies on a centralised master node, which introduces a single point of failure and restricts distributed operation. These constraints hinder the deployment of ROS1 in large-scale and mission-critical applications, such as autonomous agricultural robots.

To overcome these challenges, ROS2 was developed. It has a new architecture based on the Data Distribution Service (DDS) standard, a robust and real-time-capable communication protocol used in critical systems. DDS enables peer-to-peer communication, meaning that nodes can exchange information directly without relying on a central master [7]. This design brings improved flexibility, reliability, and scalability.

Figure 2.1 illustrates the architectural differences between ROS1 and ROS2, particularly regarding communication decentralisation and middleware layers.

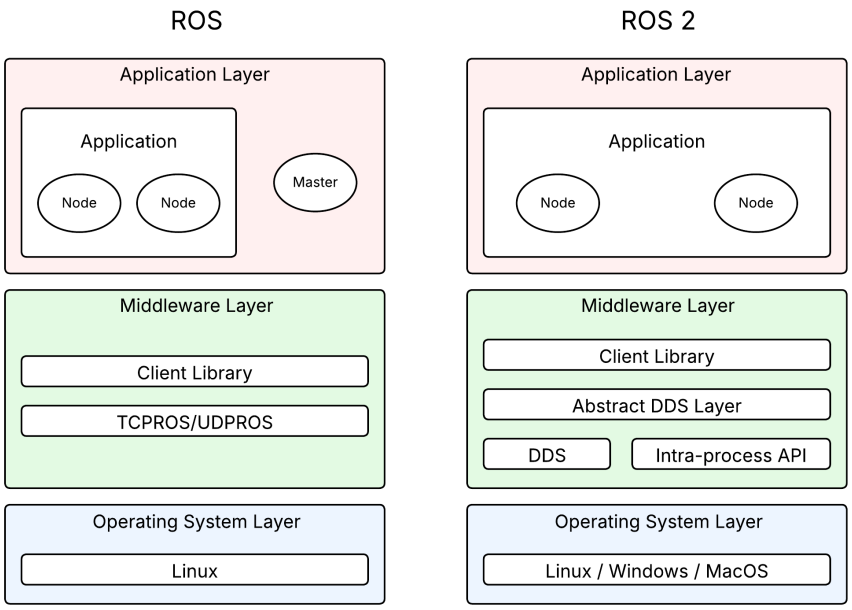


Figure 2.1: Comparison between ROS1 and ROS2 architectures

The main enhancements introduced in ROS2 include:

- **Real-Time Communication:** DDS enables deterministic messaging with fine-grained Quality of Service (QoS) settings.
- **Multi-Robot and Distributed Support:** ROS2 supports decentralised computing across networks, eliminating the need for a central master.
- **Security and Lifecycle Management:** Provides tools for managing node states, secure communication, and system introspection.
- **Cross-Platform Compatibility:** ROS2 supports Linux, Windows, and macOS, enabling broader deployment options.
- **Enhanced Modularity:** Facilitates isolated development, testing, and reuse of individual components.

ROS2 has been widely adopted as the successor to ROS1 and is increasingly used in both research and industry to build robust, real-time, and distributed robotic systems [8].

2.2.1 ROS2 for Agricultural Robotics

The architectural features of ROS2 make it particularly well-suited for agricultural robotics. In the context of large, unstructured, and dynamically changing environments, it is imperative that robots are capable of operating autonomously and reacting to all environmental stimuli.

ROS2's capabilities, like real-time data processing, modularity, and distributed communication, are critical for tasks such as navigation, obstacle avoidance, and execution of context-aware behaviours. These properties help ensure that robotic systems can adapt their behaviour on the fly, which is essential for field operations that involve uncertainty and environmental variability [5][6].

Furthermore, ROS2 improves upon limitations of the previous version by offering enhanced support for node lifecycle management, improved communication middleware (DDS), and better integration with modern development practices. These improvements facilitate the design of flexible robotic systems that can be tailored to diverse mission profiles, which is a crucial requirement for agricultural settings where variability is the norm.

2.3 Assessment of Robotic Software Components

The assessment of software components in robotics is essential to ensure the safety, reliability, and functional correctness of systems operating in dynamic environments, particularly in agriculture. The capacity of agricultural robots to function in the scenarios previously mentioned, like variable terrain, changing weather conditions and dynamic obstacles, is a critical consideration. As such, robust validation frameworks are essential to ensure that robotic software performs reliably under real-world conditions [9][10].

While software testing in traditional domains often benefits from static environments, robotics introduces unique challenges due to the interaction between hardware, perception systems, and real-time decision-making. In agricultural contexts, these difficulties are exacerbated by the complexity of sensor fusion, real-time constraints, and the need for dynamic task adaptation. The lack of standardized and repeatable validation practices in robotics has also been highlighted in work advocating for reproducible and measurable robotics research [10].

The validation of software can be approached in two main ways: static analysis and runtime assessment. Static analysis techniques enable developers to inspect source code without execution, thereby detecting potential defects and enforcing coding standards at an early stage of development. These approaches are widely adopted because they are simple to automate, applicable from early development stages, and accessible to developers without the need for expertise in formal methods [11]. In the context of ROS-based development, tools such as HAROS (High-Assurance ROS) have been proposed to support static analysis of robotic applications by checking compliance with coding standards, analysing ROS communication patterns such as publishers, subscribers, and service calls, inspecting launch file configurations, and extracting architectural models directly from source code [12]. However, static techniques have been shown to have limitations in their ability to detect runtime-specific faults, particularly those related to real-time behaviour, concurrency, or interaction with complex environments [12].

Runtime assessment complements static analysis by enabling observation of the system during execution. This in turn allows the detection of issues which may be considered to be subtle, such as temporal violations, non-deterministic failures, or incorrect coordination between components.

As highlighted by Caldas et al. [2], these runtime techniques are particularly useful for dealing with faults that are difficult or impossible to identify using static methods alone, especially in uncertain and unpredictable operational contexts.

One of the most significant difficulties in current methodologies lies in runtime validation of functional requirements, that is, the ability to verify that robotic systems continue to behave correctly while operating in the field. In ROS2, this challenge is further amplified by the complexity of its middleware architecture and distributed communication, especially those based on DDS and configurable QoS policies, which introduce additional layers of uncertainty and tuning difficulty [8]. Functional issues such as inconsistent message timing, incorrect node states, or failed sensor integration can lead to unsafe or sub-optimal behaviour, yet they are often hard to detect during development. As highlighted by Afzal et al. in [9], robotic testing frequently struggles with unpredictable environmental interactions, particularly when robots are deployed in dynamic real-world scenarios, where noise, lighting, terrain, and human interference may all affect system behaviour. For example, navigation algorithms may fail to adapt to sudden changes in terrain or loss of signal localisation, leading to incorrect path planning or localisation loss, as shown by Santos et al. [6].

Several frameworks have been proposed to address runtime verification in robotics. For example, as emphasised by Bonci et al. [8], a range of tools exists for ROS2-based testing, including simulation environments and behaviour monitoring tools. However, it also underscores the fact that the runtime validation of autonomy-related functionalities remains an open challenge.

For instance, ROSRV [13] introduces a monitoring infrastructure for ROS-based systems which intercepts messages and enforces safety and security properties during runtime without requiring any changes to the application code. These monitors are automatically generated from safety specifications and can allow, block or transform messages depending on whether they satisfy the defined properties, however, it is not compatible with ROS2.

Other efforts have focused on runtime verification techniques tailored to robotic systems. For example, Caldas et al. [2] provide 20 guidelines to support software testing and runtime verification in robotics. These include preparing ROS nodes to expose internal information that can be checked during execution, specifying system properties to be validated, and analysing execution traces to investigate failures during field-based testing. The guidelines are designed to assist developers and quality assurance (QA) teams apply runtime verification under real operating conditions.

Perez et al. [14] present an approach that enables the automatic generation of ROS2 runtime monitors from system-level requirements. This method integrates the FRET tool for capturing requirements in structured natural language and formalising them into Metric Temporal Logic (MTL), the Ogma compiler for translating MTL into stream-based monitors written in Copilot, and a ROS2 node generator that wraps the compiled monitor into a self-contained ROS2 package. The resulting monitors are fully independent of the application code and can be incorporated into ROS2 systems without modification. In addition, the monitors are compiled into hard real-time C99 code, making them particularly suitable for safety-critical systems where deterministic behaviour is essential.

A particularly relevant contribution, on which the work in this dissertation is directly based, is the Rigel assessment tool, originally developed for ROS1-based applications [15]. Rigel offers a plugin based architecture for runtime assessment of robotic systems, enabling developers to define and validate functional and temporal requirements during mission execution. Its architecture incorporates elements of static analysis frameworks, such as HAROS, but extends the concept into runtime, facilitating the evaluation of message timing, behavioural constraints, and mission progress using declarative specifications. Although the original implementation was designed for ROS1, it provides the foundation for the ROS2-compatible runtime assessment mechanism proposed in this work. A detailed description of Rigel's architecture and the assessment model is provided in Section 2.5 and Chapter 3, respectively.

To address the limitations identified in existing validation approaches, the proposed work stands out by introducing an integrated mechanism to validate ROS2 applications at runtime. This approach will evaluate component behaviour in real agriculture scenarios, with an emphasis on compliance with predefined functional requirements, generating reports on software performance. This contribution helps bridge the gap between development-time testing and field-ready runtime verification, enhancing the robustness of robotic software deployed in complex agricultural environments.

2.4 Containerisation Technologies in Robotics

2.4.1 Fundamentals

Containerisation is a technology that encapsulates an application along with all of its necessary dependencies, such as libraries and configuration files, into a isolated, portable unit, known as container. The implementation of these containers aims to ensure that applications can run consistently across different computing environments, thereby mitigating issues related to dependency conflicts and environment variability. The concept of containerisation has witnessed a notable surge in popularity in recent years, driven largely by its perceived benefits in terms of portability, reproducibility, and efficiency [10].

One of the main challenges in robotics is the difficulty of sharing and reusing software applications along with their dependencies [10]. This problem arises due to the inherent complexity of robotic systems, where software stacks often contain numerous interdependent libraries, frameworks, and configurations. In order to tackle this problem, three approaches are generally proposed:

- **Manual Documentation and Sharing:** In this approach, developers manually document the required setup information, including software dependencies, hardware configurations, experimental parameters, and operating system configurations. While this method provides a detailed blueprint of the system, it is highly labour-intensive and does not guarantee successful replication, as subtle differences in hardware and software environments often lead to inconsistent results [10].

- **Virtual Machines (VMs):** Virtual machines replicate the entire operating system and software environment, which makes them suitable for encapsulating complex systems [16]. However, VMs are considered less practical for robotics due to their relatively high memory and CPU overhead, longer boot times, and the large size of their system images; factors that can limit portability and delay deployment [17].

- **Containerisation:** This third, and increasingly preferred, approach provides an efficient, and portable alternative to manage software dependencies and configurations [18]. Unlike VMs, containers are lightweight and provide isolation without the overhead of full virtualisation, making them especially suitable for robotic contexts [17]. Containers provide isolation through the encapsulation of an application, its associated libraries, frameworks, and configuration files [15]. This isolation is pivotal in ensuring the consistent operation of software, despite variations in the underlying system, which is a feature that makes containers particularly advantageous in complex environments, such as those found in agricultural robotics.

By standardising the deployment process, containers eliminate discrepancies between development and production environments, thereby reducing the likelihood of errors and incompatibilities [15]. Portability represents a significant advantage of containerisation. Containers are characterised by their relative lightweight nature when compared to traditional virtual machines (VMs), thus allowing them to be deployed seamlessly across diverse platforms, including cloud services, local machines, and edge devices. This portability is of great importance within the field of robotics, where applications frequently necessitate transition between simulation environments and physical robots [17].

Reproducibility is a key driver for the adoption of containerisation within research and development contexts. The encapsulation of the complete software stack within a container enables researchers to ensure that experiments and results can be replicated precisely. This is especially important in robotics, where the interaction of diverse software components frequently poses significant challenges to reproducibility [15].

Several tools have been developed to support containerisation, with Docker, Podman and Kubernetes being among the most widely used. Docker[1] is the most prominent of these, providing a user-friendly interface for creating, managing, and deploying containers. Podman[2] offers similar functionality but emphasizes a daemonless architecture and increased security features. Kubernetes[3], in contrast, is a container orchestration platform that enables the deployment and management of containerised applications at scale, particularly in distributed systems.

Recent methodologies, such as those presented in [19] and [20], leverage Docker and Kubernetes to optimise the development and deployment of robotics applications across heterogeneous and hierarchical architectures. These approaches enable the integration and orchestration of multi-domain software components, ensuring scalability and efficient resource management,

[1] Docker: https://docs.docker.com/get-started/docker-overview/. Accessed on: June 7, 2025.
[2] Podman: https://github.com/containers/podman. Accessed on: June 7, 2025.
[3] Kubernetes: https://kubernetes.io/docs/concepts/overview/. Accessed on: June 7, 2025.

even in edge-cloud environments. By encapsulating ROS nodes into containers, these methodologies minimise memory and CPU overhead while maintaining functional and non-functional requirements throughout the design flow.

2.4.2 Applications in Robotics

The application of containers in robotics has significantly expanded the development and deployment of robotic systems, addressing challenges such as dependency management, replicability, and scalability. Containers allow developers to encapsulate robotic applications along with their dependencies, ensuring consistent performance across different environments, which has proved to be beneficial in areas such as research, education, and real-world deployment.

One notable application is in robotics education. Containers simplify the setup of development environments, providing students and professors with pre-configured workspaces that eliminate the need for extensive configuration. Platforms such as the one presented in [21] exemplify how containerisation and ROS-based tools can streamline the learning process by providing pre-configured templates and simulated environments, such as Gazebo, allowing students to focus on core robotics principles rather than troubleshooting software issues.

In robotics competitions and hackathons, containers also play a critical role in ensuring fairness and consistency across all teams competing. By providing a uniform environment for development and testing, containers reduce discrepancies caused by differing system configurations. For instance, competitions, such as RoboCup [22], RobotX [23] and FIRA Hackathon [24], have adopted containerisation to create standardised simulation environments. This approach allows teams to focus on developing algorithms rather than resolving compatibility issues.

Research in robotics has also benefited significantly from containerisation, particularly in the context of collaborative projects and reproducibility. Tools like SwarmRob [18] encapsulate experimental setups in containers, enabling researchers to share and replicate experiments across teams with minimal effort. This approach addresses longstanding challenges in robotics research, as noted by Bonsignorio and del Pobil [10], where diverse hardware and software configurations often hinder the replication of results.

In practical deployments, including industrial robotics, containers have been instrumental in bridging the gap between simulation and real-world robotics. By encapsulating dependencies and configurations, containers ensure that software behaves consistently across different platforms. This capability is particularly valuable in applications such as agriculture, manufacturing, and agile production chains [19], where robotic systems must transition seamlessly from testing to operation in complex environments [25], [15]. Containers enable modularity and scalability, which are essential for adapting to the dynamic demands of industrial robotics.

Another recent contribution to containerised robotic workflows is Rigel, a framework designed to streamline the development, deployment and validation of ROS-based applications using Docker [15]. Rigel builds upon previous work in static analysis, particularly the HAROS framework [12], by reusing and extending its property specification language, HAROS Property

Specification Language (HPL), for runtime verification. Its plugin-based architecture enables validation of robotic behaviour during execution, making it particularly relevant for research and field-deployed robotic systems. In the context of this dissertation, Rigel serves as the foundation for the proposed runtime validation framework, and is further described in Section 2.5.

2.5 Rigel Framework

Rigel is an open-source framework designed to support the development, validation, and deployment of ROS-based applications through containerised workflows [15]. Implemented in Python, it leverages Docker to facilitate the management of isolated development environments, ensuring reproducibility and consistency in the robotic systems' build, execution, and validation processes. Rigel is especially suited for complex robotic stacks that require the coordinated execution of multiple components, thereby enabling automated orchestration of ROS workspaces and simulation tools.

Although inspired by earlier static analysis work, Rigel repurposes and extends the HPL [12] for runtime assessment, reflecting its focus on monitoring system behaviour during execution through message exchanges over ROS topics. These specifications are then subjected to evaluation during execution, thereby enabling real-time monitoring of the system's behaviour without interfering with the operation of the robotic application.

Rigel's architecture is based on a plugin mechanism, which makes it modular and extensible. The behaviour of the framework is governed by a YAML file, the `Rigelfile`, which describes the desired workflow configuration in a structured way. This file specifies the containers to be launched, the ROS workspaces to be mounted, and the set of plugins to be activated for tasks such as environment setup, node orchestration, and runtime validation.

