FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Dedicated Scheduler for Kubernetes in Robotics Applications with ROS/ROS2

David José Soares Castro

Mestrado em Engenharia Informática e Computação

Supervisor: Prof. Rafael Lírio Arrais

Second Supervisor: Pedro Miguel Melo

July 15, 2026

Dedicated Scheduler for Kubernetes in Robotics Applications with ROS/ROS2

David José Soares Castro

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

President: Prof. Tiago Carvalho

Referee: Prof. Agostinho Gil Teixeira Lopes

Supervisor: Prof. Rafael Lírio Arrais

July 15, 2026

Resumo

A crescente complexidade dos sistemas robóticos baseados em Robot Operating System (ROS), em particular os que dependem de cargas computacionais intensivas como perceção e planeamento, tem motivado a transferência de parte do processamento do robô para recursos externos. O Kubernetes afirmou-se como a solução de referência para orquestrar aplicações em *software containers*, mas o seu agendador por omissão foi desenhado para cargas de trabalho genéricas: não considera requisitos de latência, classes de Quality of Service (QoS), proximidade a *hardware* específico, nem os padrões de comunicação próprios dos sistemas ROS. As ferramentas existentes que aproximam o ROS do Kubernetes resolvem a containerização e a orquestração, mas delegam as decisões de colocação no agendador por omissão, deixando por explorar os benefícios de um escalonamento dedicado. Esta dissertação apresenta um agendador Kubernetes dedicado a aplicações robóticas, implementado como um conjunto de dez *plugins* nativos do *Scheduler Framework*, distribuídos por cinco pontos de extensão: ordenação da fila por classe de QoS ROS, filtragem por orçamento de latência, afinidade de *hardware* e co-localização de dados partilhados, pontuação por topologia, latência, utilização de recursos e margem de *deadline*, e reserva de domínios Data Distribution Service (DDS) e de dispositivos de acesso exclusivo. O agendador é configurado de forma declarativa e integra-se no Rigel, um *framework* de containerização de aplicações ROS, cujo *plugin* de orquestração foi também migrado para suportar ROS2 no âmbito deste trabalho. Um gerador automático de perfis infere os requisitos de escalonamento a partir dos metadados que cada pacote ROS já contém. O sistema foi validado num *cluster* Minikube de três nós com uma *suite* de dez aplicações de referência. A validação funcional confirmou que todos os *plugins* produzem as decisões de colocação pretendidas, e a experiência quantitativa de teleoperação mostrou que o agendador dedicado iguala a qualidade de colocação do agendador por omissão sem custo de latência mensurável, garantindo em todas as colocações o orçamento de latência e o *deadline* declarados. O gerador de perfis inferiu corretamente a classe de QoS em três dos cinco cenários e os requisitos de *hardware* em todos. Estes resultados mostram que é possível aproximar a orquestração *cloud-native* dos requisitos das aplicações robóticas sem exigir conhecimentos de Kubernetes aos programadores de robótica.

Palavras-chave: Kubernetes, escalonamento, ROS, ROS2, robótica, orquestração de *containers*, *edge computing*

Abstract

The growing complexity of robotic systems based on Robot Operating System (ROS) — particularly those relying on computationally intensive workloads such as perception and planning — has motivated offloading part of the processing from the robot to external resources. Kubernetes has become the de facto solution for orchestrating containerised applications, but its default scheduler was designed for generic workloads: it does not consider latency requirements, Quality of Service (QoS) classes, proximity to specific hardware, or the communication patterns characteristic of ROS systems. Existing tools that bring ROS to Kubernetes solve containerisation and orchestration, but delegate placement decisions to the default scheduler, leaving the benefits of dedicated scheduling unexplored. This dissertation presents a dedicated Kubernetes scheduler for robotic applications, implemented as a set of ten native Scheduler Framework plugins across five extension points: queue ordering by ROS QoS class, filtering by latency budget, hardware affinity, and shared-data co-location, scoring by topology, latency, resource utilisation, and deadline margin, and reservation of Data Distribution Service (DDS) domains and exclusive-access devices. The scheduler is configured declaratively and integrates with Rigel, a containerisation framework for ROS applications, whose orchestration plugin was also migrated to support ROS2 as part of this work. An automatic profile generator infers scheduling requirements from the metadata every ROS package already contains. The system was validated on a three-node Minikube cluster with a suite of ten reference applications. Functional validation confirmed that every plugin produces the intended placement decisions, and the quantitative teleoperation experiment showed that the custom scheduler matches the placement quality of the default scheduler with no measurable latency overhead, while enforcing the declared latency budget and deadline on every placement. The profile generator correctly inferred the QoS class in three of the five scenarios and the hardware requirements in all of them. These results show that cloud-native orchestration can be brought closer to the requirements of robotic applications without demanding Kubernetes expertise from robotics developers.

Keywords: Kubernetes, scheduling, ROS, ROS2, robotics, container orchestration, edge computing

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework for achieving a better and more sustainable future for all. It includes 17 goals[1] to address the world's most pressing challenges, including poverty, inequality, climate change, environmental degradation, peace, and justice.

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis.

The specific Sustainable Development Goals mentioned have the following names:

SDG 7 Ensure access to affordable, reliable, sustainable and modern energy for all

SDG 8 Promote sustained, inclusive and sustainable economic growth, full and productive employment and decent work for all

SGD	Target	Contribution	Performance Indicators and Metrics
7	7.1	Enhancing the efficiency of SCADA systems can help increase the reliability of solar energy production, facilitating universal access to clean energy.	Percentage of solar plants with improved ...
	7.2	Improving the management of solar plants helps enhance the efficiency and reliability of renewable energy.	Increase in renewable energy share ...
8	8.1	Enhancing renewable energy infrastructure promotes resilience against climate-related hazards and supports sustainable energy sources.	Increase in resilience metrics ...

[1] https://sdgs.un.org/goals

Acknowledgements

I would like to express my sincere gratitude to my supervisors, Prof. Rafael Lírio Arrais and Pedro Miguel Melo, for their guidance, availability, and valuable feedback throughout the development of this dissertation. Their knowledge and continuous support were essential to bringing this work to completion.

I am also deeply grateful to my parents, whose unconditional support and encouragement made this journey possible, and to the rest of my family, for their patience and constant motivation along the way.

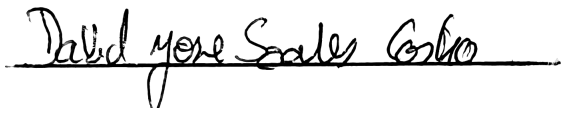
Finally, I would like to thank my friends and colleagues, for their companionship, help, and encouragement throughout these years, which made this path far more rewarding.

David José Soares Castro

Declaration of honour

I declare that this dissertation is my own work and has not previously been submitted or assessed at this or any other institution. References to other authors (statements, ideas, thoughts), as well as the use of published works, strictly comply with the rules of attribution and are duly identified in the text and in the bibliographic references, in accordance with the applicable referencing standards. I am aware that the practice of plagiarism or self-plagiarism constitutes an academic offence, as established in the U.Porto Code of Ethics.
I also declare that I have used Artificial Intelligence tools to help me on the writing of this dissertation, mostly for clarity and grammar. All the results, analysis and conclusions were of my authorship and I'm fully responsible for the content of this dissertation.

David José Soares Castro



"The best way to predict the future is to invent it."

Alan Kay

Contents

List of Figures

List of Tables

List of Acronyms

ACM Association for Computing Machinery
AI Artificial Intelligence
API Application Programming Interface
ARM Advanced RISC Machine
CAN Controller Area Network
CD Continuous Delivery
CI Continuous Integration
CPU Central Processing Unit
DDS Data Distribution Service
DoD Department of Defense
DRS Deep Reinforcement Learning Enhanced Scheduler
EC2 Elastic Compute Cloud
EO Evaluation Objective
FIFO First-In, First-Out
FPGA Field-Programmable Gate Array
GPIO General-Purpose Input/Output
GPU Graphics Processing Unit
GUI Graphical User Interface
HTML HyperText Markup Language
HTTP HyperText Transfer Protocol
HTTPS HyperText Transfer Protocol Secure
ICMP Internet Control Message Protocol
IEEE Institute of Electrical and Electronics Engineers
IP Internet Protocol
JSON JavaScript Object Notation
K3s Lightweight Kubernetes Distribution
KCSS Kubernetes Container Scheduling Strategy
LiDAR Light Detection and Ranging
ML Machine Learning
PVC Persistent Volume Claim

QoS Quality of Service
QS Query String
ROS Robot Operating System
ROS1 Robot Operating System 1
ROS2 Robot Operating System 2
RTT Round-Trip Time
SLAM Simultaneous Localization and Mapping
SoC System on a Chip
TCP Transmission Control Protocol
TTL Time To Live
UI User Interface
UID Unique Identifier
USB Universal Serial Bus
YAML YAML Ain't Markup Language

Chapter 1

Introduction

1.1 Context

Robot Operating System (ROS) is a software framework that enables the development of robotic software. The original design of ROS was crafted to foster collaboration and adaptability across diverse robotic platforms, embracing a peer-to-peer, tools-based, and multi-lingual approach. This foundational philosophy, as described in the 2009 paper [27], has enabled its widespread use but also highlights the need for advanced orchestration as systems scale. However, as ROS systems have increased in complexity, it has become more difficult to achieve minimal downtime during updates. This growing complexity is also associated with the increasing use of containers by ROS developers and the rising computational demands of robotic software, particularly those that rely on Artificial Intelligence (AI)/Machine Learning (ML) workloads.

As a result, it has become necessary to offload part of this computational processing from the robot to external resources. To support the growing complexity of modern robotic systems, especially those requiring real-time performance and multi-robot coordination, the transition to Robot Operating System 2 (ROS2) [19] has been pivotal. This evolution leverages the Data Distribution Service (DDS) to provide enhanced security, embedded support, and robust communication capabilities [19]. In this context, tools such as Kubernetes become essential, as they can be adapted to manage and orchestrate these distributed systems efficiently.

Managing large-scale distributed systems efficiently is a cornerstone of modern cloud technologies, and the principles established by Google's Borg system offer valuable insights. By abstracting resource management and ensuring high reliability across thousands of machines, Borg laid the groundwork for container orchestration tools like Kubernetes [41]. With the advances in these systems, Kubernetes has emerged as a major solution to this type of problem in many different domains, suggesting that similar approaches could also benefit ROS systems.

These tools, while critical for achieving effective orchestration, have undergone numerous improvements in other environments. However, in robotic development, they remain limited, making it difficult for developers, who often resort to manual processes. The U.S. Air Force has extended Kubernetes deployments beyond F-16s to reconnaissance aircraft like the U-2 Dragon

Lady, where the U-2 Federal Laboratory successfully ran Kubernetes during a training flight on legacy systems, demonstrating its viability for disconnected, high-stakes edge environments with real-time data processing needs [40].

In addition, the use of containers would be beneficial for this type of development, as it would allow developers to work with different robotic specifications without encountering environment related errors. In a landmark demonstration of edge computing in aviation, the U.S. Air Force deployed Kubernetes, Istio, Knative, and hardened containers on F-16 fighter jets within 45 days as part of the Department of Defense (DoD) Enterprise DevSecOps Initiative, enabling three concurrent clusters on legacy hardware for disconnected, mission-critical operations [40].

The main focus of this dissertation is the development of a dedicated Kubernetes scheduler that enables developers to configure advanced settings adaptable to robotic software requirements. This scheduler is designed to support configurable scheduling policies, plugins that can be integrated to address specific requirements, and profiles that adapt to diverse use scenarios. With this implementation, the goal is not only to improve the efficiency of robotic software scheduling but also to facilitate and promote the use of Kubernetes in robotic environments, enhancing flexibility and scalability.

In addition to the goals already mentioned, this thesis aims to integrate the proposed scheduler with Rigel [20]. Rigel is a framework developed by INESC TEC that automates the DevOps pipeline of ROS applications: from a single declarative configuration file, it generates Dockerfiles, builds container images, and runs Continuous Integration (CI)/Continuous Delivery (CD) tests for ROS workspaces, following a plugin-based architecture that allows new capabilities to be added as independent modules. One of these modules, the orchestration plugin [9], already offers an interface with Kubernetes for Robot Operating System 1 (ROS1) workloads. This work extends that integration in two directions: it adds ROS2 support to the orchestration plugin and complements it with a dedicated scheduler.

1.2 Motivation

According to recent reports, such as the ROS Metrics Report [24], significant changes have occurred in the evolution of ROS package usage and download patterns for robotic software development. The first key point concerns the growth in the number of downloads in October 2024 compared to October 2023, showing an increase of 1.58%, from 4,647,208 to 4,720,990 downloads. This increase appears reasonable, as ROS usage has consistently grown year-over-year.

Another indicator of this growth is the 23.75% increase in users on the ROS documentation website[1]. Although the number of posts on ROS Discourse[2] has decreased, the number of post views has increased by 18%. This trend is also reflected in the industrial sector, where approximately 1,250 companies are now registered as ROS users. The practical success of ROS2 is evident in its deployment across various domains, from warehouse automation to space exploration, with

[1] https://docs.ros.org
[2] https://discourse.openrobotics.org

significant cost savings and efficiency gains, such as the $1M-$5M saved by OTTO Motors [19], underscoring the demand for robust orchestration solutions.

However, when examining the total annual download values, a significant overall decrease of 3.36% becomes evident, with total downloads dropping to 531,452,142 in 2024. These statistics suggest that this decrease may result from increased container usage in robotic software development.

This situation also reveals that the use of Kubernetes and containers in robotic environments is constrained by a limited range of solutions that facilitate such implementations. In robotics, Kubernetes at the edge provides flexibility for real-time control in manufacturing and automotive industries, merging cloud-native orchestration with embedded systems to manage robot fleets with optimised scheduling for latency-sensitive tasks [35]. The evolution from ROS [23] to ROS2 [30] has also introduced more sophisticated communication patterns and distributed architectures, making orchestration increasingly critical in robotic systems.

The shift from ROS1 to ROS2 introduces critical improvements, including a standardised network transport with DDS, peer-to-peer discovery, and commercial-grade embedded support, addressing previous limitations [19]. These advancements drive the need for a scheduler tailored to these enhanced capabilities. Additionally, robotic applications tend to demand real-time or near, requiring Kubernetes schedulers to meet specific requirements such as low-latency constraints, Quality of Service (QoS) guarantees, and complex task dependencies. Currently, there is a notable gap in DevOps practices for robotics, particularly regarding automated orchestration solutions. To enhance code reusability and reduce dependency issues, ROS promotes the development of standalone libraries independent of its core framework [27]. This modular approach supports the adoption of container technologies and inspires the design of a flexible, plugin-based scheduler.

Developing software that manages scheduling and is easily configurable can provide significant value to the community by facilitating the use of Kubernetes in robotic development and enabling improved orchestration and resource management in robotic environments. Given these constraints, the main motivation is not only to develop a more accessible solution that addresses these requirements but also to advance the integration of such solutions into robotic environments, fostering new developments in this area. Another motivation is to encourage broader adoption of these technologies within the field by ensuring that the proposed tools are easier to use and adaptable to different situations. In e-commerce platforms like eBay and Shopify, Kubernetes is used to automatically scale applications during peak shopping seasons, handling variable traffic loads by rescheduling containers to optimise resource utilisation and ensure high availability without downtime [33]. The same elasticity is directly applicable to robotics: a fleet of warehouse robots facing a demand peak could have its perception and planning workloads automatically rescheduled across edge and cloud nodes, maintaining responsiveness without manual intervention, precisely the kind of capability that this dissertation aims to bring within reach of ROS developers.

1.3 Problem

Currently, the tools that allow ROS systems to run on Kubernetes in robotic environments are highly limited, presenting significant constraints to the advancement of this technology and making it difficult for developers to adopt Kubernetes in their robotic systems. This highlights the need for robust orchestration, yet robotic applications face unique challenges. Healthcare organisations leverage Kubernetes to run AI/ML workloads for predictive analytics and patient data processing, providing scalability for containerised applications in secure, compliant environments, as seen in systems managing real-time diagnostics and telemedicine services [7]. Robotic systems share these demands for scalable, dependable processing of streaming data, but add placement constraints, physical hardware access and strict latency bounds, that such deployments do not face.

Another critical problem that emerges is achieving low latency and ensuring QoS, both of which are essential requirements for robotic tasks. Handling diverse workloads, from latency-sensitive services to batch processes, is a key challenge in large-scale systems, a principle well-demonstrated by Borg's management of heterogeneous tasks. This insight, from the 2015 paper [41], highlights the limitations of standard schedulers in meeting robotic application needs. The few existing technologies that handle real-time tasks are not only nearly non-existent but also highly limited in these scenarios, exhibiting high latency that results in significant challenges for developers.

