FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Container Orchestration Solution for Robot Software

Nicholas Hopf

Master in Electrical and Computers Engineering

Supervisor: Armando Sousa, PhD

Second Supervisor: Rafael Arrais, PhD

March 31, 2025

Abstract

Robot software applications have increased in complexity, making it harder to manage code updates with minimal downtime. Deploying these applications has also become more challenging, especially as distributed environments gain popularity by offloading computational tasks to cloud or edge resources. Although orchestration platforms like Kubernetes have improved software management in other domains, their use in robotics remains limited, with developers often relying on manual processes or application-specific scripts. This dissertation addresses that gap by introducing an orchestration module for the Rigel framework, which automates the containerization of ROS-based robot applications, enables zero-downtime rolling updates, and includes features that can be extended to offload resource-intensive workloads to external machines.

In developing this module, Kubernetes is adapted to robotic requirements by implementing application-aware readiness checks that prevent updates from interrupting ongoing tasks, while also abstracting most of the complex networking setup for communication between ROS nodes. A declarative interface in the Rigel configuration file ensures that system administrators and developers only need to specify high-level parameters, leaving the module to manage container scheduling, environment setup, and data persistence, while remaining flexible to accommodate custom orchestration logic when the user requires it. An integrated observability stack combining Prometheus, Grafana, and Loki offers real-time performance and log data with sensible defaults, helping developers diagnose resource bottlenecks or software anomalies as they arise, as well as facilitating integration of custom application metrics into a unified Grafana dashboard. Validation tests focus on common ROS communication patterns for services, actions, and publish-subscribe topics, demonstrating that even high-frequency data streams can remain stable throughout application software updates.

Experimental results, conducted on a resource-constrained environment, validate the execution of zero-downtime software updates, minimal message loss, and reliable performance when subject to stress testing. These findings indicate that cloud-native approaches can be extended to robotics, removing manual overhead still present in teams that need to deploy and scale robot software. As a result, the proposed orchestration module allows developers to concentrate on developing application logic rather than tending to infrastructure details, thus providing a new level of flexibility within robotic software engineering.

Resumo

As aplicações de software robótico têm vindo a aumentar em complexidade, dificultando a gestão de atualizações de código com tempo de inatividade mínimo. A implantação destas aplicações também se tornou mais exigente, em especial à medida que ambientes distribuídos ganham popularidade ao transferir tarefas computacionais para recursos em ambientes *cloud* ou *edge*.

Embora plataformas de orquestração de software como o Kubernetes tenham melhorado a gestão de software noutros domínios, a sua utilização na robótica continua limitada, levando desenvolvedores a recorrer frequentemente a processos manuais ou a *scripts* específicos a cada aplicação. Esta dissertação aborda essa desfasagem, apresentando um módulo de orquestração para a framework Rigel, que automatiza a conteinerização de aplicações ROS, permite atualizações contínuas (zero-downtime) e inclui funcionalidades extensíveis para descarregar cargas de trabalho intensivas para máquinas externas.

No desenvolvimento deste módulo, o trabalho adapta o Kubernetes aos requisitos robóticos através de verificações de prontidão (*readiness checks*) orientadas pela aplicação, impedindo que as atualizações interrompam tarefas em curso, ao mesmo tempo que abstrai grande parte da configuração de rede complexa necessária para a comunicação entre nós ROS. Uma interface declarativa no ficheiro de configuração do Rigel garante que administradores de sistema e programadores apenas necessitam de especificar parâmetros de alto nível, cabendo ao módulo gerir o agendamento de contentores, a configuração do ambiente e a persistência de dados, mantendo-se ainda flexível para acomodar lógica de orquestração personalizada quando requerido.

Uma *stack* de observabilidade integrada, combinando Prometheus, Grafana e Loki, oferece dados de desempenho e registo em tempo real com definições pré-configuradas adequadas, ajudando os programadores a diagnosticar estrangulamentos de recursos ou anomalias de software à medida que surgem, além de facilitar a integração de métricas personalizadas da aplicação num dashboard Grafana unificado. Os testes de validação incidem em padrões de comunicação ROS comuns para serviços, ações e tópicos de publicação-subscrição, demonstrando que mesmo fluxos de dados de alta frequência se mantêm estáveis durante as atualizações de software em execução.

Resultados experimentais, realizados num ambiente com recursos limitados, comprovam a execução de atualizações de software sem paragens, a perda mínima de mensagens e um desempenho fiável mesmo sob testes de esforço. Estes resultados indicam que as abordagens cloud-native podem estender-se à robótica, eliminando a sobrecarga manual ainda presente em equipas que necessitam de implementar e escalar software para robôs. Em consequência, o módulo de orquestração proposto permite aos programadores concentrarem-se no desenvolvimento de lógica de aplicação, em vez de se preocuparem com detalhes de infraestrutura, proporcionando assim um novo nível de flexibilidade na engenharia de software robótico.

Acknowledgments

Thank you to my advisors, Dr. Armando Sousa and Dr. Rafael Arrais, for their guidance, motivation, and support throughout this dissertation project.

Thank you Pedro Melo for developing the original Rigel framework and guiding me through the implementation of the orchestration plugin.

"Gratitude is not only the greatest of virtues but the parent of all others." - Cicero

Nicholas Hopf

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) provide a global framework to address humanity's most pressing challenges. This dissertation contributes to the following SDGs through its technical innovations in robot software orchestration:

SDG 9: Industry, Innovation, and Infrastructure

Target 9.1: Develop quality, reliable, sustainable, and resilient infrastructure.
Contribution: The orchestration module enhances infrastructure reliability for robotic systems through zero-downtime updates and distributed deployments. By enabling offloading of computational tasks to cloud/edge resources, it reduces reliance on power-intensive onboard hardware, extending the operational lifespan of resource-constrained robots.
Target 9.5: Enhance scientific research and upgrade industrial technologies.
Contribution: The integration of Kubernetes with ROS democratizes access to advanced orchestration capabilities for robotics researchers. The declarative interface lowers barriers to adopting cloud-native patterns, accelerating innovation in industrial robotics and multi-robot systems.
Performance Indicators:

- Reduction in deployment downtime during software updates (validated at 99.99% communication integrity in high-frequency scenarios)
- Potentally increased resource utilization efficiency through dynamic workload distribution, enabled by orchestration technology
- Adoption rate of orchestration patterns in academic/industrial robotic projects

SDG 12: Responsible Consumption and Production

Target 12.2: Sustainable management and efficient use of natural resources.
Contribution: By optimizing resource allocation for software, the plugin minimizes computational waste in robotic deployments. The observability stack enables teams to identify and eliminate resource bottlenecks, promoting energy-efficient operations.
Performance Indicators:

- Reduction in idle compute resources through automated scaling
- Decreased hardware refresh cycles via cloud offloading capabilities
- Carbon footprint metrics from cloud providers' sustainability dashboards[1]

[1] https://aws.amazon.com/aws-cost-management/aws-customer-carbon-footprint-tool/

The provided support for distributed deployments aligns with **SDG 13 (Climate Action)** by facilitating integration with sustainable cloud platforms. Major providers like AWS[2] and Google Cloud[3] now operate carbon-neutral data centers, enabling roboticists to reduce environmental impact through strategic workload placement.