The key features of Rigel include:

- **Containerisation of ROS Workspaces:** Rigel supports the building and execution of ROS applications within Docker containers using standard tools, promoting portability and isolation.
- **Runtime Assessment Integration:** Through an extended version of HPL, Rigel enables runtime verification of ROS message flows, allowing the enforcement of temporal and functional requirements specified at the topic level.
- **Workflow Configuration:** The `Rigelfile` allows users to define workflows in a human-readable format, improving usability and reproducibility.
- **Plugin-Based Automation:** The plugin system enables automation of tasks such as simulation setup, environment provisioning, monitoring, and result logging, making it adaptable to mission-specific validation needs.

Rigel has been successfully applied in diverse contexts, including virtual robotics competitions such as Virtual RobotX, as well as industrial deployments in logistics and factory automation, as

mentioned in [15]. Its flexibility and modular design make it well-suited for both research and production scenarios. In particular, Rigel was designed to integrate with Continuous Integration and Delivery (CI/CD) pipelines, which are a set of automated practices for building, testing, and delivering software to ensure consistent and reliable development. Its command-line interface enables automated execution of containerisation, simulation setup, and runtime verification, allowing validation routines to be triggered whenever changes are committed to the codebase. This compatibility with CI/CD workflows supports reproducibility, reduces integration errors, and promotes continuous assurance of system behaviour throughout the development lifecycle.

The runtime assessment mechanism originally developed for ROS1 [15] serves as the foundation for the ROS2-based solution proposed in this dissertation. This new version introduces a custom plugin that integrates with ROS2 nodes in a containerised environment and enforces runtime constraints related to message timing and goal satisfaction.

2.6 TRIBE Navigation Stack

The Navigation Stack, developed at the TRIBE Laboratory [26] of INESC TEC, is a modular, ROS2-based system designed to address the challenges of autonomous navigation in sloped and irregular agricultural fields. It combines localisation, mapping, and planning modules in a framework tested both in simulation and on real platforms [27].

The localisation system integrates VineSLAM, a hybrid SLAM approach that fuses topological and metric information. The system uses natural landmarks, including vine trunks and masts, which are detected through classical image processing or deep-learning methods (e.g., Convolutional Neural Networks), and can incorporate artificial references such as Radio-Frequency Identification (RFID) tags [27]. These techniques help overcome GNSS limitations, particularly in agricultural environments with occlusion or multi-path interference. VineSLAM was initially developed to address the poor performance of GNSS and dead-reckoning sensors in steep slope vineyards, where signal degradation and terrain irregularities significantly affect localisation accuracy [6],

For terrain mapping, the stack incorporates Point Cloud to Grid Map and DEM (PC2GD), a tool responsible for converting 3D point clouds acquired from a LiDAR into terrain-segmented maps [27]. These include occupancy maps and elevation maps, which are used to support navigation in steep slope vineyards. The terrain maps are used by the planner to avoid risky trajectories and ensure safe paths.

A distinctive component of the TRIBE Navigation Stack is its terrain-aware path planning capability, designed to address the specific challenges of autonomous navigation in steep and uneven agricultural terrain. Path planning is performed by AgRobPP[4], an extension of the A* algorithm that incorporates slope and robot stability into its cost function [6]. The planner uses elevation and terrain-type layers derived from 3D point clouds to penalise areas with high inclination or low terrain support. It also segments the environment into safe and non-traversable regions, guiding

[4] AgRobPP repository: https://gitlab.inesctec.pt/agrob/agrob_pp. Accessed on: June 17, 2025.

the robot along stable routes that respect its centre of mass. The approach has been tested on real robots in steep-slope vineyards, where it successfully avoids risky trajectories and generates smooth and stable paths.

Despite the various capabilities offered by the TRIBE Navigation Stack, such as terrain-aware planning, localisation, and mapping, the most significant aspect in the context of this dissertation is its ability to execute autonomous missions in a realistic field environment. The stack provides a ROS2-based execution environment that supports the deployment and management of autonomous agricultural missions, thereby enabling the proposed validation framework to monitor system-level behaviour during runtime. The system's modular design and containerised deployment model make it particularly well suited for integration with Rigel, allowing for automated, reproducible, and scalable assessment workflows.

2.7 Summary

The state of the art in agricultural robotics highlights considerable progress in automation, navigation and system integration. Nevertheless, there are still significant gaps in the assurance of the reliability and adaptability of robotic software under field conditions. Containerisation technologies offer a promising solution for reproducibility and deployment, yet they do not, in and of themselves, address the complexities of runtime validation.

Existing tools and frameworks, such as simulation-based testing and containerised deployment pipelines, play a pivotal role in supporting development and integration testing, and, in some cases, deployment in final products. However, most lack integrated mechanisms for validating system behaviour at runtime. This is of particular importance in the monitoring of compliance with functional and temporal requirements during operation. This limitation is especially relevant in the domain of agricultural robotics, where variations in terrain, environmental noise, and task-specific constraints demand adaptable systems, even in high-fidelity simulation environments.

This dissertation aims to address that gap by proposing a container-based validation framework for ROS2 applications, focused on runtime verification of functional and temporal requirements. The use of the TRIBE Navigation Stack as a complex use case strengthens the practical relevance of the approach. The proposed work contributes to the field of agricultural robotics through the integration of containerisation, ROS2 modularity, and dynamic validation logic. This combination enables the creation of a scalable and reusable solution that enhances the robustness of agricultural robotic software.

Chapter 3

Runtime Assessment Mechanism

This chapter presents the runtime assessment mechanism developed to evaluate the execution of ROS2 nodes against predefined behavioural specifications. Sections 3.1 and 3.2 outline the overall assessment strategy and describe how the original ROS1 mechanism [28] was adapted to operate under ROS2. Section 3.3 describes the modular architecture of the assessment node and its internal components. Section 3.4 explains the structure and semantics of the YAML specification file used to encode the requirements. Finally, Section 3.5 presents a practical use case using the Turtlesim simulator to demonstrate both successful and failed assessments.

3.1 Overview of the Runtime Assessment Strategy

Robotic systems deployed in agricultural environments must not only perform their tasks accurately, but also operate reliably under dynamic and unpredictable conditions. Rather than relying solely on development-time testing, it is essential to verify at runtime whether the system behaves as expected during execution.

This work adopts a runtime assessment strategy based on continuous monitoring: the system observes sensor data and topic messages as the robot operates, checking compliance with a predefined set of functional and temporal requirements. These requirements are expressed in a human-readable YAML configuration file, allowing them to be defined independently of the robot's implementation.

The assessment begins as soon as the specified target node becomes active, and it terminates either when the node shuts down or the global timeout is reached. Throughout execution, the validator subscribes to relevant ROS topics and verifies whether the received messages satisfy the defined conditions.

This approach enables real-time tracking of compliance with expected behaviours during execution. While the assessment runs in parallel with the system, the results are automatically logged and summarised at the end of the mission. This improves the reliability and safety of autonomous robotic systems.

3.2 ROS1 to ROS2 Adaptation of the Runtime Assessment Tool

The runtime assessment system analysed in this work builds upon a pre-existing ROS1 implementation developed as part of the Rigel framework [28]. Its primary purpose was to monitor topic messages during a robot's execution and compare them against a list of predefined conditions, expressed in a YAML file. These included both functional objectives (e.g., "reach point X") and timing constraints (e.g., "within 60 seconds"). The tool was previously validated by means of a simple test scenario using the *Turtlesim* simulator.

The ROS1 version consisted of a central script (`main.py`) that coordinated the assessment process, supported by multiple modules, including `RuntimeAssessment.py` for managing evaluation logic, `AssessmentObjects.py` for defining the structure of events and validation requirements, `GlobalEvents.py` for centralising common definitions, and additional utilities, including `RuntimeAssessmentConfig.py`, `utils.py`, and the YAML specification file. Although functional, this ROS1 implementation was coupled to `rospy`[1] , the Python client library for ROS1, and relied on a centralised architecture using the ROS Master. This made it incompatible with ROS2, which adopts a decentralised, peer-to-peer model based on the DDS middleware. As a result, significant architectural changes were required to migrate the tool.

To support this transition, official documentation such as the *Migration Guide from ROS1* [29] was consulted extensively. It provided step-by-step instructions for replacing `rospy` with ROS2 equivalents in `rclpy`[2], guiding the restructuring and ensuring adherence to best practices.

The system was adapted to ROS2 by refactoring the ROS1 specific components to use rclpy, the ROS2 Python client library. This included updates to topic subscriptions, time handling, and node lifecycle management. The modular design of the original codebase facilitated the transition, although certain elements, such as logging and timer callbacks, required structural modifications.

Some examples of key differences encountered during the migration can be observed in Table 3.1.

Table 3.1: Example of changes during migration from ROS1 to ROS2

Function	**ROS1 (rospy)**	**ROS2 (rclpy)**
Topic Subscription	`rospy.Subscriber()`	`node.create_subscription()`
Create Publisher	`rospy.Publisher()`	`node.create_publisher()`
Create a rate	`rospy.Rate()`	`node.create_rate()`
Execution Loop	`while not rospy.is_shutdown():`	`while rclpy.ok():`
Get current time	`rospy.get_time()`	`node.get_clock().now()`

[1] http://wiki.ros.org/rospy. Accessed on: March 10, 2025.
[2] https://docs.ros.org/en/iron/p/rclpy/index.html. Accessed on: March 10, 2025.

3.3 System Architecture

The system is implemented as a standalone ROS2 node using `rclpy`, the Python client library for ROS2. During execution, it subscribes to relevant topics and evaluates received messages according to the criteria defined in a YAML specification file. The architecture is characterised by a loosely coupled, layered design, whereby configuration, runtime logic and evaluation modules are decoupled, thus promoting modularity and extensibility. The design of the system is such that it facilitates the incorporation of new types of requirements without the necessity of altering the system's core structure.

The following components constitute the core architecture:

- `RuntimeAssessment`: Manages the lifecycle of the assessment, monitors the presence of the target node, aggregates global metrics (like total execution time and message frequency), and coordinates the evaluation of both topic-specific and global requirements.
- `RuntimeAssessmentConfig`: Responsible for parsing the `specifications.yaml` file and loading its contents. It organizes the specifications into internal structures and provides configuration data to other components during initialization.
- `AssessmentObjects`: Each object is associated with a specific topic and is responsible for subscribing to it, recording incoming messages, and evaluating them against the requirements defined, such as value matching, temporal constraints, or absence of messages.
- `GlobalEvents`: Defines the enumeration of global system events that allow coordinated execution and lifecycle control across the validation components.
- `Utils`: Contains auxiliary utility functions used for message comparison, formatting, type checking, and data conversion across the assessment modules.
- `Main`: Initializes the ROS2 node, loads configuration data, and starts the validation cycle.

The overall interaction between these components is illustrated in Figure 3.1, which presents the conceptual architecture of the runtime assessment system and the data flow between configuration, orchestration, and validation modules.

Each `AssessmentObject` is linked to a specific topic and listens for matching messages. When a relevant message is received, the object checks whether it satisfies the expected content, order, and timing constraints. Upon completion of the execution process, the tool generates a report that outlines which requirements have been fulfilled and which remain unfulfilled.

This modular architecture ensures that new types of requirements can be added with minimal changes to the system's core logic, promoting reusability and extensibility in ROS2 environments.

Despite the architecture being based on a prior ROS1 implementation developed within the Rigel framework [15], the migration to ROS2 has introduced several changes. Certain modules were reused with minimal changes, while others were significantly adapted to comply with ROS2's distributed middleware and updated communication models.

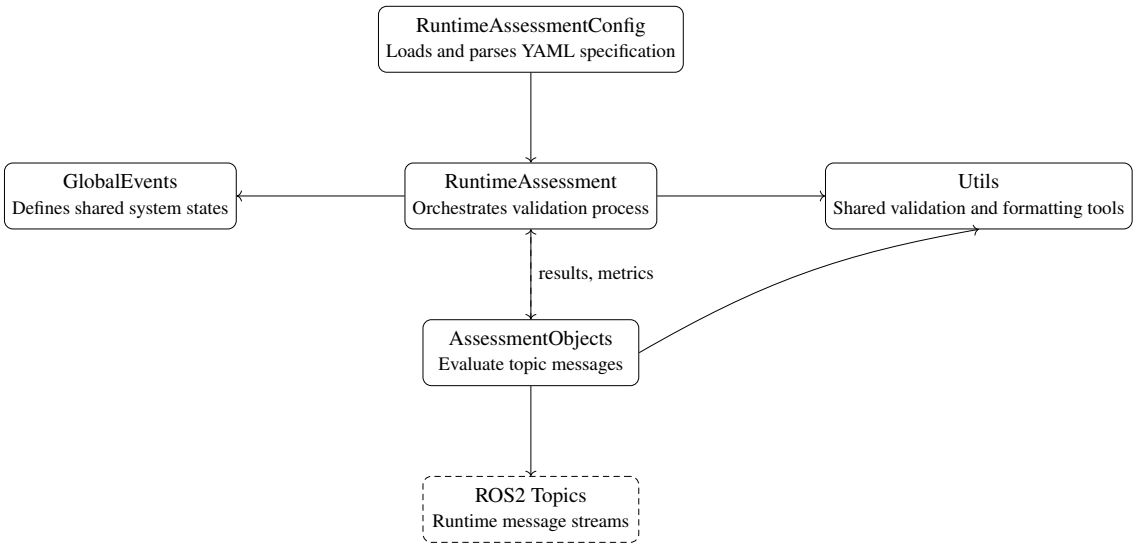


Figure 3.1: Conceptual architecture of the runtime assessment system, showing the interaction between core modules.

Table 3.2 summarises the contributions and adaptations made in each module during this migration process.

3.4 Specification Format and Assessment Logic

At the core of the runtime assessment mechanism lies the specification file, a human-readable YAML document that encodes the requirements to be verified during system execution. These specifications describe the expected behaviour by defining the messages that the system should publish or receive on specific topics, the time constraints within which these messages should occur, and the modes of evaluation to be applied (e.g., verify whether to confirm presence, absence, frequency or ordering of messages).

In addition to defining requirements, the specification file also contains a global `setup` section. This section defines general configuration parameters, present in Table 3.3, that govern the assessment process.

The field `global_timeout` is a feature that was not present in the initial ROS1 version of the tool. This feature was incorporated as a solution to ensure that assessments are concluded, even in cases where the target node remains active indefinitely, thus addressing a significant limitation that was identified during testing.

This setup ensures that the assessment node is properly initialised with awareness of which topics to subscribe, how to interpret the data, and where to record results for post-mission analysis.

Each entry in the `specifications` section corresponds to a topic to be monitored and includes parameters such as:

- `topic` (required): name of the ROS topic to monitor.
- `target` (required): the content or structure the messages should match.

Table 3.2: Summary of file-level contributions in the ROS1 to ROS2 migration

Component	Contribution
`RuntimeAssessment.py`	Adapted: updated imports, node creation logic, lifecycle handling and DDS-compliant timing.
`RuntimeAssessmentConfig.py`	Adapted: extended to support new YAML fields (e.g., `global_timeout`).
`AssessmentObjects.py`	Adapted: migrated to ROS2 with updated message handling and requirement verification logic.
`GlobalEvents.py`	Reused without changes: defines enumerations and constants independent of ROS version.
`utils.py`	Adapted: refactored spatial validation functions `unordered_points` and `ordered_points` to rely on 2D distance checking for pose evaluation, improving robustness and clarity.
`main.py`	Reused: no ROS-specific code required updates.

- `timein`, `timeout` (optional): time windows within which the message is expected. These can be omitted when timing is not relevant for a particular requirement.
- `mode` (optional): validation type (e.g., presence, absence). If omitted, the default is `exists`, which checks whether all defined targets are eventually observed.
- `temporal_consistency` (optional): whether the order or timing of messages must follow a defined sequence.
- `tolerance` (optional): margin allowed when comparing message values against expected targets.

Of these, only `topic` and `target` are mandatory. All other fields are optional and can be omitted depending on the nature of the requirement. This flexibility enables the specification of both minimal and advanced runtime checks, depending on the desired level of behavioural assurance.

3.4.1 Example Specification File

Listing 3.1 presents a simplified example of the YAML specification used for runtime assessment.

The `setup` section defines the global configuration of the assessment. The `target_node` field identifies that the ROS2 node to be monitored is `/snake_game`, while the results and logs will be stored after execution in the `log` directory. The `global_timeout` sets a maximum assessment duration of 90.0 s, ensuring the process terminates even if the node remains active indefinitely. Finally, the `topics` dictionary maps the topic names to their expected ROS2 message types, allowing the validator to subscribe correctly and interpret incoming messages.