The distributed nature of ROS, relying on a peer-to-peer topology to connect processes across multiple hosts, eliminates central server bottlenecks and supports flexible communication [27]. This design necessitates a scheduler that can adapt to such dynamic patterns in robotic environments. The standard Kubernetes scheduler was not designed with the specific needs of robotic applications in mind, such as complex task dependencies, real-time constraints, and the communication patterns inherent to ROS2 based systems.

Addressing this problem requires a solution that enables robotics teams to interact more efficiently with Kubernetes while providing low latency and the flexibility to schedule workloads according to specific requirements. One of the primary constraints that must be addressed from the outset is achieving sufficiently low latency, given the demanding nature of robotic environments. Optimising resource utilisation through fine-grained control and advanced techniques like task-packing and over-commitment is essential for high-performance systems [41]. These strategies provide a model for designing a scheduler that addresses the specific latency and QoS requirements of robotic tasks. Additionally, the scheduler must support configurable policies and plugins to adapt to different robotic scenarios and requirements.

In summary, the main challenge of this thesis lies in developing a dedicated scheduler that assists robotics developers in implementing Kubernetes while ensuring low latency and high QoS, thereby bridging the gap between cloud native orchestration practices and the specific requirements of robotic applications. A custom Kubernetes scheduler enables cloud bursting by dynamically rescheduling workloads to public clouds during high demand, as used in e-commerce for handling traffic spikes while maintaining cost efficiency and performance isolation [4]; transposed to robotics, the same mechanism would let a robot offload computationally heavy perception or planning tasks

to external resources whenever its onboard capacity is exhausted.

1.4 Research Questions

The growing use of ROS in robotic software development increases the complexity of these systems. The need for a real-time Kubernetes scheduling system with low latency and high QoS is a crucial constraint to consider during the development of such a system.

Another crucial aspect to consider is the diversity of system designs within ROS. The scheduler must provide a level of customisation that allows users to adapt it to their specific use cases. Finally, for the scheduler to reach its intended audience, it cannot remain a standalone infrastructure component: it must be integrable into the containerisation and deployment tools that robotics developers already use, of which Rigel is a representative example.

These considerations lead to three main research questions:

1. **RQ1:** How can a Kubernetes scheduler be implemented to achieve low latency and high QoS, as required by ROS based systems?

2. **RQ2:** Which methods should remain flexible and configurable by the user, and how should the user interact with this environment?

3. **RQ3:** How can the proposed scheduling approach be reliably integrated into a containerisation framework for robotic applications with Kubernetes support, so that custom scheduling becomes part of existing ROS development workflows rather than a separate infrastructure concern?

Chapter 2

State of the Art

This chapter reviews the state of the art on container orchestration for robotic systems, with a focus on Kubernetes scheduling for ROS environments. It first describes the literature review process that supports the analysis like sources, queries, and filtering criteria and then examines the retrieved body of work: default scheduler limitations, custom scheduling strategies, scheduler extensibility mechanisms, integration with ROS ecosystems, and configuration usability. The goal is to identify why existing scheduling mechanisms fall short in robotic contexts and which gaps a dedicated, configurable Kubernetes scheduler must close. From the collected literature, three main pillars emerge:

1. Kubernetes scheduling fundamentals and extensibility
2. Real-time and edge orchestration in robotics
3. Integration patterns between ROS ecosystems and cloud-native tools

The analysis is guided by the same three research questions introduced in Section 1.4 (RQ1–RQ3).

2.1 Literature Review Methodology

This work adopts an explanatory review of the literature to construct a strong narrative around the relationship between Kubernetes and robotic systems. Unlike systematic reviews that prioritise exhaustive inclusion, this approach emphasises conceptual progression: from the foundational principles of container orchestration, to real-time scheduling challenges, and finally to emerging ROS-native integrations. Nevertheless, to ensure rigour and reproducibility, the identification and selection of publications followed the core practices of systematic literature reviews in software engineering [13, 25], as applied to the ROS domain by the recent systematic review of Cardoso et al. [1]. From that review's PRISMA-aligned procedure, this work adopts its stepwise structure, formulation of the research questions, development of search queries and selection of sources, screening of retrieved records against predefined inclusion and exclusion criteria, full-text assessment of eligible studies, and synthesis of the extracted evidence, together with its two-stage

screening model (title-and-abstract screening followed by full-text review) and the practice of documenting record counts at every stage so that the selection process can be verified by third parties.

The review is based on three complementary sources. Institute of Electrical and Electronics Engineers (IEEE) Xplore and the Association for Computing Machinery (ACM) Digital Library were used to retrieve peer-reviewed, high-impact studies on scheduling algorithms and real-time systems, while Google Scholar was used to capture preprints, technical reports, and open-source project documentation (e.g., ROS2 working groups) that reflect the latest practical advancements. This combination ensures both scientific rigour and recency, critical given the rapid evolution of ROS2 and Kubernetes (post-2020).

2.1.1 Query Formulation

The search was guided by three progressively specialised query strings (QS), numbered QS1–QS3:

- **QS1**: `("kubernetes scheduler" OR "custom scheduler" OR "scheduler plugin") AND ("ROS" OR "ROS2" OR "robotic operating system" OR "robotics")`
  - *Focus*: Direct intersections of Kubernetes scheduling and ROS environments.
- **QS2**: `("real-time scheduling" OR "low latency" OR "QoS" OR "deadline") AND ("kubernetes" OR "container orchestration") AND ("ROS" OR "robot")`
  - *Focus*: Real-time constraints in containerised robotic systems.
- **QS3**: `("kubernetes" OR "k8s") AND ("ROS" OR "robotics") AND ("devops")`
  - *Focus*: DevOps and automation pipelines linking ROS workspaces to Kubernetes.

Each query targets one of the review pillars. QS1 captures the sparse but growing body of work on Kubernetes in ROS contexts, including early prototypes and scheduler extensions. QS2 targets the critical tension between Kubernetes' best-effort model and robotics' hard real-time needs, revealing gaps in latency and QoS. QS3 identifies automation tools that lower the barrier to Kubernetes adoption in robotics, justifying the need for scheduler-level configurability.

2.1.2 Identification and Filtering Processes

Initial searches returned 2275 records across all queries (Table 2.1). Records were imported into Zotero[1] for deduplication and metadata tagging.

[1] https://www.zotero.org

Table 2.1: Distribution of Retrieved Records

Database	QS1	QS2	QS3
ACM Digital Library	371	293	384
Google Scholar	476	521	219
IEEE Xplore	3	1	7

After deduplication, a two-stage screening procedure was applied, consisting of title-and-abstract screening followed by full-text review [1]. A record was *included* when its title and abstract evidenced at least one of the following:

- Discussion of custom Kubernetes schedulers or scheduler plugins.
- Real-time or edge robotics use cases involving container orchestration.
- ROS/Kubernetes integration, including DevOps pipelines for robotic workspaces.

Records were *excluded* when they (i) addressed container orchestration platforms other than Kubernetes without transferable scheduling insights, (ii) targeted generic cloud workloads with no latency, QoS, or hardware-locality concerns, or (iii) were not available in English or lacked an accessible full text. Records that passed the title-and-abstract screening were then read in full and assessed for their relevance to the three research questions, with each document examined multiple times to establish connections between this research and previous work.

Through this process, from the 2275 records initially returned, 55 publications were retained. Table 2.2 groups the principal retained publications, those analysed in depth in the remainder of this chapter, by review pillar.

Table 2.2: Principal retained publications, grouped by review pillar.

Pillar	Publications
Kubernetes scheduling fundamentals and extensibility	[3, 8, 10, 11, 15, 21, 22, 28, 31, 32, 34, 41]
Real-time and edge orchestration in robotics	[2, 5, 9, 14, 16–18, 29, 42–45]
Integration of ROS ecosystems with cloud-native tools	[6, 12, 19, 20, 27, 40]

2.2 Kubernetes Scheduling Fundamentals

2.2.1 Default Scheduler Limitations for Robotics

The default Kubernetes scheduler employs a two phase approach: filtering nodes that meet pod requirements, then scoring remaining nodes to select the optimal placement [28]. In the scoring phase, the scheduler evaluates each feasible node using a weighted combination of priority functions that assess factors such as resource availability, node affinity rules, and load balancing objectives.

Each scoring plugin assigns a numerical score to candidate nodes, and these scores are normalised and aggregated according to configured weights to determine the final node selection. This scoring mechanism, while effective for optimising resource distribution in general purpose clusters, prioritises metrics like even resource spread and preventing resource fragmentation rather than minimising communication latency or ensuring deterministic timing behaviour. While effective for web applications prioritising resource efficiency, this model presents significant limitations for robotic workloads.

Absence of Real Time Awareness: The default scheduler operates on a best effort basis without guarantees for deterministic execution or bounded latency. Robotic control loops requiring sub 100ms response times cannot rely on scheduling decisions that may introduce variable delays. The scoring algorithms prioritise resource balance across nodes rather than minimising communication latency between interdependent components.

Generic Resource Metrics: Standard scheduling considers Central Processing Unit (CPU), memory, and ephemeral storage but ignores robotics specific requirements such as proximity to ROS Master nodes, network topology constraints, or hardware accelerator availability (Graphics Processing Units (GPUs) for perception, Field-Programmable Gate Arrays (FPGAs) for sensor processing). In certain robotic applications, direct hardware access or physical proximity to specific devices (such as sensor interfaces, motor controllers, or specialised communication buses) becomes critical, requiring the scheduler to account for hardware locality constraints that standard Kubernetes metrics do not capture. The scheduler cannot make informed decisions about co-locating publisher subscriber pairs to reduce message latency.

Static Policy Configuration: Modifying scheduler behaviour requires recompiling Kubernetes components or deploying webhook based extenders that introduce additional network hops. There is no declarative interface for robotics developers to specify application specific scheduling preferences without deep Kubernetes expertise.

2.2.2 Scheduling in Distributed Robotic Systems

Several works have explored scheduling challenges when ROS applications span edge, fog, and cloud infrastructure.

Resource Management in Fog Robotics [43]: Xu et al. demonstrated Kubernetes deployments for industrial fog robots, comparing local Lightweight Kubernetes Distribution (K3s) clusters with cloud Elastic Compute Cloud (EC2) instances. Their evaluation revealed that distributed edge computing achieves superior resource utilisation for certain workloads. However, relying on default Kubernetes scheduling without robotic specific optimisations left network latency between fog nodes and robots unaddressed. The study highlighted the need for topology aware placement decisions.

Comprehensive Modeling Approach [44]: Zhang et al. proposed a scheduling strategy for allocating distributed multi robot software across edge and cloud environments. Their mathematical formulation optimised for makespan and energy consumption but required extensive manual system modeling. The approach lacks integration with existing ROS development frameworks, limiting

practical adoption. The complexity of specifying workload dependencies and QoS constraints manually demonstrates the need for declarative configuration abstractions.

Taken together, these two studies expose complementary halves of the same problem: Xu et al. show that off-the-shelf Kubernetes scheduling wastes the latency advantages that fog and edge infrastructure could offer, while Zhang et al. show that placement strategies sophisticated enough to exploit that infrastructure are too costly to specify by hand. A practical solution for robotics must therefore combine topology-aware placement logic with an automated, low-effort way of expressing workload requirements, the design tension that the custom scheduling strategies reviewed next attempt to resolve.

2.3 Custom Kubernetes Scheduling Strategies

2.3.1 Scheduler Framework and Extensibility Mechanisms

Kubernetes introduced the Scheduler Framework in v1.19 [39][2], enabling custom scheduling logic through plugins without forking the core scheduler [28, 32]. This architecture supports multiple extension points:

- **Queue Sort**: Prioritise pods in the scheduling queue based on custom criteria
- **Filter**: Eliminate unsuitable nodes using domain specific constraints
- **Score**: Rank feasible nodes according to optimisation objectives
- **Reserve/Bind**: Handle resource reservation and pod to node binding
- **PreBind/PostBind**: Execute actions before and after binding

The plugin architecture allows implementing ROS specific scheduling logic (preferring nodes with low network latency to ROS Master, avoiding nodes with incompatible hardware drivers) without modifying Kubernetes internals. However, no existing work has developed a comprehensive plugin suite for robotic workloads. While the framework enables domain specific customisation, adoption in robotics remains minimal: extending it requires proficiency in Go, the language in which Kubernetes and its Scheduler Framework are written, and there is little documented best practice for robotic scenarios [32].

2.3.2 Resource Aware Scheduling Algorithms

KCSS: Kubernetes Container Scheduling Strategy [21]: Menouer introduced resource aware scheduling considering CPU, memory, and network bandwidth. The algorithm improved resource utilisation compared to default Kubernetes scheduling in cloud environments. However, KCSS does not account for real time constraints, hardware proximity requirements, or ROS communication

[2] https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/

patterns such as publish subscribe topics, services, and actions. The optimisation prioritises resource efficiency over latency minimisation, making it unsuitable for time sensitive robotic tasks where predictable response times are critical.

Context Aware Scheduler for Edge Applications [22]: Ogbuachi et al. developed a scheduler considering network latency, device capabilities, and application context for 5G edge environments. The context aware approach aligns with robotic requirements for location sensitive scheduling decisions. However, the implementation does not expose configurable policies to developers, embedding scheduling logic directly in code. The lack of explicit support for ROS communication patterns and DDS networking limits applicability to robotic systems requiring complex inter node coordination.

These two efforts mark an interesting progression: KCSS broadens the set of resources the scheduler reasons about, and Ogbuachi et al.'s work goes further by bringing network context into the decision. Yet both retain the assumption that scheduling policy is fixed at implementation time. For robotics, where requirements vary widely between applications and even between nodes of the same application, this assumption is the binding constraint, which motivated the exploration of learned policies as a way to adapt automatically.

2.3.3 Machine Learning Based Scheduling

DRS: Deep Reinforcement Learning Enhanced Scheduler [11]: Jian et al. proposed using machine learning to optimise scheduling for microservice based systems. Their learned policies adapted better to dynamic workloads than static heuristic approaches in web service scenarios. Despite promising results, DRS does not address deterministic timing requirements of robotic control loops. The computational overhead of neural network inference during scheduling decisions could introduce unacceptable latency for real time operations. Additionally, the approach requires extensive training data collected from production workloads, limiting applicability to novel robotic applications without historical scheduling traces.

Machine Learning Based Cluster Scheduling [8]: Harichane et al. explored using ML to predict optimal scheduling for heterogeneous clusters, showing promise for workloads with predictable patterns such as repetitive robotic tasks. However, training requires substantial historical data, and the scheduler lacks mechanisms for developers to specify QoS requirements or deadline constraints explicitly. The black box nature of learned policies makes debugging scheduling decisions difficult in safety critical robotic deployments.

While learned policies remove the need to hand-tune heuristics, both works trade away two properties that robotics cannot give up: explainability of individual placement decisions, and the ability for the developer to state a hard requirement (a deadline, a hardware dependency) that the scheduler must honor rather than statistically prefer. This suggests that for robotic workloads the adaptation problem is better solved at the configuration layer letting users declare requirements than at the policy-learning layer, a direction explored by the QoS aware frameworks below.

2.3.4 QoS Aware Scheduling Frameworks

Gaia Scheduler Framework [34]: Song et al. introduced a Kubernetes based system for diverse workload types, demonstrating that custom plugins can accommodate specialised requirements. The framework provides architectural patterns for extending scheduler behaviour. While offering valuable insights, Gaia does not specifically target real time constraints or ROS communication patterns. The plugin mechanism could theoretically support robotic workloads, but no such implementation exists, and the framework lacks guidance on translating robotic QoS requirements into scheduling policies.

The tension between Kubernetes' best effort scheduling model and robotics' real time needs has been partially addressed through these research efforts, yet significant gaps remain. Existing solutions either lack ROS specific optimisations, require substantial machine learning infrastructure, or do not provide developer friendly configuration interfaces for specifying application requirements declaratively.

2.4 Integration of Scheduling with ROS Ecosystems

2.4.1 Container Based Approaches for ROS Applications

Several works have explored integrating ROS with Kubernetes, though most focus on orchestration rather than scheduling optimisation.