Table 1: SDG Contributions and Measurement Framework

SDG	**Target**	**Contribution**	**Performance Indicators**
9	9.1	Resilient infrastructure through zero-downtime updates	Deployment success rate during updates
9	9.5	Accelerated innovation via Kubernetes abstraction	Adoption in research papers/industrial projects
12	12.2	Resource-efficient deployments through autoscaling	CPU/Memory utilization metrics
13	13.3	Cloud integration for carbon-aware computing	% workloads deployed to sustainable clouds

[2]https://aws.amazon.com/sustainability/
[3]https://cloud.google.com/sustainability

Contents

List of Figures

List of Tables

Abbreviations and Symbols

AI	Artificial Intelligence
API	Application Programming Interface
AWS	Amazon Web Services
CI	Continuous Integration
CD	Continuous Delivery
CLI	Command Line Interface
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
CRIIS	INESC TEC's Center for Robotics in Industry and Intelligent Systems
DDS	Data Distribution Service
DevOps	Development Operations
E2E	End-to-End
EC2	Amazon Elastic Compute Cloud
EKS	Elastic Kubernetes Services
EOL	End of Life
GCP	Google Cloud Platform
GiB	Gibibyte (2^{30} bytes)
GKE	Google Kubernetes Engine
IaaS	Infrastructure as a Service
INESC TEC	Institute for Systems and Computer Engineering, Technology and Science
IP	Internet Protocol
K8s	Kubernetes
LTS	Long-Term Support
MRS	Multi-Robot Systems
NPS	Net Promoter Score
OCI	Open Container Initiative
OSRF	Open Source Robotics Foundation
PaaS	Platform as a Service
PEP	Python Enhancement Proposal
Pub/Sub	Publish/Subscribe
PV	Persistent Volume
QoS	Quality of Service
RAM	Random Access Memory
ROS	Robot Operating System
SaaS	Software as a Service
SoTA	State of the Art
SDG	Sustainable Development Goal
SLA	Service Level Agreement
UML	Unified Modeling Language
VM	Virtual Machine

Chapter 1

Introduction

Due to the increasing level of autonomy and complexity in modern robotic systems, the management of robot software also becomes more complex. This trend is driven by the integration of multi-domain software applications in robotics, encompassing components such as AI (artificial intelligence), cognition, and human-robot/robot-robot interaction [1]. With the concurrent advancements in sensor devices, there is a substantial rise in the volume and diversity of information that must be processed by robots' onboard processors [2].

Handling all computation and storage processes exclusively on board poses the risk of significant computational burden and inefficiency. To address such problems, the concept of *cloud robotics* was introduced in the last decade, involving the delegation of storage and computation tasks to remote servers [2, 3], as to take advantage of heterogeneous (Edge-Fog-Cloud) architectures [1]. Several robot applications already adopt this approach, where the middleware plays a crucial role in facilitating information sharing among robots, adding another layer of complexity to deployment infrastructure [2].

In response to the growing complexity of deployment environments for robotic software, *containerization* and *orchestration* technologies can offer a more standardized and automated approach to managing distributed robotic applications. These techniques facilitate communication across different network levels and promote modularity in the utilization of software components [2]. Based on the established practices of *cloud orchestration*, *robot software orchestration* involves coordinating, at both the software and hardware layers, the deployment of a set of virtualized services to achieve operational and quality objectives [4]. Consequently, *robot software orchestration* becomes crucial in addressing the complexities inherent in modern robotic systems, ensuring their efficient operation across diverse environments [1, 2].

While advanced orchestration and management tools such as Kubernetes[1] are broadly adopted in web and cloud computing domains, they are not yet prevalent in robotics [5, 6]. Developers often resort to ad hoc or manual approaches for provisioning, updating, and scaling robotic software, particularly when containerized robotic applications must operate both locally and distributedly. [1, 5]. This gap diminishes the operational efficiency, reliability, and scalability of robotic

[1] https://kubernetes.io/

systems. Furthermore, existing solutions for container-based robotics, such as official Docker images from the Robot Operating System (ROS) community, focus mainly on local deployments and do not readily integrate with widely available orchestration platforms. Consequently, this dissertation addresses the need for a more standardized, automation-focused methodology to orchestrate distributed robotic applications.

1.1 Motivation and Starting Points

Although the advantages of orchestration technologies for software applications are widely acknowledged, most robot software development tools do not provide integration with orchestration technologies in the deployment phase [1]. In contrast, containerization technologies are already generally available for robot software development, primarily driven by the ROS community.

The adoption of containers is a reliable alternative to traditional deployment methods employed by developers for distributing their ROS applications. Notably, containers ease developers' concerns related to portability and external dependencies, offering a solution without compromising significant performance losses [7]. The Open Source Robotics Foundation (OSRF) maintains official ROS Docker images[2] that are widely used by robot software developers. Additionally, Docker images of commonly used development tools, including Gazebo[3], a robot simulator for ROS, are also publicly available [7].

However, despite these advancements, there remains a gap in establishing a standardized procedure and tools for creating, testing, and deploying containerized ROS packages. This leads to the development of Rigel[4], a container-based framework developed by INESC TEC's Center for Robotics in Industry and Intelligent Systems (CRIIS) that assists the development of ROS applications throughout all stages of development [7].

Rigel addresses this gap by allowing developers to containerize and deploy ROS packages, enabling runtime testing of containerized ROS applications within simulation environments. It automates the creation of Docker images for both individual ROS packages and full workspaces, allowing flexible container configuration [7]. Despite these capabilities, Rigel currently lacks support for integration with container orchestration tools.

1.2 Research Questions and Objectives

Since Rigel is modular and extendable, this dissertation focuses on developing an orchestration module for the framework. Therefore, this research targets the integration of container orchestration tools, such as Docker Swarm[5], Hashicorp Nomad[6], Apache Mesos[7], and Kubernetes[8], con-

[2] https://hub.docker.com/_/ros
[3] https://hub.docker.com/_/gazebo/
[4] https://github.com/rigel-ros/rigel
[5] https://docs.docker.com/engine/swarm/
[6] https://www.hashicorp.com/en/products/nomad
[7] https://mesos.apache.org/
[8] https://kubernetes.io/

sidering their respective advantages and compatibility with ROS-developed robot software. The primary research goal is to contribute to the development of resilient robot software deployments, less prone to unavailability and more real-time capable, while promoting innovation in orchestration methodologies for robotic applications. With an orchestration tool integrated into Rigel, robot software developers will be able to streamline the implementation of distributed deployments, rolling updates (with no service downtime), among other features described in Section 2.1.4.