Listing 3.1: Simplified YAML specification

```yaml
setup:
```

Table 3.3: `Setup` configuration parameters

Parameter	**Description**
`target_node`	Specifies the ROS2 node to be monitored during execution. (Reused from the original ROS1 version)
`logger_path`	Path where logs and results of the assessment will be stored. (Reused from the original ROS1 version)
`global_timeout`	Defines the maximum duration of the assessment process. If the target node remains active beyond this threshold, the assessment is forcefully terminated and results are logged. Otherwise, the process ends naturally when the target node shuts down. (Introduced in this work)
`topics`	Dictionary mapping the topics of interest to their respective assessment parameters. (Reused from the original ROS1 version)

```
    target_node: /snake_game
    logger_path: ../log
    global_timeout: 90.0
    topics:
        turtle1/pose: turtlesim/Pose

specifications:

    - topic: turtle1/pose
      temporal_consistency: true
      mode: exists
      tolerance: 0.5
      target:
        - { x: 10.5, y: 5.5 }
        - { x: 10.5, y: 10.5 }
        - { x: 5.5, y: 10.5 }
        - { x: 5.5, y: 5.5 }

    - topic: turtle1/pose
      timein: 1.0
      timeout: 30.0
      target:
        - { x: 10, y: 5.5 }

    - topic: turtle1/pose
      timein: 5.0
      timeout: 15.0
      tolerance: 0.5
      target:
        - { x: 10.5, y: 8 }
```

```
  - topic: turtle1/pose
    mode: metric
    comparator: ">"
    target:
      - number_of_messages: 1000
```

Each of these blocks defines a validation rule for monitoring messages on the `turtle1/pose` topic, showcasing different runtime assessment behaviours:

- **Requirement 1**: Verifies whether four specific 2D positions are eventually published in order. This combines the `exists` mode with `temporal_consistency:true` to assess both presence and sequence.
- **Requirement 2**: Checks that a given pose (x=10, y=5.5) occurs within a defined time window, between 1.0 s and 30.0 s after assessment begins.
- **Requirement 3**: Applies a spatial tolerance of 0.5 units, allowing approximate position matching. The message must arrive between 5.0 s and 15.0 s after start.
- **Requirement 4**: Uses the `metric` mode to assert that more than 1000 messages are received.

Each of these requirements is parsed by the `RuntimeAssessmentConfig` module and transformed into an internal object handled by the `AssessmentObjects` module. This structured and modular approach enables complex validation scenarios to be defined concisely and adjusted without requiring changes to the core logic of the validator.

3.5 Runtime Assessment Use Case: Turtlesim

After the adaptation, the ROS2 version was tested using the same *Turtlesim* use case, as its behaviour was already understood from the ROS1 implementation. This allowed a direct comparison, ensuring that the validation mechanism exhibited the same expected behaviour after migration.

Turtlesim[3] is a lightweight 2D simulator included in both ROS1 and ROS2 distributions, commonly used for teaching ROS. It features a cartoon turtle that moves in response to velocity commands, providing an accessible environment for testing ROS nodes and message flows. Because the simulator is available in both ROS versions, it was used to validate the functional equivalence of the assessment mechanism before and after migration. This ensured that the ROS2 implementation preserved the expected behaviour observed in ROS1.

The test scenario consisted of making the turtle perform a predefined shape, either a square or rectangle, using control scripts such as `TurtleSquare.py` or `TurtleRectangle.py`. These

[3] http://wiki.ros.org/turtlesim. Accessed on: February 20, 2025.

scripts published velocity commands to the `turtle1/cmd_vel` topic and emitted checkpoint signals when corners were reached, using the `turtle1/checkpoint` topic. Each checkpoint triggered a string message, such as "reached 1", enabling the assessment node to track mission progress in real time.

The assessment mechanism subscribed to `turtle1/pose` and `turtle1/checkpoint` to verify that the turtle reached specific positions within specified time intervals. In addition, it also subscribed to `turtle1/cmd_vel` and `/tf` to monitor velocity commands and transformation data.

All constraints were defined in the `specifications.yaml` file using a structured format, where each requirement was associated with an event type, a target value, and a time constraint. The full configuration used in this case study is provided in Appendix A.1.

Table 3.4: ROS topics monitored during the Turtlesim Assessment

Topic	**Message Type**	**Purpose**
`/turtle1/pose`	Pose	Track real-time robot position
`/turtle1/checkpoint`	String	Confirm reaching waypoints
`/turtle1/cmd_vel`	Twist	Monitor issued movement commands
`/tf`	TFMessage	Validate transformations and spatial trajectory

To verify the robustness of the assessment mechanism, both successful and failing trajectories were tested, enabling an assessment of precision and error handling.

Successful Scenario[4]

The specification file was configured to validate whether the turtle followed a perfect square trajectory, like the one presented in Figure 3.2. If the turtle deviated from the intended path or failed to reach any of the defined checkpoints within the expected time window, the validation would fail. This enabled the distinction between correct executions (squares) and incomplete or incorrect ones (rectangles).

The requirements that the validation node checks during execution are defined in a YAML configuration file. Four out of the seven conditions evaluated for the `/turtle1/pose` topic are shown in Listing 3.1. The complete specification, both for `/turtle1/pose` and other topics, is provided in Appendix A.1.

During execution, the validation node logs the results for each requirement. As demonstrated in Figure 3.3, the output indicates that all requirements defined for the `turtle1/pose` topic have been successfully met, resulting in an overall outcome of "PASSED". Table 3.5 summarises the outcome and purpose of each requirement evaluated for the `/turtle1/pose` topic.

[4] A recording of the successful execution is available at: `https://youtu.be/_gN2YOYj3Ao`.

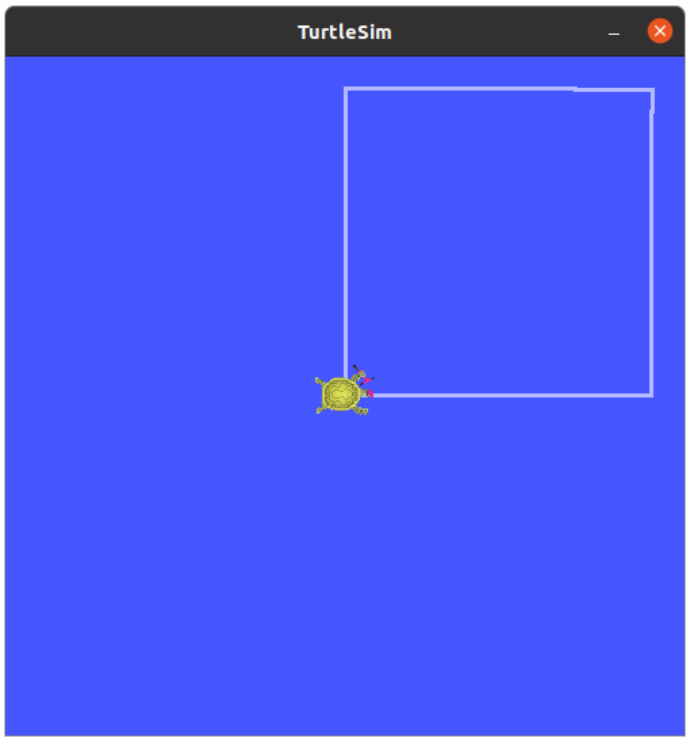


Figure 3.2: Example of a square trajectory followed by the turtle

Failure Scenario[5]

To illustrate a failure scenario, a variation of the test was performed where the turtle executed a rectangular path instead of the expected square. Figure 3.4 shows the resulting trajectory, clearly depicting the rectangular shape.

[5] A recording of the failed execution is available at: https://youtu.be/SXMCTguIjLs.

```
2025-03-31 10:31:04,518 - AssessmentObject.turtle1/pose - INFO - Global event received: NODE_REMOVED
2025-03-31 10:31:04,521 - AssessmentObject.turtle1/pose - INFO - Handling NODE_REMOVED event.
2025-03-31 10:31:04,521 - AssessmentObject.turtle1/pose - INFO - Number of events recorded: 2321
2025-03-31 10:31:04,521 - AssessmentObject.turtle1/pose - INFO - Frequency of events: 62.545743032714064 Hz
2025-03-31 10:31:04,522 - AssessmentObject.turtle1/pose - INFO - Checking requirements.
2025-03-31 10:31:04,539 - AssessmentObject.turtle1/pose - INFO - Requirement 1 of 7 PASSED.
2025-03-31 10:31:04,545 - AssessmentObject.turtle1/pose - INFO - Requirement 2 of 7 PASSED.
2025-03-31 10:31:04,547 - AssessmentObject.turtle1/pose - INFO - Requirement 3 of 7 PASSED.
2025-03-31 10:31:04,552 - AssessmentObject.turtle1/pose - INFO - Requirement 4 of 7 PASSED.
2025-03-31 10:31:04,553 - AssessmentObject.turtle1/pose - INFO - Requirement 5 of 7 PASSED.
2025-03-31 10:31:04,554 - AssessmentObject.turtle1/pose - INFO - Requirement 6 of 7 PASSED.
2025-03-31 10:31:04,554 - AssessmentObject.turtle1/pose - INFO - Requirement 7 of 7 PASSED.
2025-03-31 10:31:04,554 - AssessmentObject.turtle1/pose - INFO - Unregistering subscriber.
2025-03-31 10:31:04,554 - AssessmentObject.turtle1/pose - INFO - Assessment ended.
```

Figure 3.3: Validation log output indicating successful compliance with all requirements for the pose topic

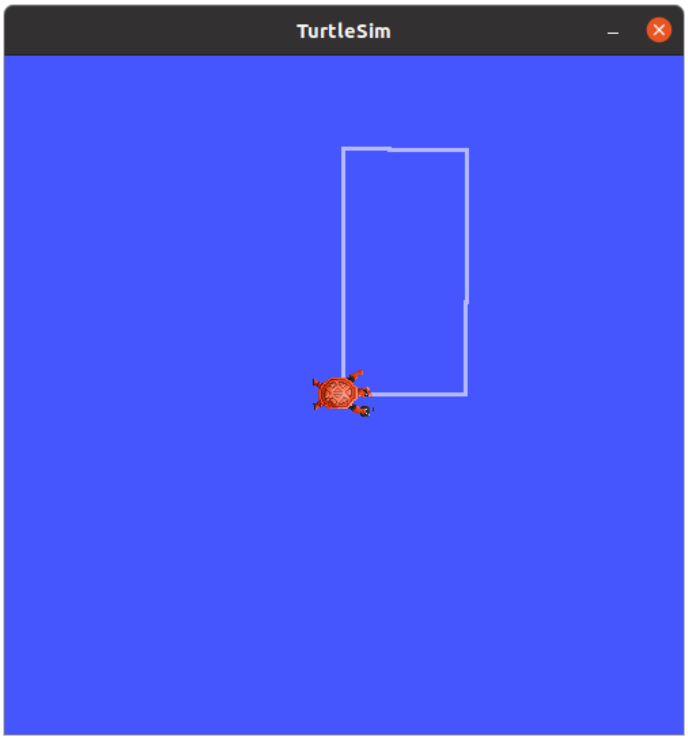


Figure 3.4: Example of a rectangle trajectory followed by the turtle

This deviation caused several requirements to fail. For instance, related to the `/turtle1/pose` topic, the recorded positions did not match the predefined target coordinates expected for a square, and the movement sequence did not respect the expected temporal ordering. Out of seven validation rules defined for this specific topic, represented in Appendix A.1, only three were satisfied.

As shown in the validation log (Figure 3.5), mismatches were detected in positional accuracy and ordering, and key points were absent. However, the absence of unintended positions was correctly confirmed, and message frequency metrics were validated. Table 3.6 summarises the individual `/turtle1/pose` topic results.

Table 3.5: Summary of validation results for the `/turtle1/pose` topic

Req. #	Result	Description
1	PASSED	Presence of messages at four expected corner positions of the square trajectory
2	PASSED	Turtle must follow waypoints in counter-clockwise sequence with spatial tolerance of 0.5
3	PASSED	Robot must reach point (10, 5.5) between 1.0 s and 30.0 s
4	PASSED	After 10 s, robot must no longer publish near (10, 5.5)
5	PASSED	Robot must reach (10.5, 8) between 5.0 s and 15.0 s, with tolerance of 0.5
6	PASSED	Messages near point (10.5, 10.5) should not appear during the initial 5.0 s
7	PASSED	At least 1000 messages must be published on the topic

```
2025-03-31 17:14:12,268 - AssessmentObject.turtle1/pose - INFO - Global event received: NODE_REMOVED
2025-03-31 17:14:12,270 - AssessmentObject.turtle1/pose - INFO - Handling NODE_REMOVED event.
2025-03-31 17:14:12,270 - AssessmentObject.turtle1/pose - INFO - Number of events recorded: 2526
2025-03-31 17:14:12,270 - AssessmentObject.turtle1/pose - INFO - Frequency of events: 62.525510235516265 Hz
2025-03-31 17:14:12,270 - AssessmentObject.turtle1/pose - INFO - Checking requirements.
2025-03-31 17:14:12,276 - AssessmentObject.turtle1/pose - INFO - Requirement 1 of 7 FAILED.
2025-03-31 17:14:12,286 - AssessmentObject.turtle1/pose - INFO - Requirement 2 of 7 FAILED.
2025-03-31 17:14:12,289 - AssessmentObject.turtle1/pose - INFO - Requirement 3 of 7 FAILED.
2025-03-31 17:14:12,291 - AssessmentObject.turtle1/pose - INFO - Requirement 4 of 7 PASSED.
2025-03-31 17:14:12,291 - AssessmentObject.turtle1/pose - INFO - Requirement 5 of 7 FAILED.
2025-03-31 17:14:12,292 - AssessmentObject.turtle1/pose - INFO - Requirement 6 of 7 PASSED.
2025-03-31 17:14:12,292 - AssessmentObject.turtle1/pose - INFO - Requirement 7 of 7 PASSED.
2025-03-31 17:14:12,292 - AssessmentObject.turtle1/pose - INFO - Unregistering subscriber.
2025-03-31 17:14:12,292 - AssessmentObject.turtle1/pose - INFO - Assessment ended.
```

Figure 3.5: Validation log output for the pose topic after rectangle trajectory

Table 3.6: Summary of validation results for the `/turtle1/pose` topic

Req. #	Result	Description
1	FAILED	Presence of messages at four expected corner positions of the **square** trajectory
2	FAILED	Turtle must follow the expected **square** corners in counter-clockwise sequence with spatial tolerance of 0.5
3	FAILED	Robot must reach point (10, 5.5) between 1.0 s and 30.0 s
4	PASSED	After 10 s, robot must no longer publish near (10, 5.5)
5	FAILED	Robot must reach (10.5, 8) between 5 s and 15 s, with tolerance of 0.5
6	PASSED	Messages near point (10.5, 10.5) should not appear during the initial 5 s
7	PASSED	At least 1000 messages must be published on the topic

This failure scenario demonstrates the system's ability to detect deviations from mission plans and enforce spatial and temporal constraints effectively.

Overall, the ROS2-based assessment tool was able to verify the turtle's trajectory against functional and temporal requirements with the same precision as in the ROS1 version. This confirms

that the adaptation preserved the expected behaviour while benefiting from ROS2's enhanced architectural flexibility.

3.6 Summary

This chapter presented the runtime assessment mechanism developed to evaluate ROS2 nodes against predefined behavioural specifications. The system operates through continuous monitoring of topic messages during execution, verifying functional and temporal requirements described in a human-readable YAML file. The mechanism was originally based on a ROS1 implementation integrated into the Rigel framework and was refactored to ROS2 using the `rclpy` library. This migration required modifications to the node lifecycle management, topic subscriptions, and timing logic, while maintaining the modular structure of the original tool.

The proposed architecture follows a layered and extensible design composed of several core modules, each with a specific responsibility within the assessment process. Requirements are expressed in a YAML specification file, whose global configuration section defines parameters such as the target node, log directory, global timeout, and monitored topics. Each topic entry contains individual rules describing message expectations, timing constraints, tolerances, and validation modes.

The assessment process terminates when the monitored node becomes inactive or when the predefined `global_timeout` is reached, ensuring that every evaluation concludes deterministically. The mechanism was validated using the Turtlesim simulator, confirming that the ROS2 implementation preserves the intended behaviour of the ROS1 version while benefiting from ROS2's improved modularity and communication model. Overall, this chapter established the foundations of a reusable and extensible runtime assessment framework for robotic applications in ROS2, which is further explored and integrated into containerised environments in the following chapter.

Chapter 4

Rigel Plugin for Runtime Assessment

The purpose of this Chapter is to present a detailed description of the integration of the runtime assessment into a containerised and automated pipeline using the Rigel framework. Building upon the runtime assessment system described in the Chapters 3 and 5, this Chapter focuses on extending that mechanism through custom plugins developed specifically for Rigel. These plugins enable the automatic orchestration, execution, and assessment of robotic missions, ensuring functional and temporal compliance with pre-defined specifications.