Container Based Design for ROS Applications [16, 17]: Lumpp et al. presented a design methodology for deploying ROS applications on Kubernetes edge cloud architectures. Their approach successfully demonstrated mixed criticality applications with separated control loops and computational workloads. The solution relies on manual pod configuration and default Kubernetes scheduling, lacking automated placement optimisation based on robotic workload characteristics. The networking abstractions, while valuable for multi host ROS Master communication, demand significant expertise and do not address scheduling decisions that could minimise communication overhead.

RobotKube Framework [14]: Lampe et al. developed orchestration capabilities for large scale multi robot systems, demonstrating centralised management of robot fleets through Kubernetes. Their deployment on TurtleBot3 platforms proved practical viability for multi robot coordination. However, RobotKube focuses on fleet management rather than fine grained scheduling optimisation. The system does not address low latency requirements through custom scheduling policies or provide configurable mechanisms for diverse industrial application scenarios. Scheduling decisions remain delegated to Kubernetes defaults.

Both lines of work prove that ROS applications can run on Kubernetes at realistic scale, but both also stop at the same boundary: once the containers exist, placement is handed back to the default scheduler. In other words, the orchestration problem has been solved ahead of the scheduling problem, leaving the latency and hardware-locality benefits of custom placement untapped precisely the gap this dissertation targets.

2.4.2 Frameworks for ROS Containerisation

The Rigel framework represents the current state of the art in automated ROS containerisation. Rigel automates the generation of Dockerfiles, container building, and CI/CD testing for ROS workspaces [20]. Rigel itself provides no built-in Kubernetes support; its plugin architecture, however, allows such capabilities to be added as separate modules. Building on these extension points, a dedicated orchestration plugin [9] manages deployment, rolling updates, and observability for containerised ROS applications. At the start of this work, however, that orchestration plugin supported ROS1 workloads only; extending it to ROS2, whose DDS-based discovery removes the ROS Master and changes how inter-node communication is configured, is one of the contributions of this dissertation, described in Chapter 3.

Moreover, Rigel's integration with Kubernetes relied exclusively on the default scheduler. While the orchestration plugin successfully handles container lifecycle management, it did not provide mechanisms for developers to specify custom scheduling strategies tailored to robotic workload requirements. There was no declarative interface for expressing scheduling preferences such as latency sensitive pod placement, QoS guarantees, or ROS topology aware node selection.

Extending Rigel to support invocation of custom scheduling strategies completes the DevOps pipeline for robotics, allowing developers to specify not only how containers are built and deployed, but also how they are scheduled across cluster resources. This integration enables robotics teams to leverage advanced scheduling techniques without requiring expertise in Kubernetes internals or Go programming.

2.5 Scheduler Configuration and Usability

2.5.1 Declarative Configuration Paradigms

Current custom scheduler implementations typically require developers to modify source code or compile custom binaries to adjust scheduling behaviour. This approach creates barriers for robotics developers who lack infrastructure engineering expertise.

Comparison of Configuration Approaches: Kubernetes supports several methods for customising scheduler behaviour, each with distinct trade offs. The Scheduler Extender mechanism allows external processes implemented in any programming language to influence scheduling decisions through HyperText Transfer Protocol (HTTP) callbacks, providing flexibility in implementation choice but introducing network latency of tens of milliseconds per scheduling decision that proves unacceptable for latency sensitive robotic workloads; additionally, this extender mechanism has been deprecated in favour of native plugins. The Scheduler Framework Plugins approach [28] offers native integration with low overhead and access to all scheduling phases, though it requires Go programming expertise and scheduler binary recompilation to modify behaviour, making it optimal for performance but demanding significant Kubernetes development knowledge; this approach demonstrates high applicability for robotic use cases if appropriate abstraction layers can hide implementation complexity from end users. Finally, implementing a Custom Scheduler as a

separate binary provides complete control with the possibility of using any programming language, but requires reimplementing core scheduling logic and maintaining compatibility with Kubernetes Application Programming Interface (API) changes, resulting in a high maintenance burden despite offering maximum flexibility for domain specific optimisations.

For robotic applications requiring low latency scheduling with developer friendly configuration, an approach combining Scheduler Framework plugins with declarative policy specification offers the best balance. Implementing core scheduling logic as plugins ensures performance, while abstracting configuration through YAML Ain't Markup Language (YAML) schemas or similar formats makes the solution accessible to robotics developers [32].

2.5.2 User Interaction and Developer Experience

The success of scheduling solutions in robotics depends not only on technical performance but also on usability. Complex configuration interfaces or requirements for specialised knowledge limit adoption in robotics teams that prioritise application development over infrastructure management.

Existing ROS tools emphasise declarative configuration through launch files and parameter servers. A custom scheduler targeting ROS ecosystems should adopt similar paradigms, allowing developers to specify scheduling preferences using familiar YAML based formats. Configuration parameters might include:

- Latency constraints for specific node types (e.g., control loops require <50ms)
- Co location preferences for publisher subscriber pairs
- Hardware affinity requirements (GPU, specific sensor interfaces)
- QoS profiles mapping to Kubernetes resource guarantees
- Fallback policies when ideal scheduling cannot be achieved

No existing scheduler provides such a declarative interface tailored to robotic application patterns, representing a significant gap in usability.

2.6 Critical Discussion and Research Gaps

Lack of ROS Specific Scheduling Optimisation: Existing Kubernetes schedulers optimise for generic cloud metrics such as CPU utilisation, memory pressure, and network bandwidth. None explicitly consider ROS communication patterns to minimise latency between publisher subscriber pairs, co-locate Service clients with servers, or ensure Action servers meet deadline constraints. Manual configuration can approximate these optimisations [17, 44] but imposes significant cognitive load on developers and lacks systematic approaches for complex applications.

Absence of Configurable Scheduling Policies: Research demonstrates that custom schedulers can enforce latency constraints and QoS guarantees [8, 22, 34]. However, implementations hardcode

policies within scheduler logic or require recompilation to modify behaviour. Robotic applications exhibit diverse QoS requirements across scenarios. Teleoperation demands sub 100ms latency while Simultaneous Localization and Mapping (SLAM) algorithms tolerate higher latency with guaranteed throughput. No existing scheduler provides a declarative interface for specifying such requirements in machine readable format that translates to scheduling decisions at runtime.

Insufficient Integration with ROS Development Workflows: Current scheduler implementations require Kubernetes expertise that most robotics developers lack. Solutions like RobotKube [14] and Lumpp et al.'s design flow [17] demonstrate Kubernetes viability for robotics but require manual YAML manifest creation and deep understanding of Kubernetes concepts. The Rigel framework shows that abstraction layers can hide complexity for containerisation and orchestration, but no scheduler follows this paradigm. The gap between accessible containerisation tools and complex scheduling mechanisms limits Kubernetes adoption in robotics.

Limited Validation in Robotic Contexts: Most custom schedulers [11, 21] evaluate using web service benchmarks or synthetic workloads. The few robotics focused studies [14, 17, 44] validate on specific platforms (TurtleBot3, industrial manipulators) without generalising findings to the broader ROS ecosystem. The absence of standardised benchmarks for robotic scheduling hinders comparative evaluation and makes assessing scheduler performance across different robotic applications difficult. Metrics such as scheduling decision latency, message delivery jitter, and task deadline satisfaction rates are rarely reported in ways that enable cross study comparison.

In summary, the literature reveals a consistent pattern: the building blocks for robotic-aware scheduling exist, an extensible Scheduler Framework, demonstrated ROS-on-Kubernetes deployments, and accessible containerisation tooling, but no work assembles them into a configurable, ROS aware scheduler with declarative policy specification integrated into the tools robotics developers already use. Closing this gap is the objective of this dissertation: a dedicated Kubernetes scheduler, invocable through Rigel, that balances developer friendly configuration with extensibility for advanced use cases, and that is validated using robotics specific metrics. The next chapter presents the contributions of this work in detail and describes how each identified gap is addressed by the implemented system.

Chapter 3

Approach

This chapter presents the solution developed in this dissertation: a dedicated Kubernetes scheduler tailored to robotic applications using ROS, together with its integration into the Rigel framework [20]. Building upon the gaps identified in Chapter 2, the work produced two interrelated software components: the **ros-aware-scheduler**, a custom out-of-tree Kubernetes scheduler, and an extension to the **rigel-orchestration-plugin** that allows developers to specify scheduling preferences declaratively in a familiar YAML configuration file. Both components were developed and validated on a local Kubernetes cluster using Minikube [38].

Concretely, this work advances the state of the art through the following contributions, each addressing one of the gaps identified in Chapter 2:

- **ROS2 support in Rigel**: the Rigel orchestration plugin, which previously supported only ROS1, was migrated to also orchestrate ROS2 workloads, handling DDS-based discovery, ROS2 environment configuration, and the `colcon` build toolchain (Section 3.3). This is a standalone contribution to the Rigel ecosystem that also serves as the foundation for the scheduler integration.
- **A ROS-aware scheduler**: ten scheduling plugins implemented across five Kubernetes Scheduler Framework extension points, encoding ROS communication semantics, QoS classes, latency budgets, deadlines, topology preferences, hardware locality, shared-data co-location, and DDS domain management, as native, low-overhead scheduling logic (Section 3.4).
- **A declarative configuration interface**: a self-contained pod annotation schema, a `scheduler` block in the Rigelfile that maps one-to-one to that schema, and a lightweight web interface, allowing scheduling behaviour to be expressed in domain terms (latency, QoS, hardware) without Kubernetes expertise (Sections 3.5 and 3.6).
- **An automatic profile generator**: a tool that infers scheduling requirements from the metadata each ROS package already contains, providing a meaningful default for developers who do not author any scheduling configuration at all (Section 3.6.2).

- **A validation methodology for robotic scheduling**: a suite of ten reference applications covering the principal robotic communication and hardware placement scenarios, together with robotics-specific evaluation metrics, used in Chapter 4 to assess the system.

The remainder of the chapter first states the design principles and overall architecture (Sections 3.1 and 3.2), then describes each component in turn: the ROS2 migration (Section 3.3), the scheduler implementation (Section 3.4), the configuration interface (Section 3.5), and the Rigel integration with automatic profile generation (Section 3.6).

3.1 Design Principles

The scheduler follows three core design principles that distinguish it from the existing solutions reviewed in Chapter 2:

ROS-Awareness: Unlike generic Kubernetes schedulers that optimise for web service metrics, this scheduler explicitly considers ROS communication patterns. This includes understanding publisher-subscriber relationships, service call dependencies, and action server locations to make informed placement decisions that minimise communication latency and maximise QoS compliance. Additionally, the scheduler recognises the unique characteristics of robotic systems: certain nodes must execute directly on the robot (such as hardware drivers for sensors and actuators), while others should be offloaded to external servers (such as computationally intensive ML/AI processes) to reduce the computational burden on battery-powered robotic platforms.

Declarative Configuration: Rather than requiring developers to modify Go source code or recompile binaries, the scheduler provides a YAML-based configuration interface familiar to ROS developers. Users specify scheduling preferences, QoS requirements, and constraints using declarative syntax that abstracts the underlying Kubernetes complexity. To further enhance usability, a web-based Graphical User Interface (GUI) complements the YAML interface, allowing scheduling parameters to be adjusted through point-and-click interactions that regenerate the underlying configuration file.

Extensibility through Plugins: The architecture supports custom scheduling algorithms through a plugin system that allows robotics researchers and practitioners to integrate specialised heuristics for specific applications (e.g., swarm coordination, multi-robot SLAM, teleoperation) without forking the core scheduler implementation. This design draws inspiration from the Rigel framework [20], which itself employs a plugin-based architecture, enabling modular and extensible functionality.

3.2 System Architecture

The implemented system is composed of two cooperating components connected by a shared annotation schema injected into Kubernetes pod specifications. Figure 3.1 illustrates how the components interact.

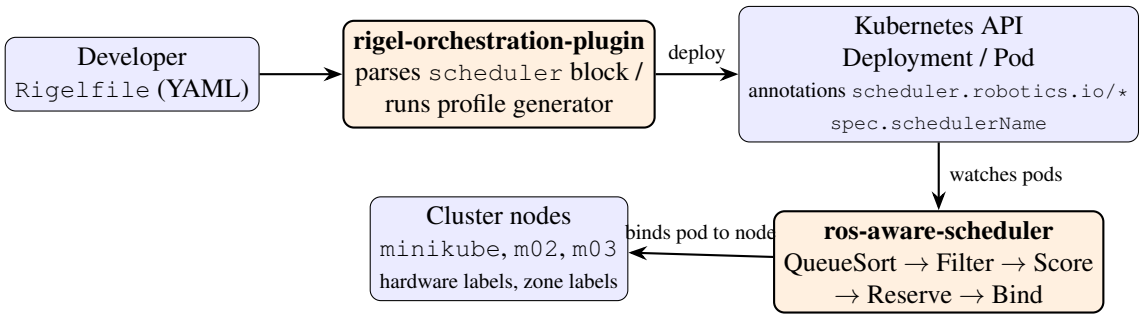


Figure 3.1: Architecture overview: the Rigel orchestration plugin translates the Rigelfile scheduling declaration into pod annotations, and the ros-aware-scheduler consumes those annotations to decide pod placement through the Scheduler Framework extension points.

The **rigel-orchestration-plugin** acts as the user-facing entry point. When deploying a ROS application, the developer declares scheduling requirements in a `Rigelfile` YAML configuration file. The plugin translates these requirements into Kubernetes pod annotations using the `scheduler.robotics.io/*` namespace and sets `spec.schedulerName` to route the pod to the custom scheduler binary. To make this entry point concrete, Listing 3.1 shows the essence of such a declaration: a short block in which the developer states the workload requirements in domain terms (a latency budget, a QoS class, a preferred zone) and leaves the translation into annotations to the plugin. The complete schema and a fully worked example are presented later in Section 3.6; the snippet here is meant only to ground the data flow described in Figure 3.1.

```
orchestration:
  scheduler:
    name: kubernetes-scheduler
    policies:
      latency_budget: "30ms"
      qos_class: "realtime"
    topology:
      preferred_zones:
        - "edge-zone-1"
```

Listing 3.1: Minimal Rigelfile scheduling declaration that the orchestration plugin translates into the pod annotations consumed by the scheduler.

The **ros-aware-scheduler** is the custom Kubernetes scheduler binary. It runs as a second scheduler alongside the default `default-scheduler` in the control plane. Pods that name the custom scheduler in their `spec.schedulerName` field are exclusively processed by it; all remaining workloads continue to use the default scheduler without interference. The scheduler reads the annotations injected by the Rigel plugin and applies a pipeline of ROS-aware decision logic through the Kubernetes Scheduler Framework extension points.

Conceptually, the system comprises four layers distributed across these two components. The

configuration layer (in the Rigel plugin and the web interface) parses and validates the user-facing YAML declarations and translates them into the annotation schema. The *scheduling core* (the scheduler binary) implements the main placement logic through the Scheduler Framework. The *plugin system* hosts the individual scheduling algorithms behind well-defined interfaces, so new heuristics can be added without touching the core. Finally, the *integration layer* (the orchestration plugin's deployment logic and the scheduler's Prometheus [26] endpoint, Prometheus being an open-source monitoring and time-series database) connects the system to external tooling, Rigel workflows on one side, observability infrastructure on the other.

This design separates concerns cleanly: Rigel handles the DevOps pipeline (building, containerising, and deploying ROS workspaces), while the custom scheduler handles placement optimisation. Neither component depends on the other at runtime, which means the scheduler can also be used without Rigel by manually annotating pod specifications.

Regarding technology selection, the scheduler was implemented in Go [36], the native language of Kubernetes. This choice ensures minimal overhead when interfacing with Kubernetes APIs, allows direct use of the Scheduler Framework without additional translation layers, and provides access to the extensive Go ecosystem for container orchestration. While alternative languages could be used through the scheduler extender mechanism, that webhook-based approach introduces tens of milliseconds of network latency per decision, unacceptable for real-time robotic workloads, and has been deprecated in favour of native plugins, as discussed in Chapter 2. The implementation builds against Kubernetes v1.31 [37]. This was not the most recent Kubernetes release at the time of writing; it was chosen because it was the newest version with a stable, well-documented vendoring path for the `k8s.io/kubernetes` scheduler packages when development started, and it remained within the officially supported version skew window[1] throughout the project. Since the Scheduler Framework API used by the plugins has been stable since v1.19 [39], the scheduler also compiles against later releases without changes.

3.3 ROS2 Migration of the Rigel Orchestration Plugin

The Rigel orchestration plugin previously supported ROS1 workloads only. A prerequisite for the scheduler integration was extending the plugin to support ROS2, as the new communication stack based on DDS introduces fundamental differences in how nodes are discovered and how inter-node communication is configured.