Additionally, this research opens opportunities for future work, especially to investigate the feasibility of integrating robotic applications with cloud-native platforms and services – namely Platform as a Service (PaaS), Infrastructure as a Service (IaaS), and Software as a Service (SaaS) – offered by major providers such as Google Cloud Platform (GCP) and Amazon Web Services (AWS).

Therefore, to guide the research, the following questions are posed:

1. RQ1: How can modern orchestration technologies be adapted to meet the real-time and hardware-specific constraints of typical ROS-based robotic applications?
2. RQ2: What methodologies are required to enable zero-downtime rolling updates, improve observability, and support distributed deployments while minimizing the learning curve for robotics developers?
3. RQ3: How can the developed Rigel orchestration plugin remain flexible for robot software developers to define custom application management logic while simplifying their decisions?

Based on those research questions, the consolidated list of objectives is presented below:

Objectives

1. O1: Simplify Kubernetes rolling updates by abstracting their functionality, enabling developers to implement continuous software delivery while minimizing disruption to robotic operations, using readiness checks to block updates or terminations until the robot is ready.
2. O2: Offer flexible deployment configurations that default to offloading workloads to external servers for scalability, while still supporting local on-robot deployments and persistent volumes to address common performance and reliability concerns in robot software.
3. O3: Validate the module through practical case studies, assessing its performance, scalability, and user-friendliness, using observability tools with pre-configured dashboards and metrics.
4. O4: Implement a plugin that integrates Kubernetes features into the Rigel workflow, ensuring that the other objectives are fulfilled.

1.3 Scope and Contributions

This dissertation contributes to the field of robot software orchestration by developing a Kubernetes-based orchestration module for the Rigel framework. By using the module, developers can automate deployment and management tasks, ensuring efficient operation and scalability of robotic applications. The research will provide insights into the integration of orchestration tools in robotics, addressing existing gaps and promoting the adoption of standardized practices.

The scope of this research is primarily centered on integrating Kubernetes-based orchestration into Rigel. While Docker Swarm or other orchestrators could be explored, Kubernetes is chosen for its community support, ecosystem maturity, and wide adoption in industry. The contributions of this dissertation include:

1. A dedicated orchestration Rigel plugin that automates tasks such as scheduling, scaling, updating, and monitoring ROS-based robotic containers.

2. An extension to Rigel's `sequence` abstraction, providing a standard method for managing multiple jobs (e.g., deployment, monitoring, readiness checks).

3. Automated provisioning of observability tools (Prometheus[9], Grafana[10], Promtail[11], Loki[12]) pre-configured for robotic applications, offering a setup for real-time monitoring of performance, logs, and lifecycle events.

4. Implementation of best practices for zero-downtime updates, including transparent rolling updates, specialized readiness gates, and failure handling that preserve ongoing robotic tasks to ensure service reliability.

5. The module enables offloading heavy computations to cloud or edge servers, while supporting local execution when needed. It manages persistent storage with Kubernetes Persistent Volumes and, by default, configures networks to ensure proper communication between ROS nodes in different pods.

This work focuses on ROS 1 application development, which is still prevalent in the robotics community, according to recent surveys [8, 9]. This decision was based on the compatibility of the Rigel framework, which only supported ROS 1 when the research began, as ROS 2 was not yet mature enough to justify a full transition. ROS 2 includes several features that address ROS 1 limitations, particularly in supporting real-time and distributed systems – exactly the type of applications that can benefit most from this dissertation work. An important change in ROS 2 is the removal of a central node, ROS Master, which is no longer necessary [10]. This is because ROS 2 introduced the Data Distribution Service (DDS) as a communication layer between processes, eliminating the need for a ROS Master node, which facilitates distributed deployments and deals

[9] https://prometheus.io/
[10] https://grafana.com/
[11] https://grafana.com/docs/loki/latest/send-data/promtail/
[12] https://grafana.com/loki

with several networking concerns by default [9]. The Rigel orchestration plugin remains relevant despite ROS 2 innovations, as it improves the developer experience of orchestrating software while supporting ROS 1 in production environments. Since long-term support for ROS 1 ends in May 2025, future work should focus on adapting the orchestration plugin to ROS 2, while also leveraging its new features and capabilities.

Overall, this dissertation demonstrates how container orchestration solutions can be adapted for robotics to reduce complexity for developers, improve reliability of deployments, and leverage the scalability of modern cloud-native systems.

1.4 Document Structure

The remainder of this document is organized as follows:

Chapter 2 presents the theoretical background relevant to robot software development, developer operations, software virtualization, and cloud infrastructure, while also describing the current state-of-the-art in robot software orchestration. It discusses pertinent surveys and case studies found during the literature review, with an analysis of different orchestration proposals.

Chapter 3 details the methodology employed in this research, encompassing the software design strategies, the incremental development approach, and the experimental setups used to validate the proposed module. It also discusses how the research questions are addressed throughout the development lifecycle.

Chapter 4 presents the implementation details of the Kubernetes-based orchestration plugin for Rigel, followed by validation experiments. Various test cases, including Continuous Integration (CI) scenarios and robotic deployments, are analyzed to assess performance, reliability, and user experience.

Finally, Chapter 5 summarizes the contributions, reiterates how the results answer the research questions, and mentions potential approaches for future work.

Chapter 2

Background and State of The Art

This chapter establishes conceptual foundations for robot software development and management, covering *developer operations*, *software virtualization*, and *cloud infrastructure*, alongside a review of the state of the art in these areas. The goal is to demonstrate why orchestration solutions are valuable for modern robotic software, as introduced in Chapter 1.

Section 2.1 provides a background of important technical concepts that support this dissertation. It starts with an overview of *Middleware for Robotics* (Subsection 2.1.1) to explain the motivation behind platforms like ROS. It then discusses *Developer Operations in Robotics* (Subsection 2.1.2), emphasizing the adaptations needed for robotic pipelines. Next, *Containerization* (Subsection 2.1.3) explains how container technology and Docker are used in robotic software workflows. Finally, Subsection 2.1.4 introduces the fundamentals of container orchestration tools.

Section 2.2 is divided in equivalent subsections, presenting the *State of The Art* in each of these areas, identifying research, tools, and case studies that relate to the requirements for orchestration in robotics.

2.1 Background

2.1.1 Middleware for Robotics

Middleware in robotics provides standardized mechanisms for communication, data exchange, and resource sharing across heterogeneous hardware and software components. The need for middleware arises from the complexity of integrating sensors, actuators, and algorithms from diverse sources, often requiring custom interfaces and protocols. Historically, frameworks like CARMEN [11] employed centralized architectures, where a single server managed inter-process communication. While effective in homogeneous networks, such designs struggled in distributed environments with mixed computational resources and latency-sensitive links (e.g., wireless bridges between onboard and offboard systems) [12]. Early alternatives, such as the Mission-Oriented Operating Suite (MOOS) for marine robotics [13] and the real-time-focused Orocos framework [14], addressed niche requirements but lacked flexibility for large-scale integration.