4.1 Overview

To enable the automated assessment of ROS2-based robotic applications within containerised environments, two Rigel plugins were developed:

- **Runtime Assessment Plugin:** This plugin encapsulates the runtime system described in Chapter 3. It is responsible for executing the verification of functional and temporal requirements based on a user-defined YAML file.

  The goal is to make runtime assessment as seamless as running a script: the developer defines the desired behaviour in the YAML specification file, and the plugin handles the execution, monitoring, and result reporting automatically.

- **Mission Deployment Plugin:** This plugin is responsible for launching the complete agricultural environment required for mission execution. It coordinates the deployment of the Gazebo simulation, the TRIBE Navigation Stack, and the mission configuration, including waypoints and mission files.

  To ensure correct and synchronised operation, the plugin manages the lifecycle of multiple containers and verifies the readiness of key ROS2 components before triggering the validation phase. This ensures that the system is fully initialised and stable before the runtime assessment begins. By automating these steps, the plugin enables consistent and reproducible deployment of robotic missions across different environments.

4.2 Runtime Assessment Plugin

The Runtime Assessment Plugin was developed as a complementary component within the Rigel execution pipeline. It extends the existing containerised workflow with runtime assessment capabilities.

This plugin is launched automatically during execution, without requiring manual activation. It is triggered alongside other mission components and performs real-time monitoring of system behaviour based on the specification defined by the user, as described in Chapter 3. This allows requirements to be assessed as the robot operates, ensuring assessment is embedded directly into the execution flow rather than added externally.

The implementation is defined in the `pluginAssessment.py` file, supported by assessment modules contained in the `assessment/` directory. The plugin follows the Rigel plugin model and is organised around a set of lifecycle methods that manage setup, execution, result extraction, and resource cleanup.

The plugin has been implemented as a Python class, name `AssessmentPlugin`, which extends Rigel's base `Plugin` class. It overrides the standard lifecycle methods `setup()`, `start()`, `process()` and `stop()` in order to manage the execution of the assessment logic. These functions are automatically invoked by the Rigel framework as part of the plugin's standard lifecycle. The plugin itself is registered in the `Rigelfile`, which determines when it is executed within the overall workflow.

The main responsibilities of the plugin include:

- **Parsing Configuration:** The plugin retrieves the path to the YAML specification file from Rigelfile and passes it to the `RuntimeAssessmentConfig` class, which is responsible for loading and interpreting the assessment configuration.
- **Executing the Assessment:** Upon activation, the plugin starts the runtime assessment process by instatiating the `RuntimeAssessment` class and calls its `run()` method. This launches the ROS2 assessment node, subscribes to specified topics and evaluates the message flow in real time, according to the requirements defined in the YAML file.
- **Assessment Log Path Handling:** The plugin extracts data from the YAML configuration file, specifically the `target_node` and `logger_path` fields, using created functions such as `get_target_node()` and `get_log_path_from_config()`. These values are used to compute the exact location of the assessment logs, which are saved under a folder named after the target node and current date (e.g., `slam_node_2025_06_22`). This mechanism ensures that log files are uniquely identified and easily retrievable after each execution.
- **Logging and Reporting:** After execution and retrieval of the exact log path, the plugin extracts summary statistics from the generated log file. These include total execution time, number of messages retrieved, average frequency, and the number of requirements passed or failed. This is done via the method `extract_last_assessment_log()`, which parses the latest assessment block in the log and aggregates relevant data.

The plugin follows Rigel's plugin lifecycle model and comprises the following core stages:

1. **Setup:** In the `setup()` function, the plugin verifies the presence of the YAML specification file and checks for the existence of the defined log path. In the event of misconfiguration, execution in terminated prematurely.

2. **Start:** The `start()` method is responsible for triggering the runtime assessment by invoking `run_assessment()`, which runs the complete monitoring process described in Chapter 3.

3. **Process:** Once the assessment ends, `process()` is responsible for parsing the log file and printing a summary of the results to the terminal. It computes the number of requirements that have been met and not met, and also extracts additional metrics, including the frequency of messages and the duration of execution.

4. **Stop:** The `stop()` function ensures that any remaining assessment resources are terminated, especially in instances where the node was still running at the time of shutdown.

The execution order of these lifecycle methods is illustrated in Figure 4.1, which shows the corresponding terminal output produced during a typical plugin run.

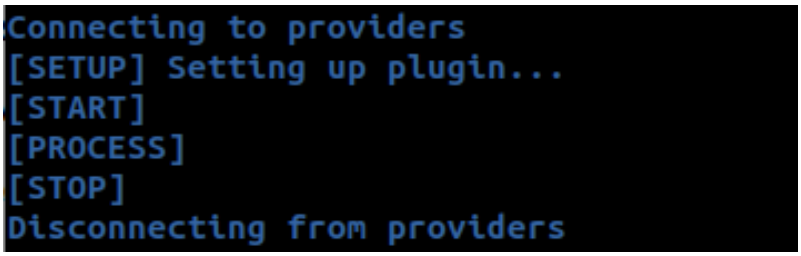


Figure 4.1: Example of terminal output illustrating the lifecycle execution of the Runtime Assessment Plugin (`setup`, `start`, `process`, and `stop`).

4.3 Mission Deployment Plugin

The *Mission Deployment Plugin* is responsible for launching the full simulation and navigation environment, which is required for the execution of the mission. It is responsible for the preparation of all necessary components, including Gazebo and the navigation stack through the orchestration of Docker Compose deployments.

The plugin is implemented in the file named `pluginLaunch.py` file. It inherits from Rigel's base `Plugin` class and overrides the core lifecycle methods: `setup()`, `start()`, and `stop()`. Although the `process()` method is defined, it remains intentionally empty, as it will not be invoked during execution. This behaviour is explained in Section 4.4, which details Rigel's concurrent execution model.

During the `setup()` phase, the plugin acquires one or more Docker Compose files paths from the `compose_files` parameter defined in the `Rigelfile`. Each path is resolved to an absolute location and stored internally for later use during container deployment.

In the `start()` function, the plugin launches all specified Docker Compose environments in the background, allowing the containers to start without blocking the execution of the assessment workflow. These environments may include the simulation world, ROS2 nodes, and other containers that are critical for the mission.

The plugin was designed to be adaptable to different robotic applications. Depending on the system, users may need to configure which container to monitor for readiness or adjust the method used to trigger the mission. For example, the command used to initiate the mission, usually `ros2 service call`, may need to match the specific service interface used by the target stack.

In the agricultural navigation use case presented in Chapter 5, the plugin monitors the container `hackathon_tribe_container` to identify a period of inactivity, indicating that the TRIBE Navigation Stack has successfully completed the initialisation process. Once the system is ready, the plugin calls a ROS2 service to launch the mission:

```
ros2 service call /motion_supervision_node/start_motion_mission std_srvs/
    srv/Empty
```

This triggers the start of the robotic mission as delineated in the configuration parameters of the container. Once the mission is launched, the plugin finishes its `start()` function, and the system is considered ready for the next stage of the pipeline: the runtime assessment of the mission.

The `process()` method is not implemented, since this plugin does not perform any runtime validation or logging. The `stop()` method is responsible for shutting down all previously launched containers.

This plugin enables the reproducible and automated deployment of complex robotic environments using containerised infrastructure. By abstracting the setup of simulation and mission execution into a Rigel plugin, it allows developers to integrate environmental initialisation directly into de pipeline.

4.4 Rigelfile Configuration and Execution Flow

The `Rigelfile` is the central configuration file used by Rigel to define and orchestrate the execution pipeline for a robotic application. It specifies the plugins to be used, their parameters, and the order in which they are executed. By describing the workflow declaratively in a YAML format, the `Rigelfile` allows developers to compose complex workflows from modular building blocks in a declarative and reproducible way.

The `Rigelfile` is written in YAML and is divided into two main sections:

- **Jobs (Plugin definitions):** This section declares the plugins to be used. Each plugin is assigned a unique job name and linked to its corresponding class implementation. The parameters passed to each plugin under the `with:` block are defined according to the specific requirements of that plugin.
- **Execution Sequences:** This section defines the order in which the jobs are executed. Each sequence is composed of multiple stages, where one or more jobs can run in parallel. Dependencies between stages ensure correct synchronisation.

Listing 4.1 shows the full `Rigelfile` used to launch the simulation environment and mission, and execute the runtime assessment plugin developed. While this configuration is tailored to the agricultural navigation use case detailed in Chapter 5, the plugins themselves are generic and reusable. Adapting the setup to other robotic missions primarily involves modifying the paths provided in the `compose_files` field, which specify the environments to be launched and the mission launch command. Other mission-specific parameters, such as the assessment specification file, should also be adjusted accordingly.

Listing 4.1: Complete `Rigelfile` used for launching the simulation and executing the runtime assessment.

```
---

vars:
  distro: "foxy"
  build_load: True
  build_push: False

application:
  distro: "{{ vars.distro }}"

jobs:
  launch_sim:
    plugin: "plugin_launch.pluginLaunch.MissionPlugin"
    with:
      compose_files:
        - "./FIRA/compose.yaml"
        - "./hackathon/compose.yaml"

  assessment:
    plugin: "plugin_assess.pluginAssessment.AssessmentPlugin"
    with:
      timeout: 1200
      config_file: "./requirements/specifications.yaml"

sequences:
```

```
  assess:
    stages:
      -
        jobs: ["assessment"]
        dependencies: ["launch_sim"]
```

In this configuration, the job entitled `launch_sim` invokes the Mission Deployment Plugin (Section 4.3), which is responsible for launching the simulation and navigation stack containers defined in the given `compose_files`, while the `assessment` job executes the Runtime Assessment Plugin (Section 4.2) based on the provided specification file. The sequence `assess` guarantees that the assessment will only run once the simulation is fully launched, as is required by the declared dependency.

Each plugin accepts parameters through the `with` block, which is defined per job. Table 4.1 summarises all parameters used in this project for both the Runtime Assessment and Mission Deployment plugins.

Table 4.1: Parameter defined in the `Rigerlfile`

Parameter	**Plugin**	**Mandatory**	**Description**
`compose_files`	Mission Deployment	Yes	List of file paths pointing to Docker Compose YAML files necessary to define the containers for simulation, navigation stack, and mission environment.
`timeout`	Runtime Assessment	No	Maximum execution duration for the plugin. Managed by Rigel.
`config_file`	Runtime Assessment	Yes	Path to the YAML specification file containing assessment requirements and setup information.

Rigel supports different execution modes within the `sequences` block: *sequential*, *concurrent*, and *parallel* [30]. This `Rigelfile` employs the **concurrent** mode, which allows multiple jobs to be defined independently, but executes them within a single thread, enforcing their declared dependencies without assigning separate execution threads.

The sequence `assess` is shown in the excerpt bellow:

Listing 4.2: Concurrent sequence used

```
sequences:
  assess:
    stages:
      -
        jobs: ["assessment"]
        dependencies: ["launch_sim"]
```

In this configuration, both `launch_sim` and `assessment` are part of the same execution thread. While only the `assessment` job is defined as a part of the current stage, the `launch_sim` job is listed as a dependency, therefore starting its execution earlier in the pipeline. This means Rigel ensures that the `assessment` job will only start after `launch_sim` has completed its initialisation, following a controlled and coordinated flow.

The actual execution order of the plugin lifecycle methods is:

Table 4.2: Execution order of plugin lifecycle functions for the `assess` sequence

Execution Order (Concurrent Mode)
MissionPLugin.setup()
MissionPLugin.start()
AssessmentPlugin.setup()
AssessmentPlugin.start()
AssessmentPlugin.process()
AssessmentPlugin.stop()
MissionPLugin.stop()

The job `launch_sim` is declared as a dependency and, by Rigel's execution logic, does not have its `process()` method called. Dependencies execute their `setup()` and `start()` methods at the beginning of the pipeline and are finalised with `stop()` after all dependent jobs complete, but they do not go through `process()`. Only jobs explicitly declared in the `jobs:` field of a stage will execute the full plugin lifecycle.

This concurrent model allows clear synchronisation of job execution while keeping plugin behaviour modular and predictable.

In practice, the termination of the assessment plugin is ensured either when the monitored node becomes inactive, indicating mission completion or abortion, or upon reaching the global timeout defined in the specification.

4.5 Execution of the Assessment Plugin

In Rigel, the execution of a plugin is prompted by running a named sequence defined in the `Rigelfile`. Each sequence describes the set of plugins to be executed, their order or dependencies, and the manner in which they should interact, as mentioned in Section 4.4. The command to initiate a sequence is formatted as follows:

```
rigel run sequence <sequence_name>
```

For this evaluation, the name of the sequence is designated `assess`, as shown in Listing 4.2. The execution was initiated from the root directory using the following command:

```
rigel run sequence assess
```

This sequence includes both the mission deployment and the assessment plugin. Rigel automatically handles the execution flow, waiting for system readiness, launching the mission, and subsequently starting the real-time assessment phase.

During execution, the assessment plugin outputs a summary of key runtime metrics to the terminal. These include:

- Total number of messages processed during mission;
- Total execution time;
- Message frequency;
- Number of requirements defined in the YAML specification file;
- Number of requirements passed and failed;
- Path to the full log file generated during assessment.

These outputs provide a concise overview of the system's runtime behaviour and are useful for quick validation. Detailed examples of such outputs are presented in Chapter 5, alongside the results obtained in simulation.

4.6 Summary

This chapter detailed the integration of the runtime assessment mechanism into the Rigel framework through the development of two dedicated plugins: the *Mission Deployment Plugin* and the *Runtime Assessment Plugin*. These components enable the automated deployment, execution, and validation of robotic missions within containerised environments, extending Rigel's functionality beyond build and launch management to include runtime verification.

The *Mission Deployment Plugin* automates the initialisation of the application containers under assessment, ensuring that all ROS2 components are correctly launched and ready before mission execution. The *Runtime Assessment Plugin* embeds the assessment logic into Rigel's execution pipeline, automatically validating mission behaviour in real time based on YAML-defined requirements. Their integration within the same execution sequence allows synchronised operation, where the assessment begins only after the application has been fully deployed.

Through this design, Rigel gains the capability to not only reproduce robotic missions reliably but also to evaluate their compliance with functional and temporal requirements in a fully automated manner. The following chapter applies this extended framework to agricultural navigation scenarios, demonstrating its effectiveness in realistic simulation workflows.

Chapter 5

Runtime Assessment in Agricultural Mission

This chapter presents the application of the runtime assessment in a realistic simulated agricultural navigation task, using the TRIBE navigation stack, developed at INESC TEC. The objective of this case study is to evaluate whether the assessment node can reliably verify task execution during a simulated agricultural mission.

5.1 Context and Environment

The navigation scenario is provided through the `hackathon2025_ws` workspace [31], developed for the FIRA 2025 Hackathon [24] and based on the TIRREX workspace [32], developed for the Roboterrium platform [33]. The environment includes a fully containerised simulation using Docker and Gazebo, with ROS2 Humble running inside a preconfigured image. This setup promotes reproducibility, simplifies deployment, and enables realistic evaluation of robotic behaviour in a simulated agricultural setting. Gazebo[1] is a widely used 3D simulator for robotics that enables realistic modelling of robots, environments, and sensor interactions.

The mission is carried out entirely in simulation and consists of a robot autonomously navigating through a sequence of predefined waypoints across a virtual field. The navigation logic is provided by the navigation stack developed by the TRIBE laboratory at INESC TEC, which was integrated into the environment to guide the robot along the intended trajectory.

The simulated platform reflects the structure of real agricultural robots, such as the Modular-E[2] robot, developed by INESC TEC and used in real-world field trials. The containerised setup allows for consistent repetition of experiments and facilitates integration with external planning and assessment tools.

[1] https://gazebosim.org/home. Accessed on: June 20, 2025
[2] https://world-fira.com/robots/modular-e/. Accessed on: June 20, 2025

The Gazebo simulation includes a detailed digital farm model composed of fields, warehouses, trees, and various static obstacles. The robot primarily operates around the central crop field, as depicted in Figure 5.1.