The migration involved several concrete changes. The plugin now infers the ROS major version from the `application.distro` field in the Rigelfile, comparing it against the known ROS1 and ROS2 distribution lists maintained by Rigel. When a ROS2 distribution is detected, the plugin suppresses ROS Master deployment (since ROS2 uses DDS peer-to-peer discovery instead of `roscore`) and generates the environment variables that DDS-based communication relies on: `ROS_DOMAIN_ID`, which partitions the network so that only nodes sharing the same domain identifier discover one another, and `RMW_IMPLEMENTATION`, which selects the underlying

[1] https://kubernetes.io/releases/version-skew-policy/

DDS middleware vendor. These replace the `ROS_MASTER_URI` and `ROS_IP` variables required by ROS1, which respectively tell each node where to reach the central ROS Master and which address to advertise for incoming connections.

Container image generation was also updated to handle the different workspace build tools: ROS1 workspaces are typically built with `catkin_make` or `catkin build`, while ROS2 workspaces use `colcon`. The plugin selects the conventional tool for each distribution, although recent ROS1 releases such as Noetic can also be built with `colcon`. A readiness probe mechanism was implemented to allow the orchestration plugin to verify that the ROS application inside the container has fully initialised before considering the pod ready for traffic.

The migrated plugin maintains full backward compatibility with existing ROS1 Rigelfiles, which require no changes. The `deploy_ros_master: true` field continues to work exactly as before for ROS1 workloads. A warning is emitted when `deploy_ros_master: true` is set for a ROS2 distribution, since roscore is not needed and its presence may cause confusion. Listing 3.2 contrasts the orchestration block of a ROS1 and a ROS2 Rigelfile: the only fields that change are the declared `distro` and the `deploy_ros_master` flag.

```
# ROS1 (unchanged): roscore is deployed as before
application:
  distro: noetic
orchestration:
  deploy_ros_master: true

# ROS2: roscore is suppressed, DDS discovery is used instead
application:
  distro: humble
orchestration:
  deploy_ros_master: false
```

Listing 3.2: Orchestration block before (ROS1) and after (ROS2) the migration. ROS1 Rigelfiles are unaffected; for ROS2, `roscore` is suppressed and DDS discovery is used instead.

3.4 Scheduler Implementation

3.4.1 Scheduler Framework Integration

The custom scheduler was implemented using the Kubernetes Scheduler Framework [28, 39]. The framework structures every scheduling cycle as a fixed pipeline of *extension points*, well-defined stages at which registered plugins are consulted. When a pod awaits placement, the framework first asks the *QueueSort* plugin in which order pending pods should be dequeued; it then runs all *Filter* plugins against every candidate node, eliminating nodes that cannot satisfy the pod's requirements; the surviving nodes are ranked by the *Score* plugins, whose weighted, normalised outputs determine

the winner; the *Reserve* stage lets plugins claim resources on the chosen node before the decision is final (with a guaranteed *Unreserve* rollback if anything later fails); and the *Bind* stage commits the pod-to-node assignment to the API server. Figure 3.2 shows this pipeline and the custom plugins attached to each stage.

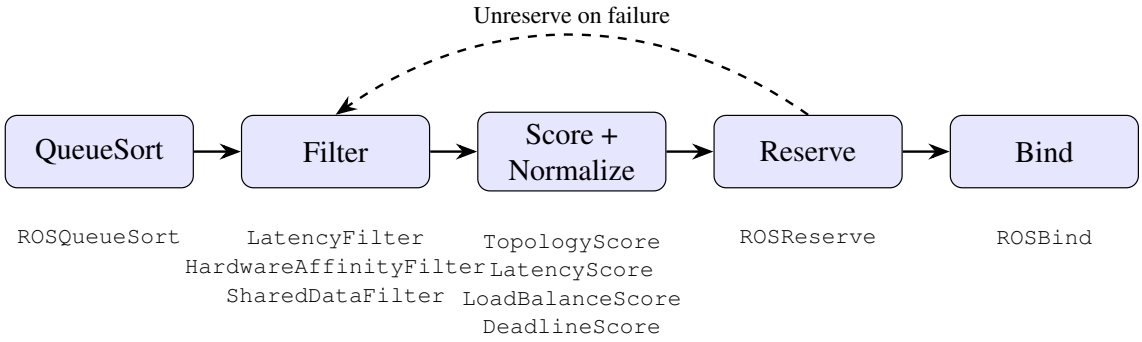


Figure 3.2: The Scheduler Framework pipeline and the ten custom plugins registered at each extension point. A failed bind triggers the automatic `Unreserve` rollback.

The scheduler binary is built from the `k8s.io/kubernetes/pkg/scheduler/framework` package and registered as an out-of-tree plugin set. Running out-of-tree means the binary does not require modifying or forking Kubernetes itself, which is essential for maintainability and compatibility across Kubernetes versions. Each plugin implements the Go interface of its extension point; the two most frequently used interfaces, `FilterPlugin` and `ScorePlugin`, are illustrated in Figure 3.3.

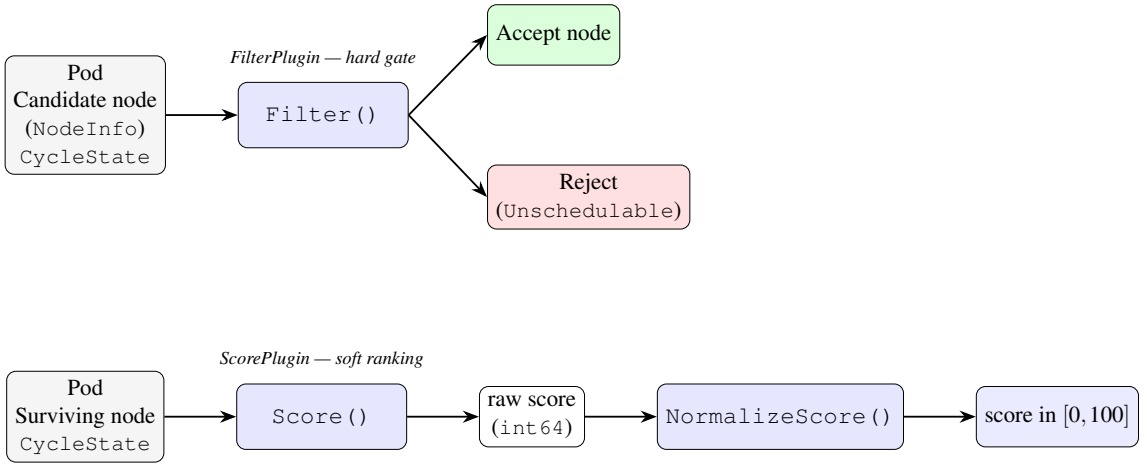


Figure 3.3: The two most frequently implemented plugin interfaces. A `FilterPlugin` acts as a hard gate, run once per candidate node, that either accepts the node or marks it `Unschedulable`. A `ScorePlugin` ranks each surviving node: its raw score is rescaled by the `NormalizeScore()` hook (exposed through `ScoreExtensions()`) into the $[0, 100]$ range that the framework aggregates, weighted per plugin. Both interfaces also expose `Name()`, the identifier under which the plugin is referenced in the scheduler configuration, and both receive the shared per-cycle scratch space `CycleState`, used, for example, to pass pre-computed latency measurements between stages.

The framework calls `Filter()` once per candidate node and `Score()` once per surviving node, so hard constraints are enforced before any ranking takes place. It then aggregates the normalised scores according to the per-plugin weights declared in the configuration, allowing fine-grained control over scheduling priorities.

Ten plugins were implemented across five extension points, as summarised in Table 3.1. The set was chosen so that each robotic requirement identified in Chapter 2 is covered by the stage best suited to enforce it: hard constraints (hardware presence, latency ceilings, mandatory co-location) are Filter plugins, because filtering is the only stage that can guarantee a constraint; soft preferences (topology proximity, latency gradient, load balance, deadline margin) are Score plugins, because scoring trades off competing objectives; urgency ordering is a QueueSort concern; and stateful, node-side bookkeeping that no built-in mechanism covers (DDS domain uniqueness, exclusive device access) lives in Reserve, the only stage with transactional semantics. Each plugin is registered with a factory function that the framework calls at startup, passing any configuration arguments declared in the `KubeSchedulerConfiguration` YAML file.

Table 3.1: Implemented scheduler plugins per extension point.

Extension Point	**Plugin**	**Purpose**
QueueSort	ROSQueueSort	Priority ordering by ROS QoS class
Filter	LatencyFilter	Reject nodes exceeding RTT budget
Filter	HardwareAffinityFilter	Reject nodes missing hardware labels
Filter	SharedDataFilter	Enforce co-location for shared data
Score	TopologyScore	Rank nodes by ROS topology preference
Score	LatencyScore	Rank nodes by TCP round-trip time
Score	LoadBalanceScore	Rank nodes by resource utilisation
Score	DeadlineScore	Rank nodes by deadline budget margin
Reserve	ROSReserve	Claim DDS domains, device slots, fleet registry
Bind	ROSBind	Commit pod-to-node assignment

3.4.2 Annotation Schema

Communication between the Rigel plugin and the scheduler is mediated by a set of Kubernetes pod annotations in the `scheduler.robotics.io/` namespace. The schema was designed to be self-contained within the pod object, avoiding the need for a Custom Resource Definition or a separate API server. Four annotation keys are defined:

- `scheduler.robotics.io/policies` — JavaScript Object Notation (JSON)-encoded `SchedulerPoliciesConfig`, carrying the latency budget, QoS class, ROS2 DDS QoS fields (`reliability`, `durability`, `deadline`), and ROS1 transport fields (`queue_size`, `tcp_nodelay`).
- `scheduler.robotics.io/topology` — JSON-encoded `SchedulerTopologyConfig`, specifying the node running the ROS Master (or a reference DDS peer) and an ordered list of preferred cluster zones.

- `scheduler.robotics.io/affinity` — JSON-encoded `SchedulerAffinityConfig`, declaring hardware requirements and co-location preferences.
- `scheduler.robotics.io/shared-data` — a JSON encoding of `SchedulerSharedDataConfig`, listing sibling pod names and Persistent Volume Claim (PVC) names that must be co-located with this pod.

The Go types that parse these annotations mirror the Pydantic models defined in the Python plugin, ensuring that JSON serialisation and deserialisation are lossless across both languages. An example annotation set injected by the plugin for a teleoperation workload is shown in Listing 3.3.

```
metadata:
  annotations:
    scheduler.robotics.io/policies: >-
      {"latency_budget":"30ms","qos_class":"realtime",
       "reliability":"reliable","deadline":"50ms"}
    scheduler.robotics.io/topology: >-
      {"master_node":"","preferred_zones":[]}
spec:
  schedulerName: kubernetes-scheduler
```

Listing 3.3: Pod annotations for a real-time teleoperation workload, injected by the Rigel orchestration plugin.

In this example, the `policies` annotation declares a 30 ms latency budget (the maximum node Round-Trip Time (RTT) that LatencyFilter will accept), the `realtime` QoS class (which places the pod in the highest scheduling tier of ROSQueueSort), `reliable` DDS delivery semantics, and a 50 ms deadline used by DeadlineScore. The `topology` annotation is present but empty, no ROS Master node or preferred zones are declared, so the topology-related plugins behave as no-ops for this pod. Finally, `spec.schedulerName` routes the pod to the custom scheduler; without this field, the pod would be processed by the default scheduler and the annotations would be ignored.

3.4.3 Queue Sorting: ROSQueueSort

The `ROSQueueSort` plugin replaces the default `PrioritySort` plugin to give ROS workloads a domain-specific first scheduling tier that Kubernetes-level priority classes cannot express. Pods are ordered by three criteria evaluated in sequence.

The first criterion is the ROS QoS tier, derived from the `qos_class` field in the policies annotation. Four tiers are defined: *realtime* (tier 3, highest), *sensor_data* and related data-pipeline classes (tier 2), *system_default* and unannotated pods (tier 1), and *best_effort* (tier 0, lowest). Secondary DDS QoS fields can override the base tier: pods with `reliability: best_effort` are always assigned to tier 0 regardless of other fields, while `liveliness: manual_by_topic`

combined with `reliability: reliable` raises the tier to level 2. This ensures that the pod ordering in the scheduling queue reflects ROS communication semantics: a real-time control loop pod will always be scheduled before a best-effort background task, independently of their Kubernetes PriorityClass. Listing 3.4 shows the three policy fields the plugin reads for this first criterion.

```
scheduler.robotics.io/policies: >-
  {"qos_class":"realtime",
   "reliability":"reliable",
   "liveliness":"manual_by_topic"}
```

Listing 3.4: Policy fields consumed by ROSQueueSort to derive the scheduling tier.

Here `qos_class` sets the base tier (`realtime` in the example, the highest), while the secondary DDS fields adjust it: `reliability` demotes the pod to tier 0 when set to `best_effort`, and `liveliness` set to `manual_by_topic` together with `reliability: reliable` raises the tier to level 2.

The second criterion is the Kubernetes-native pod priority (`spec.priority`), which preserves the operator's existing PriorityClass ordering within each ROS QoS tier. The third criterion is the pod enqueue timestamp (First-In, First-Out (FIFO)), matching the default scheduler behaviour for ties.

Pods with no `scheduler.robotics.io` annotation fall through to the second and third criteria unchanged. Malformed annotations are silently ignored, and the pod falls back to timestamp ordering so that a bad annotation never blocks scheduling.

3.4.4 Filter Plugins

LatencyFilter probes the kubelet HyperText Transfer Protocol Secure (HTTPS) port (10250) on each node's internal Internet Protocol (IP) address with a Transmission Control Protocol (TCP) `connect()` call to measure the network round-trip time from the scheduler to each node. The kubelet port was chosen because it is reachable on every cluster node without requiring Internet Control Message Protocol (ICMP) permissions or a custom DaemonSet, and the TCP connect time reflects the same network path that intra-cluster traffic would use. This probe port is fixed to the standard kubelet port rather than exposed as a configuration option, since it is present on every conformant Kubernetes node; only the probe cache lifetime, the latency budget, and the outlier factor are configurable. The filter applies a three-stage evaluation. First, nodes annotated with `scheduler.robotics.io/graceful-shutdown: "true"` are immediately rejected for new pod placement, providing an integration point for node-level degradation controllers. Second, the measured RTT is compared against the pod's `latency_budget` from its policies annotation, or, when no per-pod budget is specified, against a cluster-wide `maxLatency` value declared in the

scheduler configuration file. Nodes exceeding the ceiling are rejected. Third, an optional cluster-relative outlier check rejects nodes whose RTT exceeds a configurable factor of the cluster-mean RTT. This check relies on a `PreFilter` call that probes all nodes concurrently before the per-node `Filter` calls, storing the full RTT distribution in the `CycleState` context. RTT measurements are cached per node IP with a configurable Time To Live (TTL) (default 30 seconds) to avoid re-probing on every scheduling cycle. Listing 3.5 shows the two inputs the filter consults: the per-pod budget carried in the policies annotation, and the node-level shutdown flag.

```
# on the pod
scheduler.robotics.io/policies: >-
  {"latency_budget":"30ms"}
# on the node, set by a degradation controller
scheduler.robotics.io/graceful-shutdown: "true"
```

Listing 3.5: Inputs read by LatencyFilter: the per-pod latency budget (pod annotation) and the graceful-shutdown flag (node annotation).

The `latency_budget` value is the ceiling the measured RTT must stay under for the node to pass; when a pod omits it, the cluster-wide `maxLatency` from the configuration file applies instead. A node carrying `graceful-shutdown: "true"` is rejected outright, regardless of its measured latency.

HardwareAffinityFilter reads the `hardware_requirements` list from the affinity annotation and matches it against node labels in the `hardware.robotics.io/` namespace. Three matching rules apply depending on the requirement fields: a type-only requirement (e.g., `type: gpu`) requires the label `hardware.robotics.io/gpu` to exist with any value; a type-plus-model requirement requires the label value to match the specified model string; and a type-plus-interface requirement requires the label `hardware.robotics.io/<type>-<interface>` to exist. A node missing any single required label is marked `Unschedulable`. Listing 3.6 shows the shape of the requirement list, with one entry per matching rule.

```
scheduler.robotics.io/affinity: >-
  {"hardware_requirements":[
     {"type":"gpu","model":"nvidia"},
     {"type":"can","interface":"can0"}]}
```

Listing 3.6: Hardware requirements read by HardwareAffinityFilter and matched against node labels.