Historically, researchers and developers have recognized that standardizing communication protocols and interfaces saves significant effort when scaling or reusing software components. The concept of middleware, therefore, is tied to enabling interoperability and modularity, which are fundamental in large-scale or complex robotic deployments [9, 12].

Robot Operating System (ROS) stands out as the de facto standard middleware in the robotics community. It began as an academic project when developing large-scale service robots as part of the STAIR project at Stanford University and the Personal Robots Program at Willow Garage [12]. It then evolved into a globally adopted framework, with both commercial and research applications. ROS adopts a distributed peer-to-peer model, where functionalities are divided into nodes that communicate via publish-subscribe mechanisms for data topics [12]. As a result, developers can compose complex systems by connecting modular nodes that handle sensor data, transform coordinate frames, plan trajectories, and execute control loops. ROS has been increasingly popular because it facilitates prototyping and has a wide ecosystem of open-source packages. Nevertheless, ROS's modularity also introduces complexity in production deployments, as nodes must be consistently configured across machines or containers to preserve communication.

Overall, robotics middleware has laid the groundwork for many of the advanced behaviors seen in autonomous systems today, but it does not, on its own, resolve the challenges of deploying software consistently. As robots become more autonomous and the complexity of software increases, the entire development life cycle needs to incorporate modern practices that mitigate the complexities of testing, deploying, and maintaining distributed systems.

2.1.2 Developer Operations in Robotics

Developer Operations (DevOps) comprises a set of practices and cultural philosophies aimed at shortening the development cycle while delivering features, fixes, and updates frequently and reliably. DevOps emphasizes close collaboration between development teams and operations personnel, automation of repetitive tasks, and continuous feedback loops that inform decisions about deployment and maintenance [15, 16, 17]. Such principles have radically changed the software industry by integrating build, test, and deployment pipelines into a cohesive process.

In robotics, DevOps faces unique challenges due to the close coupling of software with physical hardware. As Cyber-Physical Systems (CPS), robots are expected to sense and actuate on real physical environments, adding an extra layer of complexity to software validation and deployment. Additionally, sensor data and actuation events must often be processed with strict latency requirements. The complexity of verifying correct real-world behavior becomes an obstacle to the kind of rapid, frequent releases that DevOps promotes. Furthermore, robotic systems featuring multiple robots or fleets amplify these issues by adding a distributed dimension to the code and data flows. Nevertheless, recent studies [18] show that the robotics industry is increasingly adopting DevOps principles, motivated by the need for faster prototyping, continuous testing in both simulation and hardware environments, and more robust post-deployment monitoring mechanisms.

One of the fundamental aspects of DevOps in robotics is the need for CI with hardware-in-the-loop testing. Even with robust simulation environments such as Gazebo[1] or Webots[2], real sensor data can differ in subtle yet significant ways from simulated inputs. Consequently, some teams have established dedicated lab environments where a CI server can deploy new builds to actual robotic hardware, run test scripts, and gather performance metrics [18]. This approach reveals hardware-related regressions early in the development cycle. The result is greater confidence that new code will behave reliably in production or in field operations.

Closely associated with CI is continuous delivery (CD), which automates the packaging and release of applications to the final execution environment. In a typical DevOps pipeline, once a merge request is approved and the tests pass, the software is packaged into containers, archived in a repository, and then automatically deployed to either cloud instances, local edge servers, or on-robot containers. The success of these pipelines relies on consistent environments and well-defined interfaces. In robotics, containerization plays a central role, as it enables developers to encapsulate not just the application code but also the various libraries and configurations needed for real-time operation in a portable manner. Thus, DevOps and containerization are interdependent in modern robotic software development workflows.

Ultimately, the main benefit of DevOps in robotics is an increase in deployment agility, allowing teams to roll out features or bug fixes much faster, a significant competitive advantage in domains like autonomous driving or industrial automation, where time-to-market is critical. Nevertheless, DevOps alone does not fully address the need to manage containers across multiple nodes and networks, which is why container orchestration tools have begun to gain traction in advanced robotics projects.

2.1.3 Containerization

Software virtualization technology is crucial for several modern development infrastructures. The delivery of virtualized applications allows for developers to have flexibility and control of its runtime environment, resulting in the abstraction of the infrastructure that will host the application. Virtual machines (VMs) are virtualization solutions that encapsulate the entire software stack, including the operating system, kernel, binaries, libraries, and applications [19]. To manage and allocate the physical hardware resources to execute multiple VMs in the host machine, VMs rely on a virtualization software known as the hypervisor. The resource-intensive nature of VMs presents challenges for portability and backup as the number of VMs on a server increases. To address these limitations, containers, a form of lightweight operating system virtualization, have been introduced as an alternative to traditional VM-based virtualization [19].

The fundamental difference between a VM and a container is that containers share the same kernel as the host machine, allowing for resource efficiency while maintaining isolation of processes and runtime environments. The aforementioned architecture difference is illustrated in

[1] https://gazebosim.org
[2] https://cyberbotics.com
[3] https://docs.docker.com/get-started/docker-overview/

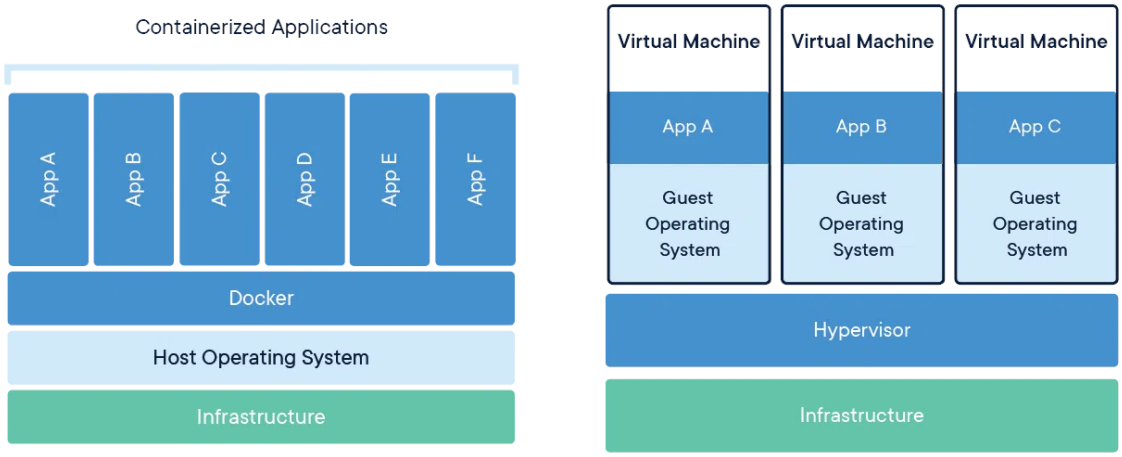

Figure 2.1: Architecture comparison between Docker containers and virtual machines[3]

Figure 2.1. The Docker project popularized the use of containers in the last decade with the introduction of a declarative standard for container configuration and a user-friendly interface. Docker enabled the rise of microservice software architectures, and it facilitates the collaboration between developers by sharing a unified deployment process, which is widely employed in CI/CD pipelines [20].