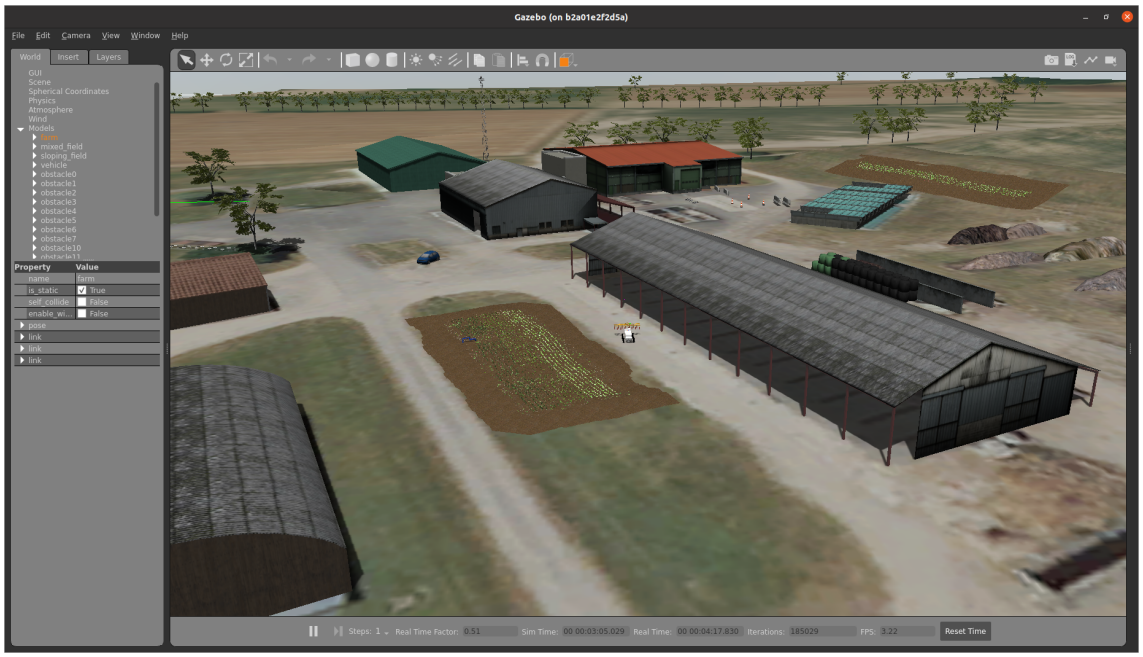


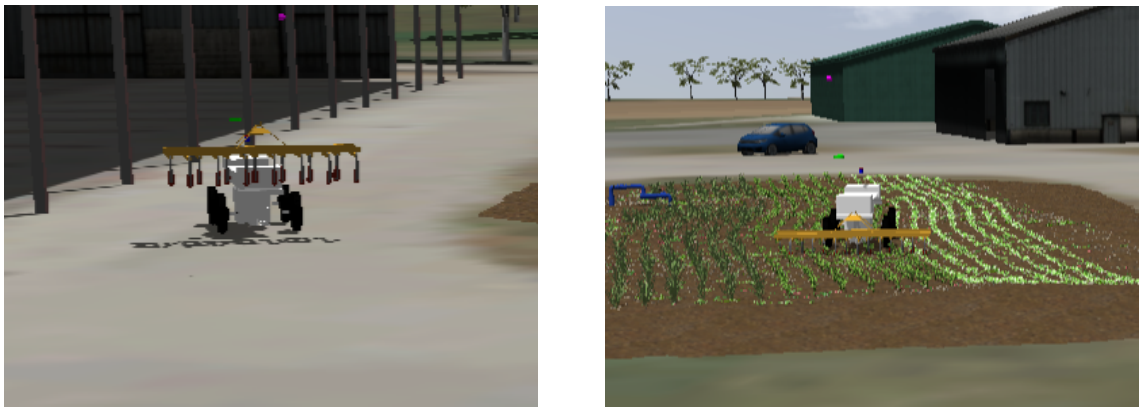
Figure 5.1: Gazebo simulation environment used in the mission.

5.2 Overview of the Assessment Scenarios

To evaluate the runtime assessment mechanism in realistic conditions, two validation scenarios were defined with increasing levels of complexity. The first scenario involved a short and controlled navigation trajectory in an obstacle-free area, designed to test basic functionality of the assessment pipeline. The second scenario represented a complete mission in the virtual field, with a larger number of waypoints and additional spatial and temporal constraints defined in the specification file.

By starting with a simplified case, the process could be incrementally verified and debugged. The more complex scenario then served to demonstrate the scalability and robustness of the assessment mechanism when exposed to longer missions.

In addition to verifying trajectory and timing, both scenarios included requirements related to the robot's tool. These constraints ensured that the tool remained raised before reaching the field zone and was only activated in the appropriate segment of the path. Figure 5.2 provides visual confirmation of the tool in both inactive and active states, as seen during mission's execution.



(a) Tool raised (inactive)

(b) Tool lowered (active)

Figure 5.2: Visual confirmation of tool state during the mission.

5.3 Scenario 1: Basic Navigation Assessment

5.3.1 Mission Description

In this initial validation scenario, the robot was assigned a short closed-loop trajectory composed of ten waypoints. The mission was designed to simulate a basic navigation task with minimal complexity, with an area free of obstacles and located near the starting zone of the simulation, serving as a controlled setting to verify the core behaviour of the assessment mechanism.

The robot begins near the starting zone and follows almost a straight line followed by a curved trajectory towards the entrance of the main crop field. Upon reaching the designated entry point, it proceeds along a straight segment that spans the length of the cultivated area. During this segment, the agricultural tool is lowered to perform simulated fieldwork, as illustrated in Figure 5.2b. After completing this part of the trajectory, the robot returns to the starting zone via a mirrored arc, completing the loop.

Figure 5.3 shows a visualisation of the planned path using RViz[3], a ROS visualisation tool commonly used to inspect robot trajectories and real-time state information in simulated environments. The waypoints, represented by orange markers, were preloaded into the navigation stack and the robot followed them autonomously.

This scenario was particularly useful for validating whether the robot correctly updated its pose along the path, respected time constraints at key waypoints, and activated or deactivated the agricultural tool at the expected locations.

[3] https://wiki.ros.org/rviz. Accessed on: June 21, 2025

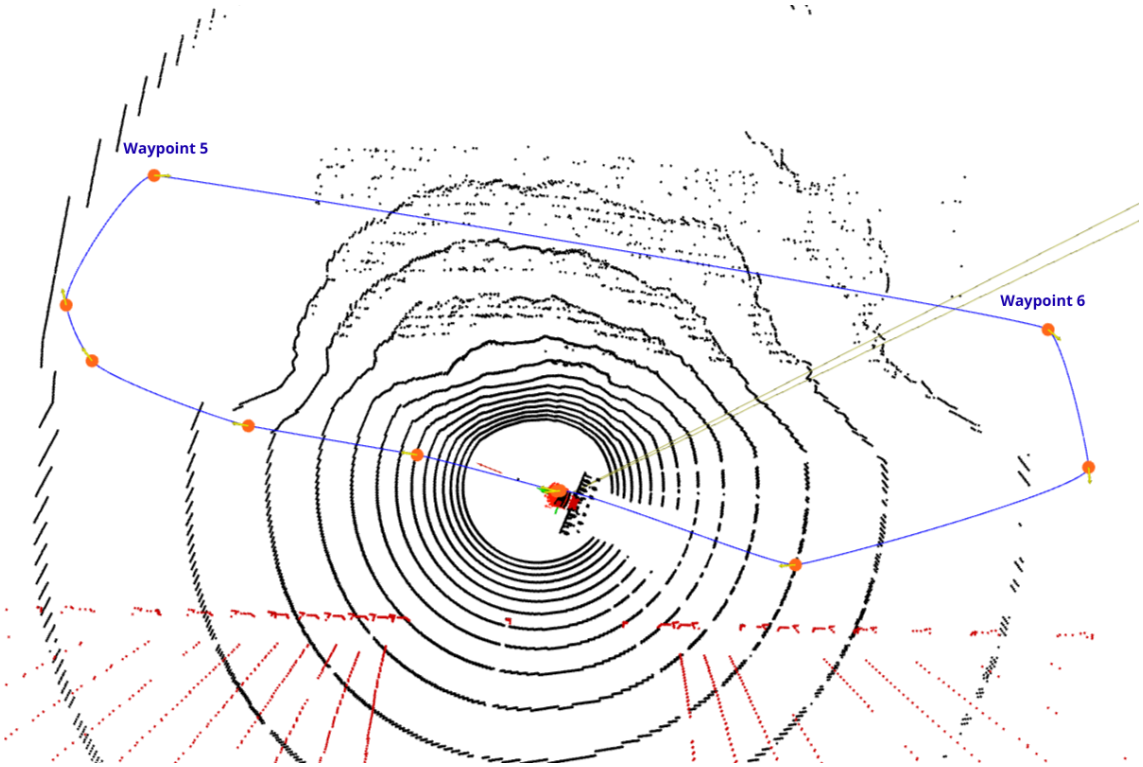


Figure 5.3: Planned navigation trajectory used in the first assessment scenario

5.3.2 Assessment Configuration

Following the structure in Section 3.4, the assessment was configured using a YAML specification file containing eleven scenario-specific requirements. These requirements targeted both topic-specific conditions and global execution metrics, targeting four main aspects of robot behaviour:

- **Waypoint tracking:** Ensuring that the robot reaches specific positions on the map, both in flexible order and in a predefined sequence;
- **Temporal Constraints:** Verifying that key waypoints, such as the entry into the field and the final destination, are reached within expected time windows;
- **Velocity checks:** Validating that the robot remains within safe dynamic bounds during movement;
- **Tool state transitions:** Confirming that the agricultural tool is only activated in the appropriate region.

Two time-bound requirements are particularly relevant: reaching waypoint 5 between 2 min and 3 min and waypoint 6 between 7 min and 9 min. These waypoints, identified in Figure 5.3, do not represent the exact points of entry and exit from the cultivated area, but rather act as temporal references that occur shortly before and after these transitions. As such, the time windows associated with them are deliberately wide, allowing the system to account for variations in execution

while still ensuring that the robot traverses the relevant operational region with temporal precision. Their validation helps correlate navigation behaviour with tool activation, without requiring the specification of exact spatial coordinates for the field boundaries.

To perform the assessment, three main ROS2 topics were subscribed: `/vineslam/pose`, for monitoring the robot's position estimated by the SLAM system; `/robot/localisation/filtered_odom`, for monitoring speed; and `/robot/implement/joint_states`, for checking the state of the tool. In addition, two global metrics were used to verify total execution time and the number of processed messages.

Table 5.1 summarises the requirements defined for this scenario, highlighting the type of validation and the criteria applied to each topic.

Table 5.1: Requirements for the first Agricultural Scenario

ID	Type	Topic/Metric	Description
1	Topic-level	`/vineslam/pose`	Waypoint 5 must be reached between 2 min and 3 min (with tolerance)
2	Topic-level	`/vineslam/pose`	Waypoint 6 must be reached between 7 min and 9 min
3	Topic-level	`/vineslam/pose`	Last waypoint must be reached
4	Topic-level	`/vineslam/pose`	Robot must pass through all nine waypoints, in any order
5	Topic-level	`/vineslam/pose`	Robot must pass through all waypoints in the predefined order
6	Topic-level	`/robot/localisation/filtered_odom`	Linear speed must never exceed 1.5 m/s
7	Topic-level	`/robot/localisation/filtered_odom`	Robot must move (speed above 0.1 m/s at any given second)
8	Topic-level	`/robot/implement/joint_states`	Agricultural tool must be inactive during initialisation (position < 0.5)
9	Topic-level	`/robot/implement/joint_states`	Agricultural tool must be activated upon entering the field (position > 0.8)
10	Global metric	`execution_time`	Total mission duration must not exceed 750 s
11	Global metric	`number_of_messages`	At least 7 000 messages must be processed during the mission.

The complete configuration for this scenario is documented in Appendix A.2.

The 7 000 message threshold used in requirement 11 was empirically determined based on observed mission behaviour. During test runs in which the robot failed to complete its path, such as cases where it got stuck, deviated from the trajectory, or was unable to recover after making a sharp turn, the number of processed messages consistently remained below this value. As such, it serves as a practical lower bound to indicate successful and continuous execution.

5.3.3 Results

The runtime assessment completed successfully the first scenario. As shown in the system log, Figure 5.4, all eleven requirements defined in the specification file were satisfied.

In total, 3377 messages were received from the `/robot/implement/joint_states` topic, 6082 messages from `/robot/localisation/filtered_odom`, and 1172 messages from `/vineslam/pose`, resulting in a combined total of 10 631 messages. This value exceeds the defined minimum of 7 000 messages, which serves as a practical threshold to distinguish successful missions from failed ones. The result was expected, given the publishing frequencies of the topics, approximately 5.55 Hz for the tool state, 1.93 Hz for the vineslam pose estimate, and 10.00 Hz for odometry. These frequencies ensured adequate temporal resolution for real-time validation.

The total execution time of the mission was 608.53 s, remaining within the allowed timeout. All spatial, temporal, and behavioural requirements were evaluated and marked as `PASSED`, as shown in Figure 5.4.

```
2025-06-27 07:38:47,961 - RuntimeAssessment./slam_node - INFO - Global timeout reached. Finishing assessment.
2025-06-27 07:38:47,964 - AssessmentObject./robot/implement/joint_states - INFO - Global event received: NODE_REMOVED
2025-06-27 07:38:47,968 - AssessmentObject./robot/implement/joint_states - INFO - Handling NODE_REMOVED event.
2025-06-27 07:38:47,968 - AssessmentObject./robot/implement/joint_states - INFO - Number of events recorded: 3377
2025-06-27 07:38:47,969 - AssessmentObject./robot/implement/joint_states - INFO - Frequency of events: 5.55419581258547 Hz
2025-06-27 07:38:47,969 - AssessmentObject./robot/implement/joint_states - INFO - Checking requirements.
2025-06-27 07:38:47,972 - AssessmentObject./robot/implement/joint_states - INFO - Requirement 1 of 2 PASSED.
2025-06-27 07:38:47,975 - AssessmentObject./robot/implement/joint_states - INFO - Requirement 2 of 2 PASSED.
2025-06-27 07:38:47,976 - AssessmentObject./robot/implement/joint_states - INFO - Unregistering subscriber.
2025-06-27 07:38:47,983 - AssessmentObject./robot/implement/joint_states - INFO - Assessment ended.

2025-06-27 07:38:47,985 - AssessmentObject./vineslam/pose - INFO - Global event received: NODE_REMOVED
2025-06-27 07:38:47,987 - AssessmentObject./vineslam/pose - INFO - Handling NODE_REMOVED event.
2025-06-27 07:38:47,987 - AssessmentObject./vineslam/pose - INFO - Number of events recorded: 1172
2025-06-27 07:38:47,988 - AssessmentObject./vineslam/pose - INFO - Frequency of events: 1.9285470148748614 Hz
2025-06-27 07:38:47,988 - AssessmentObject./vineslam/pose - INFO - Checking requirements.
2025-06-27 07:38:48,000 - AssessmentObject./vineslam/pose - INFO - Requirement 1 of 5 PASSED.
2025-06-27 07:38:48,009 - AssessmentObject./vineslam/pose - INFO - Requirement 2 of 5 PASSED.
2025-06-27 07:38:48,010 - AssessmentObject./vineslam/pose - INFO - Requirement 3 of 5 PASSED.
2025-06-27 07:38:48,092 - AssessmentObject./vineslam/pose - INFO - Requirement 4 of 5 PASSED.
2025-06-27 07:38:48,098 - AssessmentObject./vineslam/pose - INFO - Requirement 5 of 5 PASSED.
2025-06-27 07:38:48,098 - AssessmentObject./vineslam/pose - INFO - Unregistering subscriber.
2025-06-27 07:38:48,104 - AssessmentObject./vineslam/pose - INFO - Assessment ended.

2025-06-27 07:38:48,106 - AssessmentObject./robot/localisation/filtered_odom - INFO - Global event received: NODE_REMOVED
2025-06-27 07:38:48,112 - AssessmentObject./robot/localisation/filtered_odom - INFO - Handling NODE_REMOVED event.
2025-06-27 07:38:48,112 - AssessmentObject./robot/localisation/filtered_odom - INFO - Number of events recorded: 6082
2025-06-27 07:38:48,112 - AssessmentObject./robot/localisation/filtered_odom - INFO - Frequency of events: 10.001345775248874 Hz
2025-06-27 07:38:48,112 - AssessmentObject./robot/localisation/filtered_odom - INFO - Checking requirements.
2025-06-27 07:38:48,142 - AssessmentObject./robot/localisation/filtered_odom - INFO - Requirement 1 of 2 PASSED.
2025-06-27 07:38:48,175 - AssessmentObject./robot/localisation/filtered_odom - INFO - Requirement 2 of 2 PASSED.
2025-06-27 07:38:48,175 - AssessmentObject./robot/localisation/filtered_odom - INFO - Unregistering subscriber.
2025-06-27 07:38:48,180 - AssessmentObject./robot/localisation/filtered_odom - INFO - Assessment ended.

2025-06-27 07:38:48,199 - RuntimeAssessment./slam_node - INFO - Target node removed. Finishing assessment...
2025-06-27 07:38:48,199 - RuntimeAssessment./slam_node - INFO - Execution time: 608.5352244377136 seconds.
2025-06-27 07:38:48,199 - RuntimeAssessment./slam_node - INFO - Total number of messages: 10631
2025-06-27 07:38:48,199 - RuntimeAssessment./slam_node - INFO - Frequency: 17.469818628532213
2025-06-27 07:38:48,200 - RuntimeAssessment./slam_node - ERROR - This is a test error message.
2025-06-27 07:38:48,200 - RuntimeAssessment./slam_node - INFO - Assessing metric 'execution_time' with 1 requirements.
2025-06-27 07:38:48,200 - RuntimeAssessment./slam_node - INFO - Requirement 1 of 1 of metric 'execution_time' PASSED.
2025-06-27 07:38:48,200 - RuntimeAssessment./slam_node - INFO - Assessing metric 'number_of_messages' with 1 requirements.
2025-06-27 07:38:48,200 - RuntimeAssessment./slam_node - INFO - Requirement 1 of 1 of metric 'number_of_messages' PASSED.
2025-06-27 07:38:48,201 - RuntimeAssessment./slam_node - INFO - ----------------- END OF ASSESSMENT -----------------
```

Figure 5.4: Runtime assessment log results for Scenario 1.