Each entry names a device `type`; the optional `model` and `interface` fields select between the three matching rules described above. The first entry requires a node labelled `hardware.robotics.io/gpu` with value `nvidia`, and the second requires the label `hardware.robotic`

`s.io/can-can0` to exist. This filter ensures that hardware-sensitive pods (sensor drivers, GPU-accelerated perception pipelines) can only be placed on nodes that physically have the required devices, providing a hard guarantee that the default scheduler's generic affinity rules cannot offer.

SharedDataFilter addresses co-location requirements between ROS nodes that share in-memory data or persistent volumes. It reads two lists from the shared-data annotation: a list of peer pod names and a list of PVC names. For pod co-location, it queries the Kubernetes API to find the current `spec.nodeName` of each named peer; if the peer is already scheduled, only nodes matching its node are allowed. The constraint is skipped if the peer is not yet scheduled to prevent a deadlock where two pods each wait for the other. For volume co-location, the filter checks whether each listed PVC is bound to a PersistentVolume with node affinity; if so, only the node satisfying that affinity passes. ReadWriteMany PVCs are not constrained, as they are accessible from any node. Listing 3.7 shows the two lists the filter reads.

```
scheduler.robotics.io/shared-data: >-
  {"peer_pods":["slam-mapper"],
   "pvcs":["shared-map-pvc"]}
```

Listing 3.7: Co-location fields read by SharedDataFilter: the peer pods and volumes the pod must sit next to.

The `peer_pods` list names sibling pods whose node this pod must match once they are scheduled, and the `pvcs` list names the volume claims whose node affinity the pod must respect.

3.4.5 Score Plugins

TopologyScore ranks nodes based on their proximity to the ROS communication topology declared in the topology annotation. The scoring is expressed as a placement cost where a cost of zero corresponds to the optimal outcome. Exact `master_node` name matches receive cost zero; nodes in `preferred_zones[0]` receive cost one; nodes in `preferred_zones[1]` receive cost two; and so on. Nodes absent from both the master node field and all preferred zones receive cost $n+1$, where n is the number of declared preferred zones. This ensures that adding more zones to the preference list never changes the relative cost of existing zones. A `NormalizeScore` step inverts the raw costs into Kubernetes-compatible scores in the range $[0, 100]$ using the formula $\text{score}_i = 100 \times (\text{maxCost} - \text{cost}_i)/\text{maxCost}$, satisfying the framework's higher-is-better convention. When no topology annotation is present, all nodes receive a uniform bonus equal to the cluster-level `masterNodeWeight` parameter, so the plugin is a no-op for unannotated workloads. Listing 3.8 shows the two fields that drive the cost.

```
scheduler.robotics.io/topology: >-
  {"master_node":"minikube-m02",
   "preferred_zones":["edge-zone-1","edge-zone-2"]}
```

Listing 3.8: Topology fields read by TopologyScore to compute the placement cost.

A node matching `master_node` receives cost zero, a node in the first entry of `preferred_zones` receives cost one, the second entry cost two, and so on, so the order of the zone list directly encodes the preference order.

LatencyScore complements LatencyFilter by providing a continuous gradient among nodes that survived filtering. It measures TCP RTT to each node's kubelet port using the same caching mechanism as LatencyFilter and returns the raw RTT in milliseconds as the raw score. NormalizeScore inverts these values so that the lowest-RTT node receives score 100 and the highest-RTT node receives score 0. Pods without a latency annotation receive a neutral score of 50, making the plugin a no-op for unannotated workloads.

LoadBalanceScore scores nodes by their current CPU, memory, and GPU resource utilisation. Utilisation is computed from the sum of resource requests of all scheduled pods divided by the node's allocatable capacity, requiring no metrics server. The contribution of each resource type is combined using a weighted sum, where the weights are selected by the pod's QoS class: real-time workloads use CPU-heavy weights (65% CPU, 25% memory, 10% GPU) on the assumption that control loops are CPU-bound; best-effort workloads use balanced weights (35% CPU, 35% memory, 30% GPU) reflecting common background ML workloads; and all other workloads use intermediate defaults. An empty node scores 100; a fully saturated node scores 0.

DeadlineScore ranks nodes by their deadline budget margin, defined as $1 - \text{RTT}/\text{deadline}$, where `deadline` is taken from the `scheduler.robotics.io/policies.deadline` annotation and RTT is measured via TCP connect. Nodes where the RTT already exceeds the deadline receive score 0. This plugin is designed to work alongside LatencyFilter: the filter applies the hard latency ceiling, while DeadlineScore provides a soft preference for nodes with the most remaining margin, preferring nodes that can absorb transient network jitter without violating the deadline.

The three remaining score plugins all draw on the same policies annotation, each reading a different field, as shown in Listing 3.9.

```
scheduler.robotics.io/policies: >-
  {"latency_budget":"30ms",
   "qos_class":"realtime",
   "deadline":"50ms"}
```

Listing 3.9: Policy fields consumed by the score plugins: `latency_budget` by LatencyScore, `qos_class` by LoadBalanceScore, and `deadline` by DeadlineScore.

LatencyScore turns the measured RTT into a gradient bounded by `latency_budget`, LoadBalanceScore selects its resource weights from `qos_class`, and DeadlineScore measures the

remaining margin against `deadline`. A pod that omits a given field makes the corresponding plugin a no-op rather than rejecting any node.

3.4.6 Reserve and Bind Plugins

ROSReserve implements the Reserve extension point, called after node selection but before the pod-to-node assignment is committed. It handles three categories of work that the standard Kubernetes scheduler has no concept of.

The first category is DDS domain ID management. In ROS2, every participant on the same network must use a unique domain ID (0 to 232) to avoid receiving each other's traffic. If two ROS2 deployments land on the same node with the same domain ID, their publish-subscribe topics collide. ROSReserve reads the pod's `scheduler.robotics.io/dds-domain` annotation, finds an unoccupied domain ID on the target node, and writes an exclusive claim to the node annotation `scheduler.robotics.io/dds-claims` (a JSON map of domain ID to pod Unique Identifier (UID)). If the requested ID is already claimed by another pod, Reserve returns `Unschedulable` so the scheduler retries with the next candidate. Listing 3.10 shows the pod request and the resulting node-side claim.

```
# on the pod: request an automatically assigned domain
scheduler.robotics.io/dds-domain: "auto"
# on the node: claims written by ROSReserve (domain -> pod UID)
scheduler.robotics.io/dds-claims: >-
  {"5":"a1b2-...","6":"c3d4-..."}
```

Listing 3.10: DDS domain handling in ROSReserve: the pod requests a domain (pod annotation), the plugin records the claim on the chosen node (node annotation).

A value of `auto` lets the plugin pick the first free domain on the target node, while an explicit number requests that exact domain; either way the granted domain is written to the `dds-claims` map on the node, keyed by domain ID, so that a later pod requesting the same ID on the same node is rejected. The exclusive device slots described next are tracked the same way, through a parallel `device-claims` node annotation.

The second category is exclusive hardware device slot reservation. Although the hardware affinity filter ensures that a candidate node has a required device, it cannot prevent two pods from being co-scheduled on the same node both expecting sole access to the same physical interface (for example, `/dev/can0`). ROSReserve reads the hardware requirements from the affinity annotation and, for device types that require exclusive access (Universal Serial Bus (USB), Controller Area Network (CAN) bus, serial, General-Purpose Input/Output (GPIO)), checks the node annotation `scheduler.robotics.io/device-claims`. If any required device is already claimed, Reserve returns `Unschedulable` immediately without writing any partial claim, implementing

all-or-nothing atomicity. GPU devices are intentionally excluded from exclusive reservation, as Kubernetes device plugins handle GPU sharing natively.

The third category is fleet registry updates. ROSReserve writes the preliminary pod-to-node assignment to a ConfigMap named `robot-fleet-assignments`, keyed by pod UID, before binding is committed. This gives operators a live view of robot workload placement for monitoring and debugging without requiring direct access to the Kubernetes API. Fleet registry updates are best-effort: a failure is logged but never causes an `Unschedulable` result, so an optional monitoring store cannot block scheduling.

All three operations are fully reversible: if binding fails after a successful Reserve, the framework calls `Unreserve` automatically, which releases DDS domain claims, device slot claims, and fleet registry entries, leaving the node in a clean state.

ROSBind implements the Bind extension point and commits the pod-to-node assignment by creating a standard `v1.Binding` object, functionally equivalent to the default binder. Its purpose is to provide a clean extension point for future work, such as pre-bind DDS namespace configuration or fleet registry state transitions, without requiring further architectural changes.

For observability, each plugin additionally records its scheduling decisions through the Prometheus Go client library [26], exported via an HTTP endpoint on the scheduler pod. Counter vectors are maintained per plugin for filter decisions (pass or reject) and score computations. These metrics allow operators to observe scheduling behaviour in real time and identify plugins that are frequently rejecting candidate nodes, which may indicate overly restrictive configuration or a mismatch between annotation requirements and cluster capabilities.

3.5 Configuration Interface

The scheduler is configured via a `KubeSchedulerConfiguration` YAML file that specifies which plugins are enabled, their relative weights, and plugin-specific arguments. An extract of the default configuration is shown in Listing 3.11.

```
profiles:
  - schedulerName: kubernetes-scheduler
    plugins:
      queueSort:
        enabled:
          - name: ROSQueueSort
        disabled:
          - name: PrioritySort
      filter:
        enabled:
          - name: LatencyFilter
          - name: HardwareAffinityFilter
          - name: SharedDataFilter
```

```
    score:
      enabled:
        - name: TopologyScore
          weight: 10
        - name: LatencyScore
          weight: 5
        - name: LoadBalanceScore
          weight: 3
        - name: DeadlineScore
          weight: 8
  pluginConfig:
    - name: LatencyFilter
      args:
        maxLatency: "100ms"
        cacheTTL: "30s"
        outlierFactor: 0
    - name: TopologyScore
      args:
        masterNodeWeight: 80
```

Listing 3.11: Extract of the default `KubeSchedulerConfiguration` for the ros-aware-scheduler.

Each element of this configuration has a precise meaning. The `profiles` list allows a single scheduler binary to expose multiple scheduling profiles; here one profile is defined, and its `schedulerName` is the value that pods must reference in `spec.schedulerName` to be handled by it. Inside `plugins`, each extension point lists the plugins to enable: under `queueSort`, the custom `ROSQueueSort` replaces the framework's default `PrioritySort`, which must be explicitly disabled because only one queue-sorting plugin may be active at a time. The three `filter` entries activate the hard-constraint plugins, which need no weights since filtering is binary. The four `score` entries carry weights that multiply each plugin's normalised score before aggregation: with the values shown, topology affinity (10) and deadline margin (8) outweigh the raw latency gradient (5), which in turn outweighs load balancing (3), the priority order expected for typical robotic deployments. Finally, `pluginConfig` passes arguments to individual plugins: `LatencyFilter` receives the cluster-wide RTT ceiling (`maxLatency`, applied when a pod declares no budget of its own), the probe cache lifetime (`cacheTTL`), and the cluster-relative outlier factor (`outlierFactor`, where 0 disables the outlier check); `TopologyScore` receives `masterNodeWeight`, the uniform score applied when a pod declares no topology preference. The configuration file is deployed as a Kubernetes ConfigMap and mounted into the scheduler pod.

To complement direct YAML editing, a lightweight browser-based configuration interface was implemented as a separate binary (`config-ui`) that runs alongside the scheduler; Figure 3.4 shows the interface, and Figure 3.5 shows how it interacts with the rest of the system. The interface

provides two panels. The **Config Panel** exposes toggle switches for enabling or disabling each plugin, numeric sliders for adjusting plugin weights, and a text field for the global `maxLatency` threshold, alongside a live preview of the `KubeSchedulerConfiguration` YAML that the current selections generate. Changes are submitted via an HTTP POST request to `/api/config`, which regenerates the `scheduler.yaml` file on disk. The YAML codec uses targeted regular expressions for reading and a string-builder approach for writing, keeping the binary dependency-free. The **Node Telemetry Panel** displays a live table of all cluster nodes refreshed every ten seconds via `GET /api/nodes`. Each row shows the node name, topology zone, CPU utilisation as a ratio of requested-to-allocatable resources, and whether the node is marked for graceful shutdown. Nodes in graceful shutdown are highlighted in red.

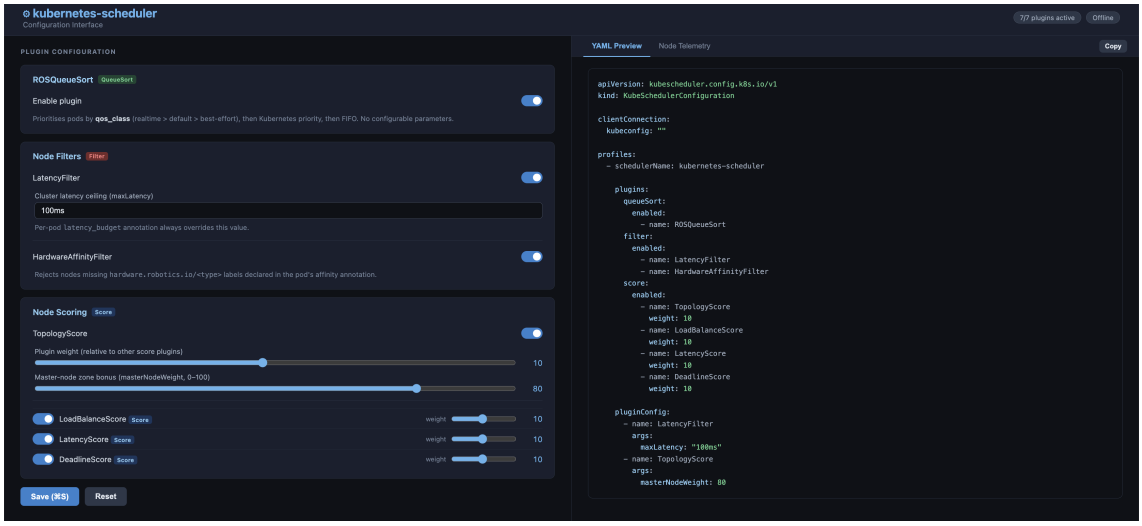


Figure 3.4: The `config-ui` Config Panel: plugin toggles and weight sliders for each extension point (left) and the live preview of the generated `KubeSchedulerConfiguration` YAML (right). The Node Telemetry Panel is available in the second tab.

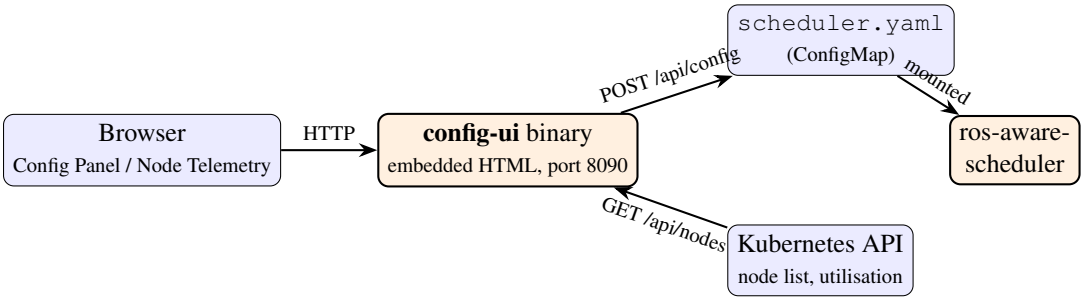


Figure 3.5: Interaction of the `config-ui` binary with the scheduler configuration and the Kubernetes API.

The interface is served on a configurable port (default 8090) and requires no build step, it is a single HyperText Markup Language (HTML) file embedded in the binary using Go's `//go:embed` directive. In production, access should be restricted via `kubectl port-forward` or an ingress with authentication, as the interface has no authentication layer.

3.6 Rigel Framework Integration

3.6.1 Scheduling Configuration Model

The orchestration plugin was extended with a `SchedulerConfig` Pydantic model that maps one-to-one to the annotation schema described in Section 3.4.2. Users declare scheduling preferences in the `orchestration.scheduler` block of their Rigelfile, as shown in Listing 3.12.

```
orchestration:
  deploy_ros_master: false
  scheduler:
    name: kubernetes-scheduler
    policies:
      latency_budget: "30ms"
      qos_class: "realtime"
      reliability: "reliable"
      deadline: "50ms"
    topology:
      preferred_zones:
        - "edge-zone-1"
```

Listing 3.12: Rigelfile `scheduler` block for a real-time teleoperation workload.