Docker employs a client/server architecture. The Docker daemon `dockerd` acts as a server, processing container builds and executing containers. Meanwhile, the Docker client serves as the primary interface to provision and interact with containers by communicating with the server through a user-friendly API. Communication between the `docker` command-line tool (client) and the `dockerd` daemon (server) occurs over Unix sockets and network ports.

Docker containers are created from container images. An image is an immutable binary package that includes all files required to run a program, namely: code, runtime, system tools, system libraries and settings. Docker's image format, now standardized by the Open Container Initiative (OCI), utilizes a series of filesystem layers that dynamically modify files during its packaging and execution phases. This layering concept enables granular changes, efficient storage, and version control [21]. The creation of Docker container images is enabled by Dockerfiles, which are text documents that specify the base image, the environment variables, and the shell commands necessary to build the image. After being created, Docker images can be published to an image registry, such as DockerHub[4] or GitHub Container Registry[5], so that other developers can download the image.

Container runtimes are the software components responsible for the execution of containerized applications. They translate container images into running instances, managing their lifecycle on the host system. The choice of runtime can influence performance, compatibility, and integration with orchestration tools. Some container runtimes include:

[4] https://hub.docker.com
[5] https://ghcr.io

- runC: Developed by the Open Container Initiative (OCI), runC is a lightweight, low-level runtime that uses Linux kernel features like namespaces and cgroups to isolate containers.
- containerd: A CNCF-maintained runtime designed for high-performance container management. Docker uses containerd as its default runtime, which, in turn, relies on runC for low-level container execution.
- cri-o: A Kubernetes-specific runtime optimized for container orchestration in Kubernetes clusters.

These runtimes enable consistent behavior across environments, but their capabilities differ in terms of features such as security, performance, and orchestration integration. For robotics, the selection of a runtime may depend on specific workload requirements, such as GPU support for machine learning tasks or compatibility with Kubernetes.

The adoption of Docker, in particular, is widespread in the robot software development community, which benefits from ROS Docker images made available by the OSRF, alongside Docker images of the Gazebo robot simulation platform and other relevant programs [7]. Containerization is particularly useful in robotics to manage software dependencies across diverse hardware platforms. Without containers, installing a robotic stack often involves configuring libraries, environment variables, and sensor communication bindings, leading to inconsistencies across deployments, which also become more time-consuming as software increases in complexity. Containers solve this problem by encapsulating the entire runtime environment, ensuring reproducibility and simplifying deployment.

Robotic applications are inherently modular, comprising multiple components such as sensor drivers, control algorithms, and user interfaces. Containers map naturally to this architecture, allowing developers to package each major component or ROS node into its own container. This modularity reduces dependency conflicts and facilitates partial updates, where only specific components are replaced without affecting the entire system.

The ROS community has adopted containerization, providing official Docker images for active ROS distributions. These images include pre-configured environments that reduce the effort needed to set up development and runtime systems [7]. Similarly, the Gazebo simulation platform offers Docker images that enable containerized testing of robotic applications. These resources streamline workflows for both developers and researchers, allowing them to focus on application logic rather than system configuration.

While containerization ensures portability and consistency, it also introduces new challenges, particularly in distributed robotic systems. Robotic applications often require containers to communicate across multiple hosts, including on-robot systems, edge servers, and cloud infrastructure. This creates complexities in networking and resource allocation.

Networking is a particular concern in containerized robotics, as ROS nodes rely on inter-process communication, which can be disrupted in containerized environments. ROS 1, for instance, uses a central ROS Master node to manage topic communication, which becomes a concern in multi-host deployments. Workarounds such as custom overlay networks or VPNs are often required to maintain stable communication between containers.

Persistent storage is another relevant challenge. Robotic systems can generate large volumes of data, such as sensor logs, maps, and calibration parameters, which must be preserved across container restarts or updates. Volume mounts allow containers to access shared storage, but managing these volumes across distributed environments requires proper execution.

In applications that require the simultaneous execution of multiple programs, the Docker Compose command serves as an alternative to orchestrate the execution behavior of container groups. While a Dockerfile specifies how a container is built, a `compose.yaml` file – historically referred to as `docker-compose.yaml` – defines how a service (i.e., a group of containers) should be run. The `compose.yaml` allows the declarative specification of running requirements, enables the coordination of networking, storage volumes, runtime variables, among other attributes [20]. While Docker Compose streamlines the orchestration of services in a user-friendly and concise way, it still presents management limitations. Since Docker Compose is only a wrapper for the standard Docker API, it does not introduce additional functionality. There are specialized orchestration solutions that allow advanced container management not provided by Compose, including the execution of health checks in production, rolling updates – which is the replacement of containers with no downtime – and coordination of multiple host machines.

2.1.4 Orchestration

In multi-node production systems, whether on-premise, fog, or cloud servers, the management of containerized applications involves complex operations. Container orchestration tools, such as Docker Swarm, Kubernetes, Apache Mesos, Amazon Elastic Container Service (ECS), Hashicorp Nomad, offer different approaches that allow IT administrators to automate host provisioning, instance execution, and container management [19]. They facilitate automatic resource allocation, reducing costs, and mitigating Service Level Agreement (SLA) violations. For instance, an orchestration tool should enable the scaling of services to accommodate for the current usage load. There are two types of scaling: horizontal scaling involves adding more nodes to a system, distributing the workload across multiple servers, and enhancing overall system capacity. Vertical scaling, on the other hand, involves increasing the resources (CPU, memory) allocated to an individual node to handle increased loads. Kubernetes, with its distributed scheduler, can perform advanced horizontal scaling routines, dynamically allocating and managing containers across a cluster of servers to meet varying demands. While both Kubernetes and Docker Swarm support vertical scaling, Docker Swarm has limitations in horizontal scaling tasks, which are more common in cloud environments [20].

While Docker Swarm is an orchestration solution integrated to the Docker engine, Kubernetes is an orchestration tool developed by Google to address more complex orchestration tasks [22].

Due to the widespread adoption of Kubernetes over Docker Swarm, Docker is increasingly integrating Kubernetes into its tooling, potentially diminishing the role of Docker Swarm as a preferred orchestration path [20].

Kubernetes introduced the concept of a *pod*, which is the foundational unit of deployment. A pod encompasses one or more interconnected containers that share common resources and network namespaces. By grouping related containers, pods facilitate cooperative operations, optimizing resource allocation and administration [19].

A Kubernetes cluster comprises two essential components: the *control plane* (master node) and the *data plane* (worker node). The control plane hosts components responsible for managing the Kubernetes cluster, while the worker nodes host the pods running one or more containers [19]. The control plane's essential components are listed in Table 2.1.