These results confirm that the runtime assessment mechanism was able to accurately validate all defined requirements in a controlled setting, providing a reliable baseline for more complex mission evaluation.

5.4 Scenario 2: Complex Navigation Assessment

5.4.1 Mission Description

The second scenario evaluates the robot's ability to complete a significantly longer and more complex mission, set in the same simulated environment previously shown in Figure 5.1. While the surroundings remain unchanged, the mission's trajectory is expanded to introduce more varied conditions, testing the system's response to signal loss and unexpected physical constraints.

The complete path consists of 49 waypoints, shown in Figure 5.5, representing a continuous navigation route that initiates in an open area, traverses a cultivated field, and was intended to continue past the warehouse into a more structured environment. However, due to physical obstructions placed within the warehouse area, the robot becomes immobilised and is unable to complete the mission. This outcome was expected and served to test the robustness of the runtime assessment system. No changes were made to the robot's navigation stack, allowing the failure to emerge naturally from the increased complexity of the environment. As a result, this scenario provides a valuable opportunity to observe how the assessment mechanism responds to mission failure, specifically by detecting unfulfilled requirements and deviations from the expected temporal and spatial behaviours.

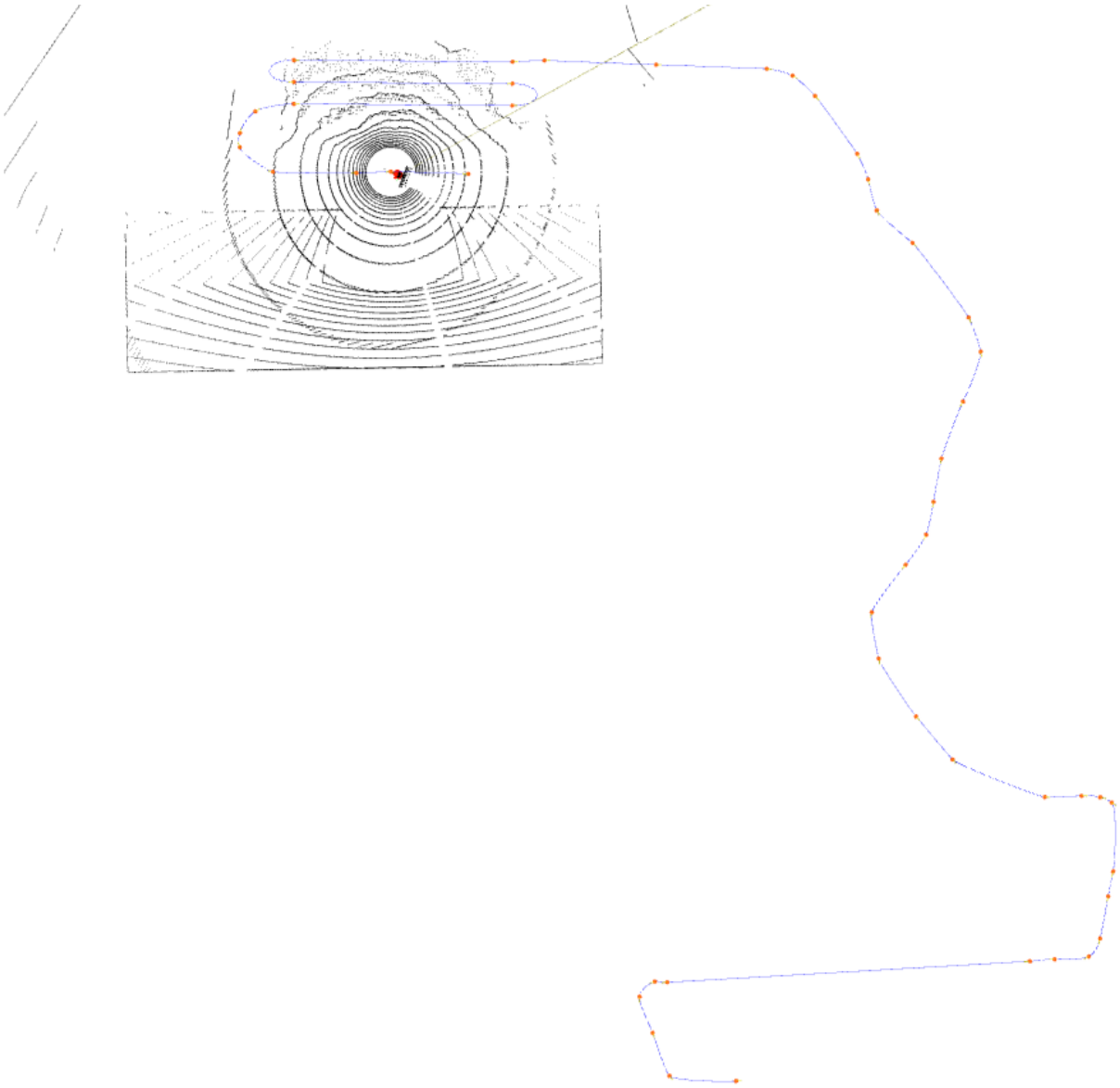


Figure 5.5: Trajectory of the second scenario

In order to facilitate targeted assessment, the mission is logically divided into four segments of different colours, each of which corresponds to a distinct phase of the trajectory, as illustrated in Figure 5.6.

- **Blue segment:** The mission begins with a linear segment, succeeded by a right curve that prepares the robot for entry into the field. During this segment, the tool remains inactive (Figure 5.2a). The robot moves at approximately 0.6 m/s on the straight section and slows to 0.4 m/s on the curve. As this phase approaches its conclusion, the tool begins its descent in preparation of activation.

- **Green segment:** This section covers navigation inside the cultivated area. The path alternates between straight segments and tight curves, and the robot moves at a velocity of 0.6 m/s along the straight crop rows and reduces its speed to 0.4 m/s when executing the tight turns. During this phase, the tool is fully deployed.

- **Orange segment:** After leaving the field, the robot travels along a straight corridor and makes a wide turn to align itself with the warehouse entrance. In this phase, the robot moves consistently at 0.6 m/s and the tool is expected to retract during this segment.

- **Final segment (without coloured box):** Upon entering the warehouse, the robot loses GPS signal and is physically blocked by an object, preventing further progress and successful completion of the this segment, allowing the assessment system to detect unfulfilled requirements and validate its response to unexpected runtime conditions.

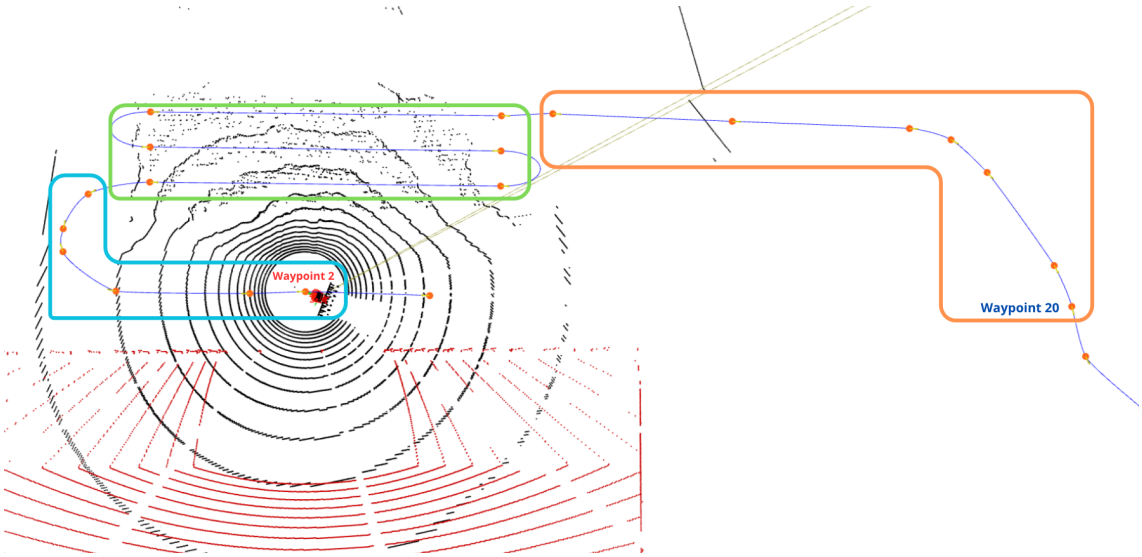


Figure 5.6: Segmented view of the trajectory of Scenario 2

The target speeds for each segment, typically 0.6 m/s for straight paths and 0.4 m/s for tighter curves, were predefined in the mission configuration files that specify the waypoints.

From approximately waypoint 20 onward, the robot enters the warehouse. At this point, two critical issues arise: first, the GPS signal is lost due to the enclosed space, making it impossible to assess requirements based on absolute coordinates; and second, the obstruction of the robot's path by a pallet placed on the ground, preventing it from exiting the warehouse. As a consequence, it is anticipated that all runtime requirements pertaining to this final segment of the mission will be unsuccessful.

5.4.2 Assessment Configuration

The second scenario was assessed using a YAML configuration that defines seventeen requirements targeting spatial navigation, actuator behavioural and global mission metrics. These were organised according to the mission's logical segmentation (blue, green, orange and unreachable) with each segment corresponding to a distinct portion of the robot's trajectory, as illustrated in Figure 5.6.

The robot's mission is comprised of 49 waypoints in total. However, the robot is initialised directly at the second waypoint. Consequently, the first assessment rule has been configured to verify that the robot did not start at the initial position (waypoint 1), confirming that it correctly spawned in proximity to the second waypoint. All subsequent requirements assume that the robot commences its trajectory from the second recorded point, which is the first to be assessed from the actual trajectory.

In all `exists` requirements related to waypoint assessment, the field `temporal_consistency: true` was applied. This enforces that the target positions must be reached sequentially and in order, rather than in arbitrary sequence, strengthening the traceability of the robot's path through the mission.

Blue Segment – Initial Approach

This segment corresponds to the beginning of the trajectory, as the robot departs from the initial area and moves towards the agricultural field. The requirements include:

- A group of `exist` rules is in place to evaluate the presence of six waypoints between 0 and 125 s;
- The implementation of an `average` constraint on the tool joint position, ensuring that it remained below 0.5 (i.e. inactive)
- An `average` constraint on the robot's velocity, which must remain under or equal to 0.6 m/s during this phase.

Green Segment – Field Operation

This phase encompasses the active traversal of the agricultural field. It is anticipated that the robot will navigate through a total of six waypoints and activate its tool, like the representation in Figure 5.2b. The following set of rules were monitored:

- A second group of `exist` requirements for six waypoints, with a time window between 120 s and 1068 s;
- An `average` constraint verifying that the tool was actively engaged (position > 0.8);
- A new `average` constraint that confirms that the robot maintained a reasonable speed of below 0.6 m/s through the field operation.

Orange Segment – Transition to Warehouse

This segment serves to assess the robot's approach to the warehouse. All target waypoints are located before the entrance, and it is expected that these will be successfully reached. The following specifications are incorporated:

- Seven waypoints are assessed using the `exist` mode between 1040 s and 1300 s;
- A `max` mode constraint on the tool position (< 0.5) has been established, confirming it's deactivation;
- The robot is required to maintain an `average` speed of 0.6 m/s, with a small tolerance.

Unreachable Segment

As mentioned in Subsection 5.4.1, the robot was not expected to reach these positions, as they are situated within and beyond the warehouse, due to a loss of GPS signal and an obstruction, a pallet, that hinders further movement.

To assess the system's ability to reason about such mission failures, two complementary rules were defined: one with the mode set to `exist`, anticipated to fail, since the robot never reached those points of the trajectory; and another with `mode:absent`, anticipated to pass, confirming the robot did not proceed beyond the blockage. This dual-rule strategy allows runtime assessment to detect trajectory incompletion, even in the absence of reliable positional data.

Global Mission Metrics

The assessment is concluded with five mission-wide evaluations:

- The maximum velocity of the robot is constrained to 1.0 m/s, with a tolerance margin of 0.05 m/s;
- In order to confirm that the robot moved, it is necessary that the speed is above 0.1 m/s;
- Total duration must exceed 1000 s;
- Between 10 000 and 20 000 messages are processed.
- Topic frequency remains above 10 Hz.

These metrics are used to ensure that the robot's runtime context was valid and that the assessment was conducted over a representative and stable execution period.

Table 5.2: Requirements for the second Agricultural Scenario

ID	Type	Segment	Topic/Metric	Description
1	Topic-level	initial point	`/vineslam/pose`	Confirm absence of waypoint 1 (spawn check)
2	Topic-level	blue	`/vineslam/pose`	Waypoints must be reached before field entry
3	Topic-level	blue	`/robot/implement/joint_states`	Tool must be inactive
4	Topic-level	blue	`/robot/localisation/filtered_odom`	Speed below 0.6 m/s during approach
5	Topic-level	green	`/vineslam/pose`	Field waypoints must all be reached
6	Topic-level	green	`/robot/implement/joint_states`	Tool must be active in the field
7	Topic-level	green	`/robot/localisation/filtered_odom`	Speed below 0.6 m/s during field traversal
8	Topic-level	orange	`/vineslam/pose`	Waypoints before warehouse entrance
9	Topic-level	orange	`/robot/implement/joint_states`	Tool must be inactive after leaving the field
10	Topic-level	orange	`/robot/localisation/filtered_odom`	Speed 0.6 m/s before warehouse
11	Topic-level	unreachable	`/vineslam/pose`	Points that should not be reached (expected `exists` failure)
12	Topic-level	unreachable	`/vineslam/pose`	Points that should not be reached (expected `absent` success)
13	Topic-level	global	`/robot/localisation/filtered_odom`	Robot must stay under 1.0 m/s
14	Topic-level	global	`/robot/localisation/filtered_odom`	Robot must move (speed above 0.1 m/s)
15	Global metric	global	`execution_time`	Execution time must exceed 1 000 s
16	Global metric	global	`number_of_messages`	10 000 - 20 500 messages must be processed
17	Global metric	global	`frequency`	Topic frequency must exceed 10 Hz

5.4.3 Results

The runtime assessment for Scenario 2 was executed to completion. The results obtained are summarised below based on three output categories: topic-specific validations, global metrics, and failure analysis.

The **odometry topic** (`/robot/localisation/filtered_odom`) recorded a total of 12 792 messages with a frequency of 10 Hz. While no specific message count was predefined, this result aligns with the expected behaviour for odometry data, which typically publishes at higher rates to support continuous motion tracking. All five requirements associated with this topic were successfully passed, including constraints related to maximum and minimum speeds throughout the mission.

The **positioning topic** (`/vineslam/pose`) recorded a total of 2 259 messages with a frequency of 1.67 Hz. Out of six evaluated requirements, five were successfully passed. The sole failure was observed in **Requirement 5**, **ID 11** in Table 5.2, which corresponds to the segment following the entrance to the warehouse (segment: *unreachable*). As previously discussed, the robot becomes immobilised and unable to access GPS signals within the warehouse, thus failing to satisfy the expected positional condition defined with `mode: exists`.

The **implement topic** (`/robot/implement/joint_states`) registered 5 364 messages, characterised by a lower frequency of 3.97 Hz. This reduced rate is consistent with the nature of actuator state updates, which change less frequently than localisation data and do not require high-frequency publishing. The assessment of all three requirements associated with this topic confirmed the successful transitions of the agricultural tool state before, during and after the cultivated field.