The block reads as a direct statement of the workload's requirements. `deploy_ros_master: false` reflects a ROS2 application, where no roscore is needed. Within `scheduler`, the `name` field selects which scheduler profile will process the pods (matching the `schedulerName` in the scheduler's configuration); `policies` carries the per-pod scheduling requirements, the 30 ms latency budget enforced by LatencyFilter, the `realtime` QoS class used by ROSQueueSort and LoadBalanceScore, the `reliable` DDS reliability setting, and the 50 ms deadline used by DeadlineScore; and `topology` expresses a soft preference for nodes in `edge-zone-1`, consumed by TopologyScore.

When the plugin creates the Kubernetes Deployment, it reads the scheduler block, serialises each sub-configuration to JSON, and injects the results as pod template annotations. It also sets `spec.template.spec.schedulerName` to route the pods to the custom scheduler. When the scheduler block is absent, the plugin falls back to the default Kubernetes scheduler transparently, preserving backward compatibility with existing Rigelfiles.

3.6.2 Automatic Scheduler Profile Generation

A key usability contribution of this work is the automatic scheduler profile generator, which infers the annotation set from the ROS package's own metadata without requiring the user to declare scheduling preferences explicitly. The generator was motivated by the observation that a typical ROS developer working on sensor drivers or motion planning is unlikely to know what

`latency_budget` value their application needs, or that their camera package implies a GPU affinity requirement.

The generator analyses the `package.xml` dependency declarations of each ROS package, which every ROS package already contains, and performs a lightweight scan of the Python source files for topic message type patterns. The inference maps package dependencies to scheduling characteristics using four lookup sets:

- **Real-time dependencies**: packages such as `geometry_msgs`, `actionlib`, `moveit_core`, and `ros2_control` indicate a control loop with latency-sensitive requirements.
- **Sensor data dependencies**: packages such as `sensor_msgs`, `cv_bridge`, and `pcl_ros` indicate a sensor data pipeline with moderate latency requirements.
- **GPU dependencies**: packages that indicate GPU hardware is required, such as `darknet_ros`, `tensorrt_ros`, and `depth_image_proc`.
- **USB hardware dependencies**: packages such as `usb_cam`, `serial`, and hardware driver packages indicate that USB-attached devices are required.

The source-level scan refines the inference by detecting message type patterns (e.g., `Twist` implies real-time, `Image` or `CompressedImage` implies sensor data and GPU) that dependency names alone may underspecify. The resulting `SchedulerProfile` carries the inferred QoS class, latency budget, deadline, hardware requirements, and a preferred zone list, and can be serialised either as a Kubernetes annotation map or as a ready-to-paste Rigelfile YAML snippet.

The generator is exposed both as a Python API (`generate_profile(package_path)`) and as a command-line tool (`ros-scheduler-profile`), allowing it to be integrated into CI/CD pipelines. When a Rigelfile includes no explicit `scheduler:` block, the orchestration plugin calls the generator automatically during the deploy phase and uses the result as a best-effort fallback, so an unmodified existing Rigelfile receives a reasonable annotation set without any user intervention.

Together, the components described in this chapter form a complete pipeline: a developer commits a ROS workspace, Rigel containerises and deploys it, and the ros-aware-scheduler places each pod according to requirements that were either declared in one short Rigelfile block or inferred automatically from the package metadata. The next chapter evaluates this pipeline, validating each plugin's placement behaviour and quantifying the end-to-end latency impact of the custom scheduler.

Chapter 4

Evaluation and Results

This chapter presents the evaluation of the system described in Chapter 3. The evaluation was conducted on a local multi-node Minikube cluster and is structured around three evaluation objectives, stated formally in Section 4.1 and referenced throughout the chapter. The evaluation environment is described in Section 4.2, followed by the reference application suite used as workload (Section 4.3). Section 4.4 covers the functional validation of each scheduler plugin, Section 4.5 presents the quantitative latency measurements for the teleoperation experiment, and Section 4.6 evaluates the automatic profile generator. Section 4.7 discusses the overall results against the research questions and the limitations of the evaluation.

4.1 Evaluation Objectives and Metrics

The evaluation addresses three evaluation objectives (EO), numbered EO1–EO3, each mapped to one of the research questions from Section 1.4:

- **EO1 — Functional correctness** (supports RQ1 and RQ3): verify that each scheduler plugin produces the placement decisions it was designed to make, across the principal robotic placement scenarios (hardware affinity, topology preference, shared-data co-location, queue ordering, and DDS domain and device reservation), and that all scenarios deploy end-to-end through the Rigel workflow.
- **EO2 — Quantitative latency performance** (supports RQ1): measure the end-to-end communication latency of a representative latency-sensitive application under the custom scheduler and under the default Kubernetes scheduler, using identical container images and cluster conditions.
- **EO3 — Configuration usability** (supports RQ2): assess how far the configuration burden can be reduced, by measuring the accuracy of the automatic profile generator against manually authored configurations.

Each objective is assessed through a small set of metrics. For EO1, the metric is *placement correctness* (M1): the fraction of deployment repetitions in which the pod lands on the intended

node (each test was repeated three times). For EO2, the metrics are the *mean round-trip time* (M2), the *RTT standard deviation* (M3, capturing jitter), and the *95th and 99th percentile RTT* (M4, capturing tail latency), all computed over more than 1000 samples per scheduler. For EO3, the metric is *inference accuracy* (M5): the fraction of reference packages for which the generator infers the same QoS class and hardware requirements as the manually authored Rigelfile.

4.2 Evaluation Environment

All experiments were conducted on a Minikube [38] cluster running on a single physical machine: an Apple MacBook Pro with an M1 Pro System on a Chip (SoC) (10-core CPU, with 8 performance and 2 efficiency cores) and 32 GB of unified memory, running macOS 26.5. The cluster comprised three virtual nodes: `minikube` (the control-plane node), `minikube-m02`, and `minikube-m03`. Each node ran a containerised Kubernetes v1.35 environment over a shared virtual network bridge (the custom scheduler, built against the v1.31 scheduler libraries, runs unchanged on this newer cluster, consistently with the Scheduler Framework stability discussed in Chapter 3). The custom scheduler ran as a Deployment in the `kube-system` namespace alongside the default scheduler, and pods were routed to each scheduler via `spec.schedulerName`.

This environment has inherent limitations compared to a physical multi-node cluster: network latency between virtual nodes is artificially low and exhibits little geographic variability, which reduces the observable benefit of latency-aware plugins such as LatencyFilter and LatencyScore. The evaluation therefore focuses on demonstrating functional correctness, that the scheduler makes the decisions it is designed to make, and uses the teleoperation RTT experiment as the primary quantitative measure, since it captures end-to-end ROS communication behaviour under both schedulers.

Node labels were applied manually to simulate hardware diversity. `minikube-m02` was labelled with `hardware.robotics.io/gpu=nvidia` and with `hardware.robotics.io/sensor-usb-cam=true`, and `minikube-m03` was loaded with a set of CPU-intensive ballast pods consuming approximately 800 millicores to simulate a heavily loaded node. It should be noted that these labels are synthetic: an Apple Silicon machine has no NVIDIA GPU, and no physical USB camera was attached to the virtual nodes. This is sufficient for the purposes of EO1 because HardwareAffinityFilter operates exclusively on node labels, exactly as it would in a production cluster, where an administrator (or a device-discovery controller) labels nodes according to their installed hardware. What is validated here is the placement logic, not the device drivers; running the GPU workload itself would additionally require a node with the physical accelerator.

4.3 Reference Application Suite

To exercise the scheduler across diverse robotic scenarios, a suite of ten reference applications was developed, five for ROS2 and five for ROS1, each combining a distinct set of scheduler plugins and communication patterns. Table 4.1 summarises the scenarios and the plugins they exercise.

Table 4.1: Reference application suite for scheduler validation.

#	Scenario	Version	Plugins Exercised
1	Teleoperation	ROS2	LatencyFilter, LatencyScore, DeadlineScore
2	Cloud Autonomous	ROS2	LatencyFilter, LatencyScore, TopologyScore
3	Industrial USB Sensor	ROS2	HardwareAffinityFilter, LatencyFilter
4	Multi-Robot Fleet	ROS2	TopologyScore, SharedDataFilter, LatencyFilter
5	Object Detection	ROS2	HardwareAffinityFilter, LatencyFilter, LatencyScore
6	Teleoperation	ROS1	LatencyFilter, LatencyScore, DeadlineScore
7	Cloud Autonomous	ROS1	LatencyFilter, LatencyScore, TopologyScore
8	Industrial USB Sensor	ROS1	HardwareAffinityFilter, LatencyFilter
9	Multi-Robot Fleet	ROS1	TopologyScore
10	Object Detection	ROS1	HardwareAffinityFilter, LatencyFilter, LatencyScore

Each application is a pair of ROS nodes (or a trio in the fleet scenario) containerised using the Rigel toolchain. Applications can be run locally with Docker Compose[1], a tool for defining and running multi-container applications, for rapid iteration, and deployed to a Kubernetes cluster using Rigel's deploy jobs. Each Rigelfile defines three deployment sequences, shown in Listing 4.1: `demo` builds and deploys with the custom scheduler, `benchmark_default` with the default scheduler, and `benchmark_custom` runs an explicit custom-scheduler benchmark. This structure allows each scenario to be evaluated with both schedulers using the same container image, enabling direct comparison.

```
sequences:
  demo:              # custom scheduler
    - build: { scheduler_name: kubernetes-scheduler }
    - deploy
  benchmark_default:    # default scheduler
    - build: { scheduler_name: default-scheduler }
    - deploy
  benchmark_custom:     # explicit custom-scheduler benchmark
    - build: { scheduler_name: kubernetes-scheduler }
    - deploy
```

Listing 4.1: The three deployment sequences declared in each Rigelfile, selecting the scheduler through the `scheduler_name` build argument.

The teleoperation applications implement a publisher-subscriber loop where an operator node publishes velocity commands on `/cmd_vel` and a robot node responds with state messages on `/robot_state`. The operator measures the round-trip time between sending a command and receiving the corresponding state, providing a quantitative end-to-end latency metric. The cloud autonomous applications simulate a Light Detection and Ranging (LiDAR)-scanning robot whose

[1] https://docs.docker.com/compose/

path planning is offloaded to a cloud zone, validating topology-aware placement. The industrial USB sensor applications require the scheduler to enforce hardware affinity for USB-attached devices such as cameras and microphones. The multi-robot fleet application validates SharedDataFilter by requiring the coordinator pod to be co-located with a shared PVC. The object detection applications require GPU node affinity for the ML inference container.

4.4 Functional Validation (EO1)

Functional validation was conducted by deploying each reference application from the suite described in Section 4.3 under both schedulers and verifying that the custom scheduler's placement decisions match the intended plugin behaviour. Each deployment was repeated three times to confirm consistency; placement correctness (M1) was 100% in every scenario reported below.

4.4.1 Hardware Affinity Enforcement (Object Detection)

The object detection scenario exercises the `HardwareAffinityFilter` plugin. The Rigelfile declares a hardware requirement of `type: gpu, model: nvidia`, which the plugin serialises into the pod annotation and which the filter enforces by requiring the label `hardware.robotics.io/gpu=nvidia` on the target node.

Under the default scheduler, the five replicas of the object detection pod were distributed across all three cluster nodes (two on `minikube`, two on `minikube-m02`, one on `minikube-m03`), without respect to GPU availability. Under the custom scheduler, all five replicas were consistently placed on `minikube-m02`, the only node carrying the required label, demonstrating that HardwareAffinityFilter correctly enforces hardware constraints as hard placement rules. A corresponding test was performed with the industrial USB sensor application, where the scheduler correctly placed the pod on the node labelled `hardware.robotics.io/sensor-usb-cam=true` and rejected all unlabelled nodes.

These results contribute to answering **RQ1**: the custom scheduler can enforce hardware locality constraints that the default scheduler cannot express, providing a hard guarantee that pods requiring physical hardware access are placed on nodes where that hardware is available.

4.4.2 Topology-Aware Placement (Cloud Autonomous)

The cloud autonomous scenario exercises the `TopologyScore` plugin. The Rigelfile declares a topology preference for `cloud-zone-1` and `cloud-zone-2`. In the Minikube cluster, the node `minikube-m02` received the label `topology.kubernetes.io/zone=cloud-zone-1` to simulate a cloud zone node. The path planner pod was deployed with both schedulers.

Under the default scheduler, the pod was placed on `minikube` (the node with the most free CPU at the time of scheduling). Under the custom scheduler, TopologyScore assigned a normalised score of 100 to `minikube-m02` (preferred zone, cost 1 after inversion) and a score of 0 to `minikube` and `minikube-m03` (no zone match, cost $n+1=3$). The pod was placed on

`minikube-m02` in all three deployment repetitions, demonstrating that the topology preference overrides pure load balancing.

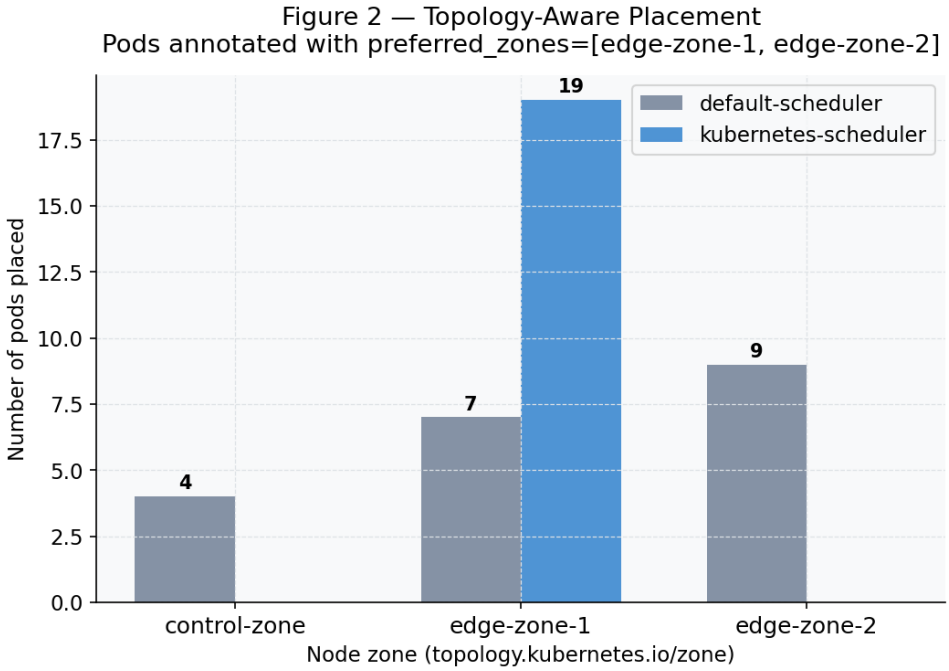


Figure 4.1: Zone distribution of replicas annotated with `preferred_zones=[edge-zone-1, edge-zone-2]` under each scheduler. The default scheduler spreads the replicas across all zones (4/7/9), while the ros-aware-scheduler concentrates every observed replica (19) in the first preferred zone.

To observe this behaviour at a larger scale, a complementary experiment deployed a multi-replica workload annotated with `preferred_zones=[edge-zone-1, edge-zone-2]` under both schedulers, with the cluster zones relabelled accordingly. Figure 4.1 shows the resulting zone distribution: the default scheduler spread the replicas across all three zones according to free capacity (4 in the control zone, 7 in `edge-zone-1`, and 9 in `edge-zone-2`), while the custom scheduler placed every observed replica in `edge-zone-1`, the first preferred zone.

The cost model described in Section 3.4 produces a clear preference ordering without requiring the score values to be tuned: the ordering is determined solely by the zone list in the annotation, and the NormalizeScore step maps whatever cost range appears in a given cycle to $[0, 100]$. This is observable in the scheduler's Prometheus metrics, where the `score_computations` counter for `TopologyScore` increases by the number of candidate nodes per scheduling cycle.

4.4.3 Shared Data Co-location (Multi-Robot Fleet)

The multi-robot fleet scenario exercises the `SharedDataFilter` plugin. The fleet coordinator pod declares a shared-data constraint on a PVC named `shared-map-pvc`. The PVC was created as a ReadWriteMany volume backed by a local storage class.

Since ReadWriteMany PVCs are not subject to node affinity constraints, the SharedDataFilter in this scenario did not restrict placement to a specific node, consistent with the filter's documented behaviour for ReadWriteMany volumes. However, when the experiment was repeated with a ReadWriteOnce PVC bound to a specific node, the filter correctly restricted the coordinator pod to that node across all three repetitions.