Table 2.1: Main components of the Kubernetes control plane [21]

Component	**Description**
Kube-API Server	Provides access to the Kubernetes control plane via an API, handling both external and internal requests.
Kube-Scheduler	Facilitates pod scheduling on specific nodes based on automated workflows and user-defined conditions.
Kube Controller Manager	Monitors and controls the state of the cluster, managing controllers that automate varied tasks at the cluster or pod level.
etcd	A fault-tolerant, distributed key–value database serving as a centralized repository for storing and managing configuration information and cluster status.

The data plane is also responsible for managing and routing network traffic within the cluster. The container runtime engine is a data plane component that executes and manages containers, while Kubelets are components that allow for communication with the control plane, managing the lifecycle of containers on each node [19]. Figure 2.2 represents a concise diagram with the aforementioned Kubernetes components.

While the use of Kubernetes as an orchestration layer provides the previously mentioned benefits, hosting a Kubernetes cluster adds complexity to the system [21]. This complexity drives the adoption of IaaS solutions, such as Amazon's Elastic Kubernetes Services (EKS) and Google Kubernetes Engine (GKE). Both solutions offer payment per usage and integration with continuous delivery pipelines, while abstracting hosting procedures and offering high availability. However, it is possible to use tools such as Minikube[6] and Kind[7] as developer-friendly deployable Kubernetes cluster solutions within a local virtual machine. Though limited to a single-node cluster, both Kind and Minikube are widely used platforms for local Kubernetes execution [21].

To interact with a Kubernetes cluster, users primarily utilize the official Kubernetes client, `kubectl`, a command-line tool that communicates with the Kubernetes API [21]. `kubectl` allows users to manage various Kubernetes objects, such as Pods, ReplicaSets, and Services, allowing the system administrator to explore and verify the overall health of the cluster. For example,

[6] https://github.com/kubernetes/minikube
[7] https://kind.sigs.k8s.io/

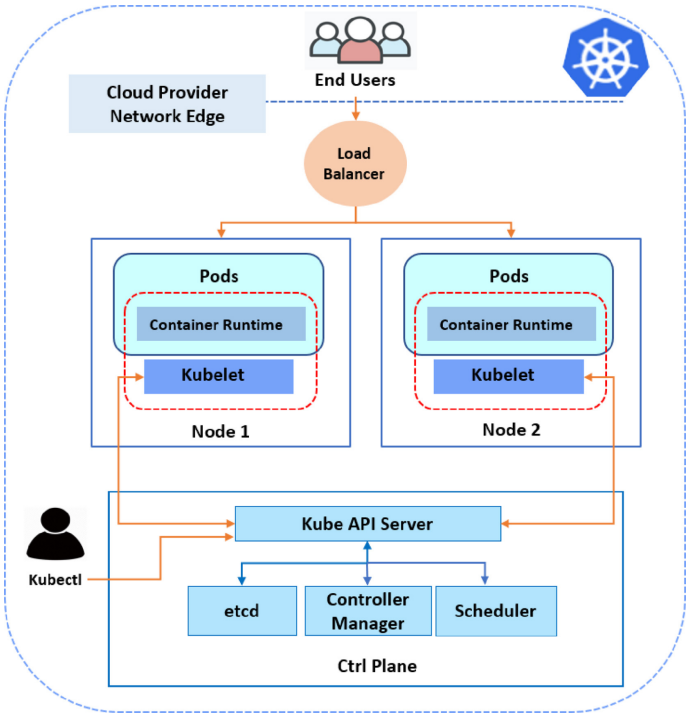

Figure 2.2: The structure of Kubernetes [19].

users can verify the cluster's version by running `kubectl version` and to verify the cluster's health diagnostics by running `kubectl get componentstatuses`. Additionally, users can list all nodes in the cluster with `kubectl get nodes`. This provides information on the nodes, differentiating between control-plane nodes containing essential components like the API server and scheduler, and worker nodes where user containers operate [21]. For users preferring a graphical user interface, most cloud providers offer infrastructure representation integrated into their platforms.

Container orchestration encompasses the technologies and tools that manage container lifecycles across multiple machines or clusters. By automating tasks like scheduling, scaling, monitoring, and rollbacks, an orchestrator ensures that the right containers are running in the right places, even in the face of failures or variable load.

In robotics, orchestrators can dramatically simplify the process of running distributed systems. For instance, when a robot's on-board system hits its CPU limit while processing sensor data, an orchestrator can spin up additional compute nodes in a cloud environment or an edge cluster, moving certain computation-intensive processes off the robot. This dynamic allocation relieves the robot's primary system, which can then focus on time-critical tasks such as motor control or collision avoidance. Similarly, orchestrators can detect failures in specific containers – like a perception node that has frozen or crashed – and automatically restart them, ensuring that the robot's overall functionality remains stable.

Despite these benefits, the use of orchestration in robotics introduces complexities not typical of web or enterprise use cases. A key difficulty is the real-time dimension. Many orchestrators are

designed for high availability but do not inherently provide deterministic scheduling guarantees. While some robotics applications can tolerate slight variations in timing, others require deterministic execution. This discrepancy means that not all robotic workloads are suitable for simple orchestration strategies, and specialized real-time extensions may be required. Another challenge arises from the network constraints of mobile robots, which may not always have a stable link to a cloud-based orchestrator. Thus, there is a preference for local or edge-based Kubernetes deployments, or simplified orchestration solutions that operate effectively within an unreliable network environment.

In any case, orchestration represents an important methodology for improving the reliability and maintainability of complex robotic applications. The subsequent sections examine the state of the art for these technologies in robotics and discuss how the proposed project – integrating Kubernetes orchestration into the Rigel framework – seeks to address existing gaps.

2.2 State of The Art

This section details several solutions recently developed for robot software development and management, expanding on the concepts introduced in the previous section.

2.2.1 Cloud Robotics Overview

The development of robotic systems is a domain currently undergoing significant research and development. These systems often incorporate local processing capabilities to ensure low-latency responses and handle network unavailability. Lately, there is an increase of robot software that is dependent on handling large datasets and the integration of machine learning algorithms for enhanced robot perception and decision-making. This increase of computational complexity challenges onboard robot computing units and storage devices, which may not be sufficient for most complex tasks [3].

In response, a new development tendency involves integrating cloud computing resources with robotic systems, which established a field commonly referred to as Cloud Robotics [23]. Cloud robotics and automation systems leverage distributed data and code to support a variety of applications, including networked remote operation, mobile robot groups, assembly lines, processing plants, home automation, and human computation systems [3]. Consequently, high-demand storage and computation tasks are delegated to remote servers [2], as to take advantage of large-scale cloud computing platforms, such as Amazon Elastic Compute Cloud, Google Compute Engine, and Microsoft Azure Virtual Machines.

Reliance on cloud servers may introduce new challenges, such as latency in time-sensitive robotics applications. Therefore, new distributed computational architectures are being proposed to address the efficacy of available computing resources. Besides fog and edge servers, computational resources can also be shared among a robot swarm through a mobile ad hoc cloud architecture, enabling the use of idle computing resources on-demand. This enables the deployment of

computing-intensive applications without relying on a cloud server or developed communication infrastructures [24].