Global metrics confirmed proper mission duration and logging consistency:

- **Execution time:** 1350.95 s (`Requirement 1 of 1 PASSED`);
- **Number of messages:** 20 415 total (`Requirement 1 of 1 PASSED`)
- **Frequency:** 15.11 Hz (`Requirement 1 of 1 PASSED`).

Figure 5.7 presents the assessment log, with each requirement colour-coded according to its associated mission segment, as defined in Table 5.2. This visual mapping clarifies the outcome of each requirement in the context of the mission. Notably, Requirement 5 of `/vineslam/pose`, marked in the red segment, failed as expected due to the robot being immobilised after entering the warehouse, while all other requirements passed successfully.

```
2025-06-28 10:32:58,068 - RuntimeAssessment./slam_node - INFO - Global timeout reached. Finishing assessment.
2025-06-28 10:32:58,150 - AssessmentObject./robot/localisation/filtered_odom - INFO - Global event received: NODE_REMOVED
2025-06-28 10:32:58,178 - AssessmentObject./robot/localisation/filtered_odom - INFO - Handling NODE_REMOVED event.
2025-06-28 10:32:58,178 - AssessmentObject./robot/localisation/filtered_odom - INFO - Number of events recorded: 12792
2025-06-28 10:32:58,179 - AssessmentObject./robot/localisation/filtered_odom - INFO - Frequency of events: 10.00053460060701 Hz
2025-06-28 10:32:58,179 - AssessmentObject./robot/localisation/filtered_odom - INFO - Checking requirements.
2025-06-28 10:32:58,201 - AssessmentObject./robot/localisation/filtered_odom - INFO - Requirement 1 of 5 PASSED.
2025-06-28 10:32:58,294 - AssessmentObject./robot/localisation/filtered_odom - INFO - Requirement 2 of 5 PASSED.
2025-06-28 10:32:58,323 - AssessmentObject./robot/localisation/filtered_odom - INFO - Requirement 3 of 5 PASSED.
2025-06-28 10:32:58,411 - AssessmentObject./robot/localisation/filtered_odom - INFO - Requirement 4 of 5 PASSED.
2025-06-28 10:32:58,515 - AssessmentObject./robot/localisation/filtered_odom - INFO - Requirement 5 of 5 PASSED.
2025-06-28 10:32:58,515 - AssessmentObject./robot/localisation/filtered_odom - INFO - Unregistering subscriber.
2025-06-28 10:32:58,525 - AssessmentObject./robot/localisation/filtered_odom - INFO - Assessment ended.

2025-06-28 10:32:58,527 - AssessmentObject./vineslam/pose - INFO - Global event received: NODE_REMOVED
2025-06-28 10:32:58,531 - AssessmentObject./vineslam/pose - INFO - Handling NODE_REMOVED event.
2025-06-28 10:32:58,531 - AssessmentObject./vineslam/pose - INFO - Number of events recorded: 2259
2025-06-28 10:32:58,532 - AssessmentObject./vineslam/pose - INFO - Frequency of events: 1.67418670396174 Hz
2025-06-28 10:32:58,532 - AssessmentObject./vineslam/pose - INFO - Checking requirements.
2025-06-28 10:32:58,553 - AssessmentObject./vineslam/pose - INFO - Requirement 1 of 6 PASSED.
2025-06-28 10:32:58,561 - AssessmentObject./vineslam/pose - INFO - Requirement 2 of 6 PASSED.
2025-06-28 10:32:58,739 - AssessmentObject./vineslam/pose - INFO - Requirement 3 of 6 PASSED.
2025-06-28 10:32:58,936 - AssessmentObject./vineslam/pose - INFO - Requirement 4 of 6 PASSED.
2025-06-28 10:32:58,965 - AssessmentObject./vineslam/pose - INFO - Requirement 5 of 6 FAILED.
2025-06-28 10:32:58,993 - AssessmentObject./vineslam/pose - INFO - Requirement 6 of 6 PASSED.
2025-06-28 10:32:58,993 - AssessmentObject./vineslam/pose - INFO - Unregistering subscriber.
2025-06-28 10:32:59,004 - AssessmentObject./vineslam/pose - INFO - Assessment ended.

2025-06-28 10:32:59,005 - AssessmentObject./robot/implement/joint_states - INFO - Global event received: NODE_REMOVED
2025-06-28 10:32:59,016 - AssessmentObject./robot/implement/joint_states - INFO - Handling NODE_REMOVED event.
2025-06-28 10:32:59,016 - AssessmentObject./robot/implement/joint_states - INFO - Number of events recorded: 5364
2025-06-28 10:32:59,016 - AssessmentObject./robot/implement/joint_states - INFO - Frequency of events: 3.972692306302164 Hz
2025-06-28 10:32:59,017 - AssessmentObject./robot/implement/joint_states - INFO - Checking requirements.
2025-06-28 10:32:59,028 - AssessmentObject./robot/implement/joint_states - INFO - Requirement 1 of 3 PASSED.
2025-06-28 10:32:59,057 - AssessmentObject./robot/implement/joint_states - INFO - Requirement 2 of 3 PASSED.
2025-06-28 10:32:59,074 - AssessmentObject./robot/implement/joint_states - INFO - Requirement 3 of 3 PASSED.
2025-06-28 10:32:59,075 - AssessmentObject./robot/implement/joint_states - INFO - Unregistering subscriber.
2025-06-28 10:32:59,083 - AssessmentObject./robot/implement/joint_states - INFO - Assessment ended.

2025-06-28 10:32:59,102 - RuntimeAssessment./slam_node - INFO - Target node removed. Finishing assessment...
2025-06-28 10:32:59,102 - RuntimeAssessment./slam_node - INFO - Execution time: 1350.9520692825317 seconds.
2025-06-28 10:32:59,103 - RuntimeAssessment./slam_node - INFO - Total number of messages: 20415
2025-06-28 10:32:59,103 - RuntimeAssessment./slam_node - INFO - Frequency: 15.111564994931365
2025-06-28 10:32:59,104 - RuntimeAssessment./slam_node - ERROR - This is a test error message.
2025-06-28 10:32:59,111 - RuntimeAssessment./slam_node - INFO - Assessing metric 'execution_time' with 1 requirements.
2025-06-28 10:32:59,111 - RuntimeAssessment./slam_node - INFO - Requirement 1 of 1 of metric 'execution_time' PASSED.
2025-06-28 10:32:59,112 - RuntimeAssessment./slam_node - INFO - Assessing metric 'number_of_messages' with 1 requirements.
2025-06-28 10:32:59,112 - RuntimeAssessment./slam_node - INFO - Requirement 1 of 1 of metric 'number_of_messages' PASSED.
2025-06-28 10:32:59,113 - RuntimeAssessment./slam_node - INFO - Assessing metric 'frequency' with 1 requirements.
2025-06-28 10:32:59,113 - RuntimeAssessment./slam_node - INFO - Requirement 1 of 1 of metric 'frequency' PASSED.
2025-06-28 10:32:59,113 - RuntimeAssessment./slam_node - INFO - ---------- END OF ASSESSMENT ----------
```

Figure 5.7: Runtime assessment log results for Scenario 2. Requirements are colour-coded according to the corresponding mission segment: pink (initial point), blue (blue segment), green (green segment), orange (orange segment), red (unreachable segment), and yellow (global requirements).

The output confirms that Rigel correctly enforces spatial, temporal, and semantic rules in real time. The only anticipated failure (Requirement 5 of 6) illustrates that unattainable objectives are accurately identified, thereby substantiating the assessment system capacity to handle partial mission completion and environmental limitations.

5.5 Runtime Assessment Plugin Execution

To demonstrate the integration of the runtime assessment mechanism in the Rigel pipeline, the execution of the `assessment` plugin was executed as part of the evaluation for Scenario 1. This example corresponds to the basic agricultural navigation task described in Section 5.3.

The plugin was triggered using the `assess` sequence defined in the `Rigelfile`, as detailed in Section 4.5. This sequence combines both the mission deployment and the assessment phases, allowing the plugin to monitor system behaviour in real time.

Figure 5.8 presents the corresponding resulting terminal output. It shows the plugin lifecycle, including container launch, the service call that starts the mission, and the assessment phase.

```
WARNING - Press CTRL-C / CTRL-Z to stop execution
Connecting to providers
[SETUP] Launching containers.
[INFO] Containers launched.
[START] Preparing to launch mission.
[+] Running 1/1
 ✔ Container hackathon_tribe_container  Started
[+] Running 2/2
 ✔ Container hackathon-simulator-1      Started
 ✔ Container hackathon-localisation-1   Started
[INFO] Waiting for tribe container to go idle...
[INFO] Logs have been idle for 10 seconds.
[INFO] Launching mission...
[INFO] Mission launched in container 'hackathon_tribe_container'.
[SETUP] Setting up assessment...
Path to requirements (specifications.yaml): /home/ana/Documents/specifications.yaml
Log path defined in configuration file: ./log
[START] Runtime Assessment started
Runtime Assessment Plugin running
Published Global Event: GlobalEvents.NODE_ADDED
Published Global Event: GlobalEvents.NODE_REMOVED
[PROCESS] Processing assessment...
Messages: 9441 | Execution time: 610.0028579235077s | Frequency: 15.476976668827131 Hz

 Total number of requirements:  11

 Number of requirements passed:  11

 Number of requirements failed:  0

 Full logs available at:
  ./log/slam_node_2025_06_27/slam_node_assessment.log
[STOP] Stop Runtime Assessment Plugin
[STOP] All containers stopped.
Disconnecting from providers
```

Figure 5.8: Terminal output from a plugin execution

The output in Figure 5.8 corroborates that the execution of the plugin followed the anticipated lifecycle order presented in Table 4.2. Only the `assessment` job executed the `process()` function, while the dependency (`launch_sim` job) finalised the sequence with `stop()`, as expected in Rigel's concurrent execution model.

In this particular instance, a total of 11 runtime requirements were evaluated, as detailed in Appendix A.2, with all of them being successfully met. Furthermore, the runtime assessment plugin reported that 9441 messages were received over approximately 610 seconds, yielding a message frequency of 15.47 Hz. These metrics demonstrate that the system exhibited stable runtime behaviour throughout the mission.

At the end of execution, the plugin outputs the location of the full assessment log file:

`./log/slam_node_2025_06_27/slam_node_assessment.log`

These results validate the correct operation of both plugins and confirm that the integrated pipeline can reliably deploy, execute, and assess agricultural robotic missions in an automated and reproducible manner.

5.6 Summary

This chapter applied the runtime assessment mechanism to a realistic simulated agricultural navigation mission based on the TRIBE stack, demonstrating its ability to validate behaviour under containerised, reproducible conditions. Two scenarios of increasing complexity were executed in Gazebo: a short baseline trajectory used to verify core functionality, and an extended route segmented into distinct phases designed to stress spatial and temporal constraints, velocity bounds, and tool-state transitions. Requirements were specified in YAML and assessed in real time using ROS2 topic data, correlating waypoint progression, speed constraints, and activation of the agricultural tool with the different phases of the mission.

In the first scenario, all requirements were satisfied and runtime metrics (message counts, frequency, and total duration) confirmed stable behaviour. In the second scenario, the assessment correctly identified the expected failure after the robot entered the warehouse, where GPS was unavailable and a physical obstruction prevented progress, flagging the unreachable goals while validating the absence of unintended positions. The results show that the mechanism enforces spatial, temporal, and semantic constraints in real time, distinguishes successful from incomplete missions, and remains robust to degraded sensing conditions.

Overall, the study validates the suitability of runtime assessment for agricultural navigation tasks, providing actionable, mission-level feedback and logs that support analysis and iteration. The next chapter integrates this mechanism into the Rigel framework via dedicated plugins, enabling automated deployment and assessment within a unified, container-based execution pipeline.

Chapter 6

Conclusions

6.1 Outcomes

This dissertation addressed the problem of validating ROS2 applications in the context of agricultural robotics, a field characterised by unpredictable conditions, variable mission goals, and dynamic environments. In such scenarios, runtime assessment becomes a critical tool for ensuring system correctness, since traditional static verification methods are insufficient to capture temporal or context-dependent faults.

To respond to these challenges, this work introduced a runtime assessment plugin for ROS2, integrated into the existing Rigel container-framework. The plugin enables assessment of robotic behaviour during execution, using externally defined YAML specifications that describe functional and temporal requirements. These are evaluated in real time over the ROS2 message streams, allowing failures to be detected during mission runtime.

The developed plugin was integrated into an end-to-end workflow and evaluated in two realistic agricultural navigation scenarios. Results confirmed the system's ability to:

- Assess topic-level behaviour such as waypoint traversal, tool state transitions and motion constraints;
- Detect deviations from expected performance in real time;
- Correctly report failures, for example when robot becomes immobilised or when an agricultural tool is not working.

These results demonstrate that the runtime assessment can be effectively applied to containerised ROS2 systems, supporting modularity, reproducibility and improved software reliability. The use of declarative validation rules, combined with integration into Rigel's plugin system, also improves the accessibility and scalability of this approach in multi-component robotic systems.

6.2 Research Objectives and Questions Revisited

This section reflects on the main objectives defined in Section 1.3, and how their fulfilment addressed the core research questions of this dissertation.

The first objective was to research and develop a runtime assessment mechanism capable of monitoring compliance with spatial and temporal requirements during execution. This was achieved through the implementation of a ROS2-compatible plugin that observes message flows in real time and validates them against mission-specific constraints expressed in a structured YAML format.

Another objective focused on integrating this plugin within Rigel, a container-based framework. This integration enabled automated execution and validation workflows in isolated, reproducible environments, which is a feature that is particularly beneficial for systematic testing of robotic systems.

Finally, the proposed approach was validated in two realistic agricultural navigation scenarios of increasing complexity. The results demonstrated the solution's ability to detect deviations, confirm correct behaviour, and adapt to different mission profiles without compromising runtime performance.

Through the successful fulfilment of these objectives, the dissertation answered the two research questions posed: (RQ1) by demonstrating that runtime assessment can be effectively performed in ROS2 agricultural scenarios using a plugin-based structure that supports scalability; and (RQ2) by showing that containerisation supports reproducibility and isolation in runtime assessment workflows.

6.3 Limitations and Future Work

Despite the successful implementation and validation process for the runtime assessment plugin, several limitations were identified during the course of development and experimentation. These, in turn, highlight directions for future improvement.

- **Topic-Centric Observability:** The current model relies exclusively on ROS topic data for inference. This limits the system's visibility to message-level behaviour and may overlook internal node states, logic errors, or coordination failures that do not manifest at the topic interface.
- **Fixed Temporal Constraints:** Time windows for requirement assessment are specified manually and remain static. In practical missions, execution times may vary, especially in cases involving obstacle avoidance or detours. This can lead to false negatives when the robot arrives at a target point later than expected due to external interference, even if the trajectory is valid.

Future improvements could involve support for adaptive or relative temporal analysis, where delays are not treated as binary failures but instead quantified and reported. Rather than classifying a requirement as simply passed or failed based on fixed thresholds, the plugin could record that a condition was fulfilled with delay, explicitly reporting the deviation from the expected time window.

This would allow developers to distinguish between tolerable delays and true execution faults, enabling more informative runtime diagnostics while maintaining behavioural assessment.

References

[1] V. Marinoudi, C. G. Sørensen, S. Pearson, and D. Bochtis, "Robotics and labour in agriculture. a context consideration," *Biosystems Engineering*, vol. 184, pp. 111–121, 2019, ISSN: 1537-5110. DOI: https://doi.org/10.1016/j.biosystemseng.2019.06.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1537511019303617.

[2] R. Caldas, J. A. P. García, M. Schiopu, P. Pelliccione, G. Rodrigues, and T. Berger, "Runtime verification and field-based testing for ROS-based robotic systems," *IEEE Transactions on Software Engineering*, vol. 50, no. 10, pp. 2544–2567, Oct. 2024, ISSN: 2326-3881. DOI: 10.1109/tse.2024.3444697. [Online]. Available: https://ieeexplore.ieee.org/document/10638820.

[3] C. Cheng, J. Fu, H. Su, and L. Ren, "Recent advancements in agriculture robots: Benefits and challenges," *Machines*, vol. 11, no. 1, 2023, ISSN: 2075-1702. DOI: 10.3390/machines11010048. [Online]. Available: https://www.mdpi.com/2075-1702/11/1/48.

[4] L. Benos, C. Sørensen, and D. Bochtis, "Field deployment of robotic systems for agriculture in light of key safety, labor, ethics and legislation issues," *Current Robotics Reports*, vol. 3, pp. 49–56, Jun. 2022. DOI: 10.1007/s43154-022-00074-9. [Online]. Available: https://doi.org/10.1007/s43154-022-00074-9.