The queue ordering behaviour of ROSQueueSort was verified by simultaneously submitting pods with `qos_class: realtime`, `qos_class: best_effort`, and no annotation. Inspection of the scheduler logs confirmed that the realtime pod was dequeued and scheduled first in all cases, followed by the unannotated pod, and finally the best-effort pod, matching the three-tier ordering defined in Section 3.4.

4.4.4 Reserve Plugin Behaviour (DDS Domain and Device Slots)

The ROSReserve plugin was tested by deploying two ROS2 pods on the same node, both annotated with `scheduler.robotics.io/dds-domain: "auto"`. The scheduler assigned domain ID 5 to the first pod and domain ID 6 to the second, writing distinct entries to the `scheduler.robotics.io/dds-claims` node annotation. When a third pod was submitted requesting domain ID 5 explicitly, ROSReserve returned `Unschedulable` and the scheduler retried with the second node, where domain ID 5 was available.

The device slot reservation behaviour was validated by deploying two pods both declaring `hardware_requirements: [type: can, interface: can0]` on the same node. The first pod succeeded; the second received `Unschedulable` from ROSReserve because the claim for `can-can0` was already present in the node annotation. Automatic cleanup via `Unreserve` was verified by manually failing the binding for a pod that had successfully completed Reserve: the node annotation was cleaned of the pod's DDS domain claim before the next scheduling attempt.

4.5 Quantitative Latency Evaluation (EO2)

4.5.1 Teleoperation Experiment Design

The teleoperation application was used as the primary quantitative evaluation scenario because it provides a clear end-to-end latency metric: the round-trip time between the operator node sending a velocity command on `/cmd_vel` and receiving the robot's state response on `/robot_state`. The operator node measures this RTT by timestamping each command at publication and recording the elapsed time when the corresponding state message arrives, yielding one RTT sample per control cycle at 10 Hz; both nodes emit structured JSON timing lines to standard output, so the samples are extracted directly from the pod logs without additional instrumentation.

In this experiment, the operator and robot nodes run as two processes inside a single pod, defined by the teleoperation Rigelfile. The same pod specification is deployed twice: once routed to the default scheduler (the `benchmark_default` job, which injects

`schedulerName: default-scheduler` and no annotations) and once routed to the custom scheduler with the full annotation set (the `benchmark_custom` job). Communication between the two nodes therefore traverses the intra-pod ROS2 stack (DDS over the loopback interface) rather than an inter-node network hop; what the experiment isolates is the effect of the scheduler's placement decision, which cluster node hosts the control loop, on the loop's responsiveness and jitter. To make that decision non-trivial, the experiment was run with the ballast workload described in Section 4.2 active on `minikube-m03`.

The runs collected 1101 RTT samples under the default scheduler and 1034 under the custom scheduler. The custom scheduler configuration used LatencyFilter with a 30 ms budget, LatencyScore, and DeadlineScore with a 50 ms deadline, matching the teleoperation Rigelfile. These values were not arbitrary: as discussed in Chapter 2, human-in-the-loop teleoperation demands sub-100 ms end-to-end latency, and control loops typically require responses below 50 ms. The 50 ms deadline encodes the control-loop bound directly, and the 30 ms per-node RTT budget leaves the remaining margin of the deadline available for application processing and DDS middleware overhead, so that a node admitted by the filter can still meet the deadline under transient jitter.

4.5.2 Results and Validity

Table 4.2 presents the descriptive statistics of both runs, and Figure 4.2 visualises the comparison (mean with standard deviation, and the p95/p99 tail percentiles).

Table 4.2: Teleoperation RTT statistics under default and custom schedulers.

Scheduler	Placed on	n	Mean (ms)	Std (ms)	p95 (ms)	p99 (ms)
default-scheduler	`minikube-m02`	1101	0.71	1.12	1.01	3.41
kubernetes-scheduler	`minikube-m02`	1034	0.66	0.81	0.98	1.56

The first observation concerns placement: both schedulers placed the teleoperation pod on `minikube-m02`, away from the ballast-loaded `minikube-m03`. This is the expected outcome, because the ballast pods declare CPU requests, making the load visible to the request-based scoring of the default scheduler as well as to the custom scheduler's LoadBalanceScore. With identical placement, the two RTT distributions are, as expected, very close: the means (0.71 ms versus 0.66 ms) and 95th percentiles (1.01 ms versus 0.98 ms) are practically indistinguishable. The custom-scheduler run exhibited a lower standard deviation (0.81 ms versus 1.12 ms) and 99th percentile (1.56 ms versus 3.41 ms), but since both pods ran on the same node, this difference is attributable to run-to-run variability of the shared host rather than to the scheduling decision, and no scheduler advantage is claimed from it.

The substantive results of the experiment are therefore twofold. First, the custom scheduler imposes *no measurable latency penalty*: routing a pod through ROSQueueSort, the three filter plugins, the four score plugins, and the Reserve/Bind pipeline yields the same placement quality and the same RTT profile as the default scheduler when both have visibility of the cluster load. Second, the custom scheduler adds enforcement that the default scheduler cannot provide: every

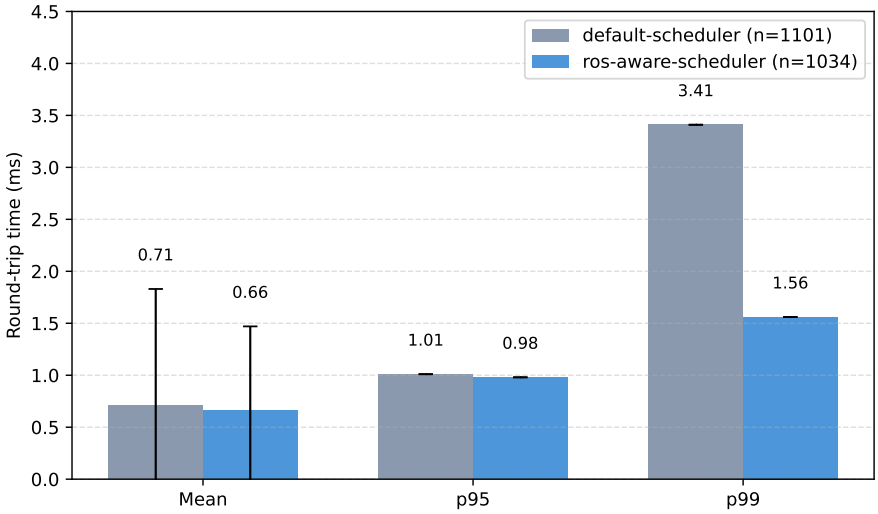


Figure 4.2: Teleoperation RTT under each scheduler: mean (with standard deviation as error bar), 95th and 99th percentiles.

one of the 1034 samples remained far below the 30 ms latency budget admitted by LatencyFilter and the 50 ms deadline scored by DeadlineScore, and these bounds are *guaranteed at admission time*, a node that violated the budget would have been rejected, whereas the default scheduler would have placed the pod regardless. The placement-quality advantage of the custom scheduler appears precisely when the two schedulers disagree, which the functional validation demonstrates: in the topology experiment (Figure 4.1) the default scheduler scattered replicas across all zones while the custom scheduler honoured the declared preference, and in the hardware affinity tests the default scheduler placed GPU workloads on nodes without the required hardware.

Two observations bound the validity of these results. First, communication between the operator and robot is intra-pod (DDS over loopback), so the absolute RTT values are far lower than in a deployment where the two nodes run in separate pods on different machines; the experiment measures the placement effect on a co-located control loop, not network path selection. Second, the Minikube virtual nodes share a single physical host, so even a cross-node deployment in this environment would traverse the host's bridge interface rather than a physical network. In a physical heterogeneous cluster, where inter-node round-trips reach tens of milliseconds and real CPU load is not always declared through requests, the latency probing of LatencyFilter and LatencyScore differentiates placements in situations where the request-based default scheduler is blind, and a future evaluation on such hardware would be expected to show placement divergence and correspondingly larger RTT differences.

4.6 Automatic Profile Generator Evaluation (EO3)

The profile generator was evaluated against the five ROS2 reference application packages by running `ros-scheduler-profile <package_path>` and comparing the inferred annotation

set to the manually authored Rigelfile for each scenario. Table 4.3 summarises the inference results.

Table 4.3: Profile generator inference accuracy against manually authored Rigelfile configurations.

Scenario	Inferred QoS	Expected QoS	HW inferred	HW expected
Teleoperation	realtime	realtime	none	none
Cloud Autonomous	best_effort	sensor_data	none	none
Industrial USB Sensor	sensor_data	sensor_data	usb	usb
Multi-Robot Fleet	best_effort	realtime	none	none
Object Detection	sensor_data	sensor_data	gpu	gpu

The generator correctly inferred the QoS class for three of the five scenarios (M5 = 60% for QoS class, 100% for hardware requirements). For the teleoperation application, the presence of `geometry_msgs` in the dependencies (via the `Twist` type used for velocity commands) triggered the `_REALTIME_DEPS` heuristic, matching the manual configuration. The industrial USB sensor and object detection applications were also classified correctly, with the USB hardware requirement inferred from `usb_cam` and the GPU requirement from `cv_bridge` and `depth_image_proc`.

Two misclassifications were observed. The cloud autonomous application was classified as `best_effort` because its primary dependencies (`nav_msgs`, `geometry_msgs`) did not intersect with the `_SENSOR_DEPS` set (the `geometry_msgs` hit went to `_REALTIME_DEPS`, but the source scan found no Twist pattern in a planner-side package). In practice, the generated profile would still produce functional scheduling, the pod would be admitted to the cluster and placed without latency violation, but the latency budget would be more conservative than necessary (500 ms instead of 150 ms). The multi-robot fleet application was classified as `best_effort` because the coordinator package's dependencies did not include any real-time or sensor-data packages; its communication is primarily through `std_msgs` and fleet status topics not covered by the current inference sets.

These results demonstrate that the inference is reliable for common ROS patterns (manipulation, perception, hardware drivers) but may underspecify workloads whose real-time character is not directly reflected in their package.xml dependencies. The tool is designed to be overridden by an explicit `scheduler:` block in the Rigelfile, so misclassification produces a conservative but non-blocking default rather than an incorrect hard constraint.

4.7 Discussion

4.7.1 Summary of Results Against Research Questions

The evaluation results can be mapped to the three research questions stated in Section 1.4.

RQ1 — achieving low latency and high QoS through custom Kubernetes scheduling, is addressed by the LatencyFilter, LatencyScore, and DeadlineScore plugins, which together implement a three-layer strategy: hard rejection of nodes exceeding the latency budget, continuous gradient ranking among survivors, and deadline-margin preference. The teleoperation experiment (EO2) shows that this strategy adds no measurable latency overhead relative to the default scheduler,

both produced equivalent placements and RTT profiles when the cluster load was visible to both, while guaranteeing at admission time that every placement satisfies the declared 30 ms budget and 50 ms deadline. The placement advantage materialises when the two schedulers disagree, as the functional validation (EO1) demonstrates: LatencyFilter correctly rejects nodes exceeding a configured RTT budget, the topology experiment shows the default scheduler scattering replicas that the custom scheduler concentrates in the preferred zone, and the hardware affinity tests show the default scheduler placing GPU workloads on nodes without the required hardware.

RQ2 — configurable user interaction, is addressed at two levels. The Rigelfile `scheduler:` block provides a declarative YAML interface requiring no Kubernetes expertise, directly expressing domain-relevant concepts (latency budget, QoS class, hardware type) rather than Kubernetes-internal objects. The automatic profile generator further reduces the configuration burden (EO3): for three of the five reference scenarios, the correct QoS class and hardware requirements were inferred without any manual annotation, and in all five cases the generated profile was safe in the sense that no unnecessary pods were rejected. The configuration User Interface (UI) provides an additional visual interaction mode, though its primary use case is development and debugging rather than production configuration.

RQ3 — reliable integration with a containerisation framework for robotic applications, is addressed by the plugin's backward-compatible extension of the Rigelfile schema and the annotation-based communication contract. All ten reference applications deploy successfully through the Rigel workflow (EO1), and the `benchmark_default` and `benchmark_custom` job variants confirm that switching between the default and custom schedulers requires only changing the `scheduler.name` field in the Rigelfile, with no changes to application code or container images.

4.7.2 Limitations

Several limitations of the current evaluation should be acknowledged. First, all experiments were conducted on a single physical host using Minikube virtual nodes. The intra-cluster RTT in this environment is artificially low (sub-5ms), which limits the visibility of the latency-aware plugins' effectiveness. A future evaluation using a real heterogeneous cluster with measurable geographic or topological latency differences between nodes would more directly demonstrate the value of LatencyFilter and TopologyScore.

Second, the evaluation did not include large-scale or failure injection scenarios, which would have characterised scheduler scalability and resilience under varying cluster sizes, degraded network conditions, and node failures. These scenarios were descoped due to infrastructure constraints and remain as directions for future work.

Third, the profile generator evaluation was conducted on a small set of five handcrafted reference applications, which may not be representative of the full diversity of ROS packages in the wild. A broader evaluation against a sample of open-source ROS packages would provide more reliable accuracy estimates for the inference heuristics.

Fourth, usability interviews with external robotics developers were not conducted within the dissertation timeline. Qualitative feedback from developers who had not participated in building

the system would be valuable for assessing whether the Rigelfile `scheduler:` interface and the generated profiles are intuitive in practice.

Overall, the functional validation confirmed that every plugin makes the placement decisions it was designed to make, the teleoperation experiment showed that the custom scheduler matches the default scheduler's placement quality with no measurable latency penalty while enforcing the declared latency budget and deadline on every placement, and the profile generator proved a useful default for developers who author no scheduling configuration. The next chapter draws the overall conclusions of this work and details the future work directions identified above.

Chapter 5

Conclusions and Future Work

This dissertation addressed the gap between cloud-native orchestration and the specific requirements of robotic applications, by designing, implementing, and evaluating a dedicated Kubernetes scheduler for ROS workloads, integrated with the Rigel containerisation framework. This chapter summarises the work and its contributions, revisits the research questions, and identifies directions for future work.

5.1 Summary of Contributions

The starting point of this work was the observation, substantiated in Chapter 2, that although Kubernetes provides an extensible Scheduler Framework and ROS applications have been successfully containerised and orchestrated, no existing system combined the two: placement decisions for robotic workloads remained delegated to a default scheduler that is blind to latency budgets, QoS classes, hardware locality, and ROS communication semantics. The work produced the following concrete contributions:

- **ROS2 support in the Rigel orchestration plugin.** The plugin, previously limited to ROS1, now detects the ROS distribution declared in the Rigelfile, suppresses ROS Master deployment for DDS-based ROS2 workloads, generates the appropriate environment configuration, and supports the `colcon` build toolchain, while remaining fully backward compatible with existing ROS1 Rigelfiles. This is a standalone contribution to the Rigel ecosystem.
- **The ros-aware-scheduler**, a custom out-of-tree Kubernetes scheduler implementing ten plugins across five Scheduler Framework extension points: queue prioritisation by ROS QoS class, latency filtering and scoring, hardware affinity filtering, shared-data co-location, topology-aware scoring, load-balance scoring, deadline-margin scoring, and ROS-specific reservation of DDS domain IDs and exclusive hardware devices.
- **A declarative configuration interface** consisting of a self-contained pod annotation schema, a concise `scheduler` block in the Rigelfile expressed in domain terms (latency, QoS, hardware, topology), and a lightweight web interface for configuration and node telemetry.

- **An automatic profile generator** that infers scheduling requirements from the `package.xml` metadata and source patterns that every ROS package already contains, providing a safe default when developers author no scheduling configuration.
- **A validation methodology** comprising ten reference applications, five per ROS version, covering teleoperation, cloud-offloaded autonomy, hardware-bound sensing, multi-robot coordination, and GPU-bound perception, together with robotics-specific evaluation metrics.

5.2 Answers to the Research Questions

RQ1 — How can a Kubernetes scheduler be implemented to achieve low latency and high QoS, as required by ROS based systems? The answer demonstrated by this work is a layered combination of native Scheduler Framework plugins: hard latency ceilings enforced at the Filter stage, continuous latency and deadline-margin gradients at the Score stage, and urgency ordering at the QueueSort stage, all driven by per-pod declarations rather than hardcoded policy. The evaluation showed that this strategy carries no measurable latency cost: in the teleoperation experiment the custom scheduler matched the default scheduler's placement and RTT profile (mean 0.66 ms versus 0.71 ms, over more than a thousand samples each) while guaranteeing the declared 30 ms budget and 50 ms deadline at admission time, and every placement constraint validated functionally (hardware affinity, topology preference, co-location, DDS domain exclusivity) was enforced correctly in all repetitions.