In another research initiative, resource and knowledge sharing were implemented in a local robot cluster, resorting to Kubernetes to perform load balancing optimizations [25]. The authors compared the performance of two different AWS EC2 instances with the performance of a cluster composed by three Raspberry Pi machines with minikube as a Kubernetes deployment solution. While the local cluster had, in total, 4vCPUs and 8GB RAM available, one of the EC2 instances had 1GB RAM and 1vCPU, while the other had 4GB RAM and 2vCPUs. Each platform was tested with a YOLO-v3-based [26] image detection and recognition Flask application. The performance results are listed in Table 2.2, which demonstrates that the use of Kubernetes was successful to achieve dynamic load distribution and resource allocation among the robotic modules. Although the cluster hardware was notably more capable than the hardware in the EC2 instances, the case study still showcases how a robot system can benefit from the local distribution of computing resources. In particular, the local cluster demonstrated superior performance in terms of resource utilization and scalability, despite the EC2 instance having a faster response time in specific scenarios. This highlights both the potential of edge computing solutions to efficiently manage computational loads and optimize resource use in modular robotics, as well as the potential of delegating computational tasks, including time-sensitive applications, to cloud servers.

Table 2.2: Performance Comparison of a Cluster Architecture with Traditional Computation offloading [25].

Architecture	**CPU Usage (%)**	**RAM Usage (%)**	**Response time (s)**	**Network Throughput (Mbps)**	**Cores**	**Pods**	**Performance Metric**
t2.micro EC2 AWS	99.7	90.8	Timeout	70	1	1	-
t2.medium EC2 AWS	1.3	28.19	3.49	250.2	2	1	573.52
K3s Edge cluster	11	34	5.94	68.8	4	2	1120

While there are already several robotic systems utilizing Kubernetes as an orchestration solution [1, 25, 27, 5, 28], each author independently developed a distinct orchestration framework. In robot software development teams, this process can be time-consuming, and demands different technical skills from the developers. Thus, smaller development teams would benefit from a framework that enables the use of Kubernetes in a standard initial setup, adapted to robotic systems, allowing them to focus on the development of the robot software itself rather than setting up the orchestration solution.

2.2.2 Frameworks for Containerization of ROS Applications – Rigel

The lack of standard tooling for creating, testing, and deploying containerized ROS packages motivated the development of Rigel. Rigel is a container-based framework developed by INESC TEC's Center for Robotics in Industry and Intelligent Systems (CRIIS) that assists the development of ROS applications throughout all stages of development [7]. The features already implemented in

Rigel, displayed in Figure 2.3, are mainly accesible through a *Rigelfile*, which is a YAML configuration file that declaratively specifies the parameters used for containerizing, testing, and deploying ROS packages. For instance, a Rigelfile may contain the ROS distribution name, environment variables, simulation requirements and the deployment registry. Rigel also offers a command line interface (CLI) with dedicated commands for core functionalities, such as Rigelfile generation, external plugin installation, containerization, testing, and deployment of ROS packages [7]. With Rigel, developers can both use external container images or build their own without the need to directly edit a Dockerfile – although users can also provide their own Dockerfiles. Docker image generation involves a two-stage pipeline: first Rigel installs external package dependencies listed in a rosinstall file. Then the framework includes the system dependencies and builds the ROS workspace. Private repository cloning is also possible using SSH keys, which are securely handled by Rigel.

Simulation environments, listed in the Rigelfile's simulation section, can be run locally or remotely to test ROS packages. Rigel creates an abstraction layer for containerized ROS nodes to interact as if in a real-case scenario [7]. To assess the simulation results, Rigel subscribes to ROS topics at run-time based on the requirements declared in the Rigelfile through the HAROS Property Specification Language (HPL). Regarding the deployment phase, Rigel provides a mechanism to deploy containerized packages to remote Docker image repositories. The framework interfaces with the Docker daemon to tag the desired image and deploy it to the specified registry [7]. Rigel is designed to be extended through custom plugins, allowing developers to create, share, and integrate plugins with the main framework components. External plugins should comply with a simple programmatic interface, so they can be installed using the Rigel CLI. The tool simplifies plugin installation by wrapping pip, the official package installer for the Python programming language. Message logging and error handling mechanisms are built into Rigel to aid plugin development.

Although the development of the Rigel framework was guided by containerization principles, it currently lacks integration with container orchestration tools. The introduction of the orchestration module proposed within the scope of this dissertation would support the implementation of new features, such as executing concurrent simulations and deploying more complex container systems that implement load balancing, rolling updates, and telemetry.

The efficacy of the Rigel framework was initially validated through two distinct use cases [7]. The first solution was a ROS-based mobile manipulator with image acquisition and gripping hardware that arranged automotive and aerospace kits. During these projects, the researches developed a CI/CD pipeline encompassing Docker image generation and deployment, cloud-based simulations using AWS RoboMaker, and simulation assessment to ensure the solution's expected behavior [7]. The second use case was an autonomous virtual maritime robot that would serve as an illustrative contestant to the Virtual RobotX competition[8]. If the contestants had access to the framework, the teams would benefit from a simplified and faster submission process, since Rigel provides automatic container creation, deployment of the containerized solution to Docker Hub,

[8] https://robotx.org/programs/vrx-2023

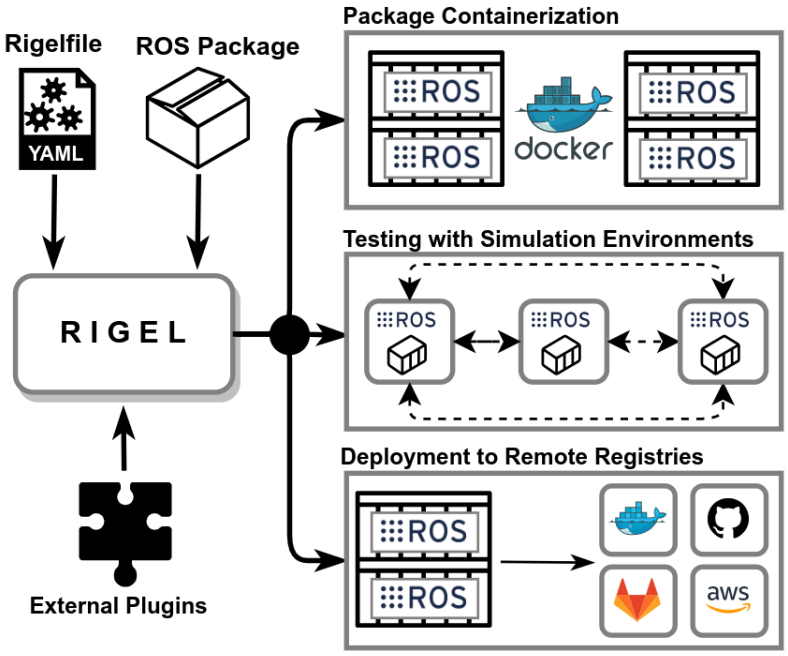

Figure 2.3: Map of the Rigel framework core capabilities [7].

and simulation success assessment in Gazebo. The framework was also successfully tested with the solutions developed by other contestants [7].

2.2.3 Middleware for Robotics – ROS

Robot Operating System (ROS) has become the dominant middleware solution for robotics, largely due to its extensive ecosystem and modular architecture. The available packages cover domains from navigation and perception to manipulation and multi-robot coordination. This ecosystem is buttressed by a large contributor base that frequently releases improvements, bug fixes, and new functionalities. ROS also benefits from a wealth of educational materials, ranging from official tutorials to third-party guides, which substantially lowers the learning curve for new roboticists.

Although ROS has proven effective in research environments, industrial deployments face additional constraints. Safety-critical systems demand rigorous testing, certification, and often hard real-time capabilities. In addressing these concerns, ROS 2 introduced a revamped architecture built on the Data Distribution Service (DDS) standard, offering built-in QoS options such as best-effort or reliable communication, and features like node-level security. Despite these improvements, setting up and maintaining a ROS 2 environment across distributed systems can still be labor-intensive, particularly if different ROS versions or distributions are used in parallel.

A number of industrial robots, including mobile manipulators and autonomous guided vehicles, demonstrate the viability of ROS in production [29, 30]. However, most of these systems rely on manual configuration processes or specialized infrastructure scripts. Operators may need to manually update each robot in a fleet, which can be error-prone and time-consuming. Containerization, as explored earlier, eases this burden by bundling code and libraries, but does not

inherently solve how these containers are deployed at scale. This is where orchestration enters, potentially simplifying the provisioning of entire fleets of robot containers, each guaranteed to have the correct software version and resource allocation.

The most up-to-date ROS Metrics Report, published in January 2024 corresponding to data collected up to the end of 2023, indicates that 57.88% ROS downloads are still ROS 1, while 42.12% are ROS 2.[9] This shows that ROS 1 is still widely used, and the transition to ROS 2 is still ongoing.

2.2.4 Developer Operations in Robotics

Developer Operations in robotics is still a recent field. Robot software developers frequently use automation scripts and continuous integration systems to optimize their experimental workflows, especially for repeated simulation runs. However, there is no single standard approach, and the diversity of hardware platforms complicates matters. A developer might implement a Jenkins pipeline that pulls the latest code, launches Docker builds, and then runs integration tests in Gazebo. Another developer might adopt GitHub Actions or GitLab CI, customizing specialized runners that interface with physical robots in a lab environment. Without a unifying framework, these solutions remain localized, each addressing only the specific team's needs.

Nevertheless, the robotics community has started to converge on certain best practices. One is the use of ephemeral simulation environments, typically in cloud or containerized settings, to run large sets of tests in parallel. This approach reveals regression failures early, especially beneficial for complex behaviors like navigation or manipulation tasks. Another emerging practice is to maintain separate "staging" and "production" environments for robotic applications. In a production environment, downtime can be costly and sometimes hazardous if robots are operating in public or industrial settings. A staging environment, by contrast, offers a sandbox where new features are tested without affecting ongoing operations. While this practice is commonplace in web services, it is only now gaining traction in robotics as better virtualization and orchestration solutions become available.

Overall, the assimilation of DevOps principles into robotics underscores the community's recognition that the field has become software-driven. The complexities of sensor fusion, AI-based planning, and distributed coordination can be more effectively managed by adopting practices that ensure consistent, automated, and traceable workflows.

2.2.5 Containerization

Containerization in robotics has seen particularly swift adoption over the past few years, as it aligns well with how robotic algorithms are developed and tested. The official ROS Docker images reduce the boilerplate setup necessary to initiate a consistent development environment, addressing a significant pain point in robotics education and early-stage prototyping. In more advanced settings, teams add layers of specialized libraries – such as GPU-enabled deep learning frameworks

[9] http://download.ros.org/downloads/metrics/metrics-report-2024-01.pdf

– on top of these base images, creating custom images that can run specialized computations on high-performance hardware.

Beyond local development, containerization streamlines integration testing in remote environments. A typical scenario might involve a developer pushing their updated Docker image to a container registry, which is then automatically pulled by a cluster of test servers to run both software- and hardware-in-the-loop tests. By using container tags and versioning, teams ensure that each test references a precise snapshot of the software environment, thereby minimizing the risk of environmental inconsistencies.

However, while containerization itself has become more widespread, orchestrating multiple containers across fleets of robots or distributed clusters remains a challenge. An organization may containerize each module effectively, only to rely on ad hoc approaches for launching and managing them on different platforms. This fosters a need for integrated orchestration solutions, particularly those that can handle robot-specific constraints such as real-time scheduling, sensor data flows, and network intermittencies.

2.2.6 Orchestration

Contemporary orchestration solutions for containerized applications, especially Kubernetes, hold significant promise for robotics. These platforms provide standardized resource descriptions, allowing operators to define how many replicas of a service should run, how they should be exposed via network interfaces, and how to handle failures. They also offer rolling updates, ensuring that when a new container version is deployed, the old version is gracefully shut down only after the new one is confirmed healthy.

In web applications, this approach has revolutionized uptime and reliability. In robotics, however, zero-downtime rolling updates might require additional logic to ensure that sensors remain operational or that robots do not abruptly switch control algorithms mid-task. Despite these complexities, pioneering projects have shown that Kubernetes can indeed orchestrate ROS-based systems, scaling them dynamically based on computational needs. One example is using edge nodes for local sensor processing and cloud nodes for machine learning inference, seamlessly shifting workloads as demands change.

Yet, these proofs of concept often rely on custom scripts, limited documentation, or advanced Kubernetes knowledge that is not common among robot engineers. Smaller teams, in particular, face a steep learning curve if they wish to benefit from automated scaling, resource management, and sophisticated networking features. Persistent storage – critical for storing sensor logs or map data – also poses a unique challenge, as robots might operate in constrained local networks that do not seamlessly tie into the typical storage layers used by Kubernetes in data centers.

Taken together, the state of the art shows a clear progression: from manual configuration of robotic middleware and DevOps pipelines to containerized deployments and, more recently, to partial or experimental use of orchestrators. What remains is a cohesive, user-friendly framework that integrates the advanced features of container orchestration into day-to-day robotic development processes. This dissertation aims to fill that gap by extending Rigel – a container-based

framework for ROS – with Kubernetes-based orchestration features that are robust yet accessible to developers with varying levels of orchestration expertise. The subsequent chapters will detail how these features are designed, implemented, and validated.

Chapter 3

Methodology and Approach

This chapter details the methodological steps taken to design, implement, and validate the Kubernetes-based orchestration module for the Rigel framework. Section 3.1 outlines the preliminary work, including the literature review and the analysis of typical robotic development workflows, which established the foundation for this research. Section 3.2 describes the requirement listing process, distinguishing between must-have and nice-to-have features. Sections 3.3 and 3.4 explain the iterative development approach and tools employed for quality assurance, and Section 3.4 presents an