[5] A. Bechar and C. Vigneault, "Agricultural robots for field operations: Concepts and components," *Biosystems Engineering*, vol. 149, pp. 94–111, 2016, ISSN: 1537-5110. DOI: https://doi.org/10.1016/j.biosystemseng.2016.06.014. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1537511015301914.

[6] L. Santos, F. Santos, J. Mendes, *et al.*, "Path planning aware of robot's center of mass for steep slope vineyards," *Robotica*, vol. 38, no. 4, pp. 684–698, 2020. DOI: 10.1017/S0263574719000961. [Online]. Available: https://www.cambridge.org/core/journals/robotica/article/path-planning-aware-of-robots-center-of-mass-for-steep-slope-vineyards/2D301E1A70D8231B668A7745F9CEA8DF.

[7] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in *2016 International Conference on Embedded Software (EMSOFT)*, 2016, pp. 1–10. DOI: 10.1145/2968478.2968502. [Online]. Available: https://ieeexplore.ieee.org/document/7743223.

[8] A. Bonci, F. Gaudeni, M. C. Giannini, and S. Longhi, "Robot operating system 2 (ROS2)-based frameworks for increasing robot autonomy: A survey," *Applied Sciences*, vol. 13, no. 23, 2023, ISSN: 2076-3417. DOI: 10.3390/app132312796. [Online]. Available: https://www.mdpi.com/2076-3417/13/23/12796.

[9] A. Afzal, C. Le Goues, M. Hilton, and C. S. Timperley, "A study on challenges of testing robotic systems," in *2020 IEEE 13th international conference on software testing, validation and verification (ICST)*, IEEE, 2020, pp. 96–107. DOI: 10.1109/ICST46399.2020.00020. [Online]. Available: https://ieeexplore.ieee.org/document/9159069.

[10] F. Bonsignorio and A. P. del Pobil, "Toward replicable and measurable robotics research [from the guest editors]," *IEEE Robotics & Automation Magazine*, vol. 22, no. 3, pp. 32–35, 2015. DOI: 10.1109/MRA.2015.2452073. [Online]. Available: https://ieeexplore.ieee.org/document/7254310.

[11] A. Santos, A. Cunha, N. Macedo, and C. Lourenço, "A framework for quality assessment of ROS repositories," in *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2016, pp. 4491–4496. DOI: 10.1109/IROS.2016.7759661. [Online]. Available: https://ieeexplore.ieee.org/document/7759661.

[12] A. Santos, A. Cunha, and N. Macedo, "The high-assurance ROS framework," Jun. 2021, pp. 37–40. DOI: 10.1109/RoSE52553.2021.00013. [Online]. Available: https://ieeexplore.ieee.org/document/9474545/.

[13] J. Huang, C. Erdogan, Y. Zhang, *et al.*, "ROSRV: Runtime verification for robots," in *Runtime Verification*, B. Bonakdarpour and S. A. Smolka, Eds., Cham: Springer International Publishing, 2014, pp. 247–254, ISBN: 978-3-319-11164-3. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-319-11164-3_20.

[14] I. Perez, A. Mavridou, T. Pressburger, A. Will, and P. J. Martin, "Monitoring ROS2: From requirements to autonomous robots," *Electronic Proceedings in Theoretical Computer Science*, vol. 371, pp. 208–216, Sep. 2022, ISSN: 2075-2180. DOI: 10.4204/eptcs.371.15. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.371.15.

[15] P. Melo, R. Arrais, S. Teixeira, and G. Veiga, "A container-based framework for developing ROS applications," in *2022 IEEE 20th International Conference on Industrial Informatics (INDIN)*, 2022, pp. 280–285. DOI: 10.1109/INDIN51773.2022.9976083. [Online]. Available: https://ieeexplore.ieee.org/document/9976083.

[16] J. Smith and R. Nair, "The architecture of virtual machines," *Computer*, vol. 38, pp. 32–38, Jun. 2005. DOI: 10.1109/MC.2005.173. [Online]. Available: https://ieeexplore.ieee.org/document/1430629.

[17] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, and N. Thoai, "Using docker in high performance computing applications," in *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, 2016, pp. 52–57. DOI: 10.1109/CCE.2016.7562612. [Online]. Available: https://ieeexplore.ieee.org/document/7562612.

[18] A. Pörtner, M. Hoffmann, S. Zug, and M. König, "Swarmrob: A docker-based toolkit for reproducibility and sharing of experimental artifacts in robotics research," English, in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, ser. IEEE International Conference on Systems, Man, and Cybernetics Proceedings, United States: IEEE, 2018, pp. 325–332. DOI: 10.1109/SMC.2018.00065. [Online]. Available: https://ieeexplore.ieee.org/document/8616060.

[19] F. Lumpp, M. Panato, F. Fummi, and N. Bombieri, "A container-based design methodology for robotic applications on kubernetes edge-cloud architectures," in *2021 Forum on specification & Design Languages (FDL)*, 2021, pp. 01–08. DOI: 10.1109/FDL53530.2021.9568376. [Online]. Available: https://ieeexplore.ieee.org/document/9568376.

[20] F. Lumpp, M. Panato, N. Bombieri, and F. Fummi, "A design flow based on docker and kubernetes for ROS-based robotic software applications," *ACM Trans. Embed. Comput. Syst.*, vol. 23, no. 5, Aug. 2024, ISSN: 1539-9087. DOI: 10.1145/3594539. [Online]. Available: https://doi.org/10.1145/3594539.

[21] J. Cañas Plaza, E. Perdices, L. García-Pérez, and J. Fernández-Conde, "A ROS-based open tool for intelligent robotics education," *Applied Sciences*, vol. 10, p. 7419, Oct. 2020. DOI: 10.3390/app10217419. [Online]. Available: https://www.mdpi.com/2076-3417/10/21/7419.

[22] T. Yamamoto, K. Terada, A. Ochiai, F. Saito, Y. Asahara, and K. Murase, "Development of human support robot as the research platform of a domestic mobile manipulator," *ROBOMECH Journal*, vol. 6, no. 1, p. 4, Apr. 2019. DOI: 10.1186/s40648-019-0132-3. [Online]. Available: https://robomechjournal.springeropen.com/articles/10.1186/s40648-019-0132-3.

[23] Y. Huang, T. Chuang, N. Lin, *et al.*, "Team NCTU: toward ai-driving for autonomous surface vehicles - from duckietown to robotx," *CoRR*, vol. abs/1910.14540, 2019. arXiv: 1910.14540. [Online]. Available: http://arxiv.org/abs/1910.14540.

[24] F. H. 2025, *Fira hackathon 2025 official page*, https://firahackathon.github.io/#, Accessed: June 19, 2025, 2025.

[25] P. Melo, R. Arrais, and G. Veiga, "Development and deployment of complex robotic applications using containerized infrastructures," in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, 2021, pp. 1–8. DOI: 10.1109/INDIN45523.2021.9557386. [Online]. Available: https://ieeexplore.ieee.org/document/9557386.

[26] INESC TEC, *TRIBE - Laboratory of Robotics and IoT for Smart Precision Agriculture and Forestry*, Accessed: February 2025. [Online]. Available: https://www.inesctec.pt/en/laboratories/tribe-laboratory-of-robotics-and-iot-for-smart-precision-agriculture-and-forestry.

[27] L. C. Santos, A. S. Aguiar, F. N. Santos, A. Valente, J. B. Ventura, and A. J. Sousa, "Navigation stack for robots working in steep slope vineyard," in *Intelligent Systems and Applications*, K. Arai, S. Kapoor, and R. Bhatia, Eds., Cham: Springer International Publishing, 2021, pp. 264–285, ISBN: 978-3-030-55180-3. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-55180-3_21.

[28] *ROS1 runtime assessment*, https://github.com/MisterOwlPT/runtime_assessment, Accessed on: Februray 15, 2025, 2025.

[29] Open Robotics, *Migration guide from ROS1*, https://docs.ros.org/en/foxy/The-ROS2-Project/Contributing/Migration-Guide.html, Accessed: 2025-04-07.

[30] *Plugin compliant with new features. issue #4*, https://github.com/Kazadhum/file_intro_plugin/issues/4, Accessed: June 2025, 2023.

[31] FIRA 2025 Hackathon Workspace, *hackathon2025_ws: ROS2 Simulation Workspace for FIRA 2025 Hackathon*, https://github.com/FiraHackathon/hackathon2025_ws, Accessed: 2025-06-09, 2025.

[32] TIRREX Project Team, *Tirrex_workspace: ROS2 Workspace for the Roboterrium Platform*, https://github.com/Tirrex-Roboterrium/tirrex_workspace, Accessed: June 19, 2025, 2024.

[33] TIRREX Project, *Roboterrium platform*, https://tirrex.fr/plateforme/roboterrium/, Accessed: June 19, 2025, 2024.

Appendix A

Full YAML Specification

This appendix includes the full `specifications.yaml` file used during the assessment scenarios. It contains all configuration fields and validation rules for the runtime assessment.

A.1 Turtlesim Scenario

Listing A.1: Complete specifications.yaml file used for Turtlesim example

```yaml
setup:
  target_node: /snake_game
  logger_path: ../log
  global_timeout: 90.0
  topics:
    turtle1/pose: turtlesim/Pose
    turtle1/cmd_vel: geometry_msgs/Twist
    turtle1/checkpoint: std_msgs/String
    tf: tf2_msgs/TFMessage

specifications:

  - topic: tf
    temporal_consistency: true
    tolerance: 0.5
    target:
      - { "transforms[0].transform.translation.x": 10.5, "transforms[0].
          transform.translation.y": 5.5 }
      - { "transforms[0].transform.translation.x": 10.5, "transforms[0].
          transform.translation.y": 10.5 }
      - { "transforms[0].transform.translation.x": 5.5, "transforms[0].
          transform.translation.y": 10.5 }
      - { "transforms[0].transform.translation.x": 5.5, "transforms[0].
          transform.translation.y": 5.5 }

```

```yaml
- topic: tf
  timein: 1.0
  timeout: 30.0
  target:
    - { "transforms[0].transform.translation.x": 10.5, "transforms[0].
        transform.translation.y": 5.5 }

- topic: tf
  mode: absent
  timein: 10.0
  target:
    - { "transforms[0].transform.translation.x": 10, "transforms[0].
        transform.translation.y": 5.5 }

- topic: tf
  mode: absent
  timeout: 5.0
  target:
    - { "transforms[0].transform.translation.x": 10, "transforms[0].
        transform.translation.y": 10 }

- topic: tf
  mode: metric
  comparator: "<"
  target:
    - number_of_messages: 10


- topic: turtle1/pose
  mode: exists # default value // or max, min, exists, absent, average,
      metric
  target:
    - { x: 10.5, y: 5.5 }
    - { x: 5.5, y: 5.5 }
    - { x: 5.5, y: 10.5 }
    - { x: 10, y: 10.5 }


- topic: turtle1/pose
  temporal_consistency: true
  tolerance: 0.5
  target:
    - { x: 10.5, y: 5.5 }
    - { x: 10.5, y: 10.5 }
    - { x: 5.5, y: 10.5 }
```

```
63        - { x: 5.5, y: 5.5 }
64
65    - topic: turtle1/pose
66      timein: 1.0
67      timeout: 30.0
68      target:
69        - { x: 10, y: 5.5 }
70
71    - topic: turtle1/pose
72      mode: absent
73      timein: 10.0
74      target:
75        - { x: 10, y: 5.5 }
76
77    - topic: turtle1/pose
78      timein: 5.0
79      timeout: 15.0
80      tolerance: 0.5
81      target:
82        - { x: 10.5, y: 8 }
83
84    - topic: turtle1/pose
85      mode: absent
86      timeout: 5.0
87      target:
88        - { x: 10.5, y: 10.5 }
89
90    - topic: turtle1/pose
91      mode: metric
92      comparator: ">"
93      target:
94        - number_of_messages: 1000
95
96    - topic: turtle1/cmd_vel
97      mode: average
98      comparator: "<"
99      target:
100       - linear.x: 1.5
101
102   - topic: turtle1/cmd_vel
103     mode: max
104     comparator: ">"
105     target:
106       - angular.z: 0.2
107
```

```yaml
- topic: turtle1/cmd_vel
  mode: max
  target:
    - linear.x:
      - min: 0.5
      - max: 1.5

- topic: turtle1/cmd_vel
  mode: min
  comparator: "!="
  target:
    - linear.x: 0.1

- topic: turtle1/cmd_vel
  mode: metric
  target:
    - frequency:
      - min: 55
      - max: 65

- topic: turtle1/checkpoint
  target:
    - data: "reached 1"
    - data: "reached 2"
    - data: "reached 3"
    - data: "reached 4"

- topic: turtle1/checkpoint
  temporal_consistency: true
  target:
    - data: "reached 1"
    - data: "reached 2"
    - data: "reached 3"
    - data: "reached 4"

- topic: turtle1/checkpoint
  timein: 2
  timeout: 6
  target:
    - data: "reached 1"

- topic: turtle1/checkpoint
  mode: absent
  target:
    - data: "reached 5"
```

```
153
154   - topic: turtle1/checkpoint
155     mode: metric
156     comparator: "<"
157     target:
158       - number_of_messages: 5
159
160   - metric: "execution_time"
161     comparator: "<"
162     target: 80.0
163
164   - metric: "number_of_messages"
165     target:
166       - min: 1000
167       - max: 10000
168
169   - metric: "frequency"
170     comparator: ">"
171     target: 60
```

A.2 Simple Agricultural Scenario

Listing A.2: Complete specifications.yaml file used for the first agricultural scenario

```
1  setup:
2    target_node: /slam_node
3    logger_path: ./log
4    global_timeout: 750
5    topics:
6      /robot/implement/joint_states: sensor_msgs/JointState
7      /robot/localisation/filtered_odom: nav_msgs/Odometry
8      /vineslam/pose: geometry_msgs/PoseStamped
9
10
11 specifications:
12
13   - topic: /vineslam/pose
14     timein: 120
15     timeout: 180
16     tolerance: 0.5
17     target:
18       - { pose.position.x: 130.76, pose.position.y: 117.08 }
19
20   - topic: /vineslam/pose
21     timein: 420
```

```yaml
    timeout: 540
    tolerance: 0.5
    target:
      - { pose.position.x: 89.89, pose.position.y: 108.24 }

  - topic: /vineslam/pose
    timein: 600
    timeout: 750
    tolerance: 0.8
    target:
      - { pose.position.x: 107.969, pose.position.y: 123.534 }

  - topic: /vineslam/pose
    tolerance: 0.8
    target:
      - { pose.position.x: 114.700, pose.position.y: 124.475 }
      - { pose.position.x: 122.419, pose.position.y: 126.142 }
      - { pose.position.x: 130.222, pose.position.y: 126.063 }
      - { pose.position.x: 132.291, pose.position.y: 124.128 }
      - { pose.position.x: 130.764, pose.position.y: 117.079 }
      - { pose.position.x: 89.890, pose.position.y: 108.242 }
      - { pose.position.x: 85.787, pose.position.y: 113.419 }
      - { pose.position.x: 96.640, pose.position.y: 122.643 }
      - { pose.position.x: 107.969, pose.position.y: 123.534 }

  - topic: /vineslam/pose
    temporal_consistency: true
    tolerance: 0.8
    target:
      - { pose.position.x: 114.700, pose.position.y: 124.475 }
      - { pose.position.x: 122.419, pose.position.y: 126.142 }
      - { pose.position.x: 130.222, pose.position.y: 126.063 }
      - { pose.position.x: 132.291, pose.position.y: 124.128 }
      - { pose.position.x: 130.764, pose.position.y: 117.079 }
      - { pose.position.x: 89.890, pose.position.y: 108.242 }
      - { pose.position.x: 85.787, pose.position.y: 113.419 }
      - { pose.position.x: 96.640, pose.position.y: 122.643 }
      - { pose.position.x: 107.969, pose.position.y: 123.534 }

  - topic: /robot/localisation/filtered_odom
    mode: max
    comparator: "<="
    target:
      - { twist.twist.linear.x: 1.5 }
    tolerance: 0.05
```

```yaml

  - topic: /robot/localisation/filtered_odom
    mode: max
    comparator: ">"
    target:
      - { twist.twist.linear.x: 0.1 }
    tolerance: 0.05

  - topic: /robot/implement/joint_states
    mode: max
    timein: 0
    timeout: 90
    comparator: "<"
    target:
      - position[0]: 0.5

  - topic: /robot/implement/joint_states
    mode: min
    timein: 120
    timeout: 540
    comparator: ">"
    target:
      - position[0]: 0.80

  - metric: "execution_time"
    comparator: "<="
    target: 750.0

  - metric: "number_of_messages"
    target:
      - min: 7000
      - max: 20000
```