RQ2 — Which methods should remain flexible and configurable by the user, and how should the user interact with this environment? The implemented system delegates to the user exactly the decisions that are application-specific, latency budget, QoS class, deadline, hardware requirements, topology preferences, and co-location constraints, and keeps everything else (probing, caching, normalisation, reservation bookkeeping) internal to the scheduler. The user interacts at three levels of decreasing effort: explicit Rigelfile declarations, the web-based configuration interface for cluster-level tuning, and the automatic profile generator, which inferred correct QoS classes for three of five reference applications and correct hardware requirements for all five, without any user input.

RQ3 — How can the proposed scheduling approach be reliably integrated into a containerisation framework for robotic applications with Kubernetes support? The integration is achieved through a deliberately thin contract: the orchestration plugin serialises the user's declarations into pod annotations and a scheduler name, and the scheduler consumes only those. Neither component depends on the other at runtime, the Rigelfile extension is backward compatible, and switching between the default and custom schedulers requires changing a single field. All ten reference applications deploy end-to-end through the Rigel workflow.

5.3 Impact

Beyond the research questions, the system has practical relevance for several classes of robotic applications. Teleoperation systems gain the ability to specify and enforce strict latency budgets; multi-computer robotic platforms, such as mobile manipulators that distribute perception, navigation, and safety functions across machines, benefit from topology-aware placement; perception pipelines requiring GPU acceleration obtain hard hardware-affinity guarantees; and edge-cloud hybrid systems can express zone preferences that the scheduler honours over plain load balancing. By exposing all of this through the Rigelfile and the automatic profile generator, the work lowers the Kubernetes adoption barrier for robotics teams without infrastructure engineering backgrounds, and provides a platform on which researchers can build further domain-specific scheduling algorithms without reimplementing core orchestration logic.

5.4 Future Work

The limitations identified in Chapter 4 translate directly into future work directions:

- **Evaluation on physical heterogeneous clusters.** The single-host Minikube environment compresses inter-node latency to sub-5 ms values. Deploying the scheduler on a physical edge-cloud cluster with measurable topological latency differences, spanning heterogeneous CPU architectures (for example Advanced RISC Machine (ARM) and x86) and GPU-equipped servers, would quantify the full benefit of LatencyFilter, LatencyScore, and TopologyScore, and would allow measuring the decrease in onboard resource usage (and hence energy consumption) when robot workloads are offloaded with confidence.
- **Scalability and resilience characterisation.** Repeating the evaluation with larger clusters, degraded network conditions, and injected node failures would characterise scheduling throughput, decision latency, and recovery behaviour under stress, complementing the functional validation presented here.
- **Broader profile generator validation.** Evaluating the inference heuristics against a representative sample of open-source ROS packages would yield more reliable accuracy estimates and would likely suggest extensions to the dependency lookup sets, such as coverage for fleet-coordination packages whose real-time character is not visible in their dependencies.
- **Usability studies.** Semi-structured interviews and task-based studies with robotics developers external to the project would assess whether the Rigelfile `scheduler:` block and the generated profiles are intuitive in practice, and would guide the evolution of the configuration interface.
- **Scheduler extensions.** The ROSBind plugin was deliberately designed as a clean extension point: pre-bind DDS namespace configuration, fleet registry state transitions, and dynamic rescheduling in response to runtime telemetry (e.g., moving a pod when its measured latency

degrades) are natural next steps. Hardening the configuration interface with authentication would make it suitable for production use.

- **Simulation-based and competition-scenario evaluation.** Exploring Gazebo to exercise the scheduler under realistic robot dynamics and sensor workloads, before validating on physical hardware, would provide a reproducible testbed that bridges the gap between the current Minikube setup and a full physical cluster. Subsequently, adopting robotics-competition scenarios as evaluation workloads would stress the scheduler with the heterogeneous, latency-sensitive, and multi-robot task mixes that characterise real deployments, offering a richer and more demanding benchmark for placement quality.

In closing, this dissertation showed that the gap between cloud-native orchestration and robotic workloads can be closed without forking Kubernetes, without new cluster-side APIs, and without demanding Kubernetes expertise from robotics developers: ROS-aware placement logic fits naturally into the Scheduler Framework, and a thin declarative contract suffices to put it in the hands of the tools, and the developers, that already build and deploy robotic software.

References

[1] Marta Cardoso, Rafael Arrais, and Armando Sousa. "Automatic Fault Detection and Diagnosis in ROS-Based Robotic Systems Using Generative AI: A Systematic Literature Review". In: *Applied Sciences* 16.11 (2026), p. 5545. DOI: 10.3390/app16115545. URL: https://doi.org/10.3390/app16115545.

[2] Stefano Carpin. "Scheduling Problems for Robotics in Precision Agriculture". In: *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2022, pp. 1357–1361. DOI: 10.1109/ISCAS48785.2022.9937482. URL: https://doi.org/10.1109/ISCAS48785.2022.9937482.

[3] Oskar Casquero et al. "Distributed scheduling in Kubernetes based on MAS for Fog-in-the-loop applications". In: *2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 2019, pp. 1213–1217. DOI: 10.1109/ETFA.2019.8869219. URL: https://doi.org/10.1109/ETFA.2019.8869219.

[4] Abhishek Dasgupta and Pradipta Kumar Banerjee. *k8s-custom-scheduler: Custom Kubernetes Scheduler for GPU Workloads*. https://github.com/IBM/k8s-custom-scheduler. accessed Oct. 21, 2025. 2018.

[5] Bowen Dong et al. "Review of applicability of heuristic scheduling algorithms in mobile robotics". In: *Robotic Intelligence and Automation* 46 (2024), pp. 30–48. DOI: 10.1108/RIA-05-2024-0121. URL: https://doi.org/10.1108/RIA-05-2024-0121.

[6] Tomoya Fujita. *ROS with Kubernetes*. Open Robotics Discourse. accessed Oct. 09, 2025. Nov. 2022. URL: https://discourse.openrobotics.org/t/ros-with-kubernetes/28107.

[7] Grid Dynamics. *Kubernetes use cases beyond container scheduling*. Grid Dynamics Blog. accessed Oct. 21, 2025. Jan. 2025. URL: https://www.griddynamics.com/blog/kubernetes-use-cases.

[8] Harichane, Makhlouf, and Belalem. "A Proposal of Kubernetes Scheduler Using Machine-Learning on CPU/GPU Cluster". In: *Intelligent Algorithms in Software Engineering*. Springer, 2020, pp. 598–607. DOI: 10.1007/978-3-030-51965-0_50. URL: https://doi.org/10.1007/978-3-030-51965-0_50.

[9] Nicholas Pedrosa Hopf. "Container Orchestration Solution for Robot Software". MA thesis. Faculdade de Engenharia, Universidade do Porto, 2025. URL: https://repositorio-aberto.up.pt/handle/10216/166251.

[10] Aled James et al. "A Low Carbon Kubernetes Scheduler". In: *ICT4S2019: 6th International Conference on Information and Communication Technology for Sustainability*. 2019. URL: https://ceur-ws.org/Vol-2382/ICT4S2019_paper_28.pdf.

[11] Zhaolong Jian et al. “DRS: A deep reinforcement learning enhanced Kubernetes scheduler for microservice-based system”. In: *Software: Practice and Experience* 53.11 (2023), pp. 2124–2145. DOI: 10.1002/spe.3284. URL: https://doi.org/10.1002/spe.3284.

[12] Murat Karslioglu. *Kubernetes - A Complete DevOps Cookbook: Build and manage your applications, orchestrate containers, and deploy cloud-native services*. Packt Publishing, 2020. ISBN: 9781838828042. URL: https://www.packtpub.com/en-us/product/kubernetes-a-complete-devops-cookbook-9781838828042.

[13] Barbara Kitchenham and Stuart Charters. *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Tech. rep. EBSE 2007-001. Keele University and Durham University, 2007.

[14] Bastian Lampe et al. “RobotKube: Orchestrating Large-Scale Cooperative Multi-Robot Systems with Kubernetes and ROS”. In: *2023 IEEE 26th International Conference on Intelligent Transportation Systems (ITSC)*. 2023, pp. 2719–2725. DOI: 10.1109/ITSC57777.2023.10422370. URL: https://doi.org/10.1109/ITSC57777.2023.10422370.

[15] Tianzhe Li et al. “CAROKRS: Cost-Aware Resource Optimization Kubernetes Resource Scheduler”. In: *2024 9th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*. 2024, pp. 127–133. DOI: 10.1109/ICCCBDA61447.2024.10569976. URL: https://doi.org/10.1109/ICCCBDA61447.2024.10569976.

[16] Francesco Lumpp et al. “A container-based design methodology for robotic applications on kubernetes edge-cloud architectures”. In: *2021 Forum on Specification & Design Languages (FDL)*. 2021, pp. 1–8. DOI: 10.1109/FDL53530.2021.9568376. URL: https://doi.org/10.1109/FDL53530.2021.9568376.

[17] Francesco Lumpp et al. “A Design Flow Based on Docker and Kubernetes for ROS-based Robotic Software Applications”. In: *ACM Transactions on Embedded Computing Systems* 23.5 (2023), pp. 1–24. DOI: 10.1145/3594539. URL: https://doi.org/10.1145/3594539.

[18] Francesco Lumpp et al. “Enabling Kubernetes Orchestration of Mixed-Criticality Software for Autonomous Mobile Robots”. In: *IEEE Transactions on Robotics* 40 (2024), pp. 540–553. DOI: 10.1109/TRO.2023.3334642. URL: https://doi.org/10.1109/TRO.2023.3334642.

[19] Steven Macenski et al. “Robot Operating System 2: Design, architecture, and uses in the wild”. In: *Science Robotics* 7.66 (2022), eabm6074. DOI: 10.1126/scirobotics.abm6074. URL: https://arxiv.org/abs/2211.07752.

[20] Pedro Melo, Rafael Arrais, et al. “A Container-Based Framework for Developing ROS Applications”. In: *2022 IEEE 20th International Conference on Industrial Informatics (INDIN)*. 2022. DOI: 10.1109/INDIN51773.2022.9976083. URL: https://doi.org/10.1109/INDIN51773.2022.9976083.

[21] Tarek Menouer. “KCSS: Kubernetes container scheduling strategy”. In: *The Journal of Supercomputing* 77 (2021), pp. 4267–4293. DOI: 10.1007/s11227-020-03427-3. URL: https://doi.org/10.1007/s11227-020-03427-3.

[22] Michael Chima Ogbuachi et al. “Context-aware Kubernetes scheduler for edge-native applications on 5G”. In: *Journal of Communications Software and Systems* 16.1 (2020), pp. 85–94. DOI: 10.24138/jcomss.v16i1.1027. URL: https://doi.org/10.24138/jcomss.v16i1.1027.

[23] Open Robotics. *ROS.org | Powering the world's robots*. Ros.org. 2020. URL: https://www.ros.org/.

[24] Open Source Robotics Foundation, Inc. *2024 ROS Metrics Report*. Open Robotics Discourse. accessed Oct. 09, 2025. Mar. 2025. URL: https://discourse.openrobotics.org/t/2024-ros-metrics-report/42354.

[25] Matthew J. Page et al. "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews". In: *BMJ* 372 (2021), n71. DOI: 10.1136/bmj.n71. URL: https://doi.org/10.1136/bmj.n71.

[26] Prometheus Authors. *Prometheus: Monitoring System and Time Series Database*. Accessed June 11, 2026. 2025. URL: https://prometheus.io/.

[27] Morgan Quigley et al. "ROS: an open-source Robot Operating System". In: *ICRA Workshop on Open Source Software*. 2009. URL: http://www.robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf.

[28] Zeineb Rejiba et al. "Custom Scheduling in Kubernetes: A Survey on Common Problems and Solution Approaches". In: *ACM Computing Surveys* 55.7 (2022), pp. 1–36. DOI: 10.1145/3544788. URL: https://doi.org/10.1145/3544788.

[29] Catarina Rema et al. "Task Scheduling with Mobile Robots—A Systematic Literature Review". In: *Robotics* 14.6 (2025), p. 75. DOI: 10.3390/robotics14060075. URL: https://doi.org/10.3390/robotics14060075.

[30] Open Robotics. *ROS 2 Documentation — ROS 2 Documentation: Foxy documentation*. docs.ros.org. URL: https://docs.ros.org/en/foxy/index.html.

[31] John Rothman et al. "An RL-Based Model for Optimized Kubernetes Scheduling". In: *2023 IEEE 31st International Conference on Network Protocols (ICNP)*. 2023, pp. 1–6. DOI: 10.1109/ICNP59255.2023.10355623. URL: https://doi.org/10.1109/ICNP59255.2023.10355623.

[32] Khaldoun Senjab et al. "A survey of Kubernetes scheduling algorithms". In: *Journal of Cloud Computing* 12.1 (2023), p. 87. DOI: 10.1186/s13677-023-00471-1. URL: https://doi.org/10.1186/s13677-023-00471-1.

[33] Solwey Consulting. *Kubernetes for Beginners: Understanding the Basics and Real-World Implementations*. Solwey Consulting Blog. accessed Oct. 21, 2025. 2024. URL: https://www.solwey.com/posts/kubernetes-for-beginners-understanding-the-basics-and-real-world-implementations.

[34] Shengbo Song et al. "Gaia scheduler: A kubernetes-based scheduler framework". In: *2018 IEEE Intl Conf on Parallel & Distributed Processing with Applications*. 2018, pp. 252–259. DOI: 10.1109/BDCloud.2018.00048. URL: https://doi.org/10.1109/BDCloud.2018.00048.

[35] SYSGO. *Kubernetes at the Edge: Merging Cloud-Native Flexibility with Real-Time Computing*. SYSGO Blog. accessed Oct. 21, 2025. 2025. URL: https://www.sysgo.com/blog/article/kubernetes-at-the-edge-merging-cloud-native-flexibility-with-real-time-computing.

[36] The Go Authors. *The Go Programming Language*. Accessed June 11, 2026. 2025. URL: https://go.dev/.

[37] The Kubernetes Authors. *Kubernetes v1.31 Release Notes*. Accessed June 11, 2026. 2024. URL: https://kubernetes.io/blog/2024/08/13/kubernetes-v1-31-release/.

[38] The Kubernetes Authors. *Minikube: Local Kubernetes Documentation*. Accessed June 11, 2026. 2025. URL: https://minikube.sigs.k8s.io/docs/.

[39] The Kubernetes Authors. *Scheduling Framework — Kubernetes Documentation*. Accessed June 11, 2026. 2020. URL: https://kubernetes.io/docs/concepts/scheduling-eviction/scheduling-framework/.

[40] U.S. Air Force. *U-2 Federal Laboratory Successfully Runs Kubernetes on Aircraft*. U.S. Air Force News. accessed Oct. 09, 2025. Oct. 2020. URL: https://www.af.mil/News/Article-Display/Article/2360000/u-2-federal-laboratory-successfully-runs-kubernetes-on-aircraft/.

[41] Abhishek Verma et al. "Large-scale cluster management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. Bordeaux, France: ACM, Apr. 2015. DOI: 10.1145/2741948.2741964. URL: https://dl.acm.org/doi/10.1145/2741948.2741964.

[42] Chongkun Xia et al. "Microservice-based cloud robotics system for intelligent space". In: *Robotics and Autonomous Systems* 110 (2018), pp. 139–150. DOI: 10.1016/J.ROBOT.2018.10.001. URL: https://doi.org/10.1016/J.ROBOT.2018.10.001.

[43] Xu et al. "A Scalable Resource Management Architecture for Industrial Fog Robots". In: *Intelligent Robotics and Applications (ICIRA 2021)*. Springer, 2021, pp. 79–91. DOI: 10.1007/978-3-030-89095-7_7. URL: https://doi.org/10.1007/978-3-030-89095-7_7.

[44] Yongzhou Zhang et al. "A Comprehensive Modeling and Scheduling Approach for Allocating Distributed Multi-Robot Software to the Edge/Cloud". In: *2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2024. DOI: 10.1109/IROS58592.2024.10802443. URL: https://doi.org/10.1109/IROS58592.2024.10802443.

[45] Naweiluo Zhou et al. "Container orchestration on HPC systems through Kubernetes". In: *Journal of Cloud Computing* 10.1 (2021), p. 16. DOI: 10.1186/s13677-021-00231-z. URL: https://doi.org/10.1186/s13677-021-00231-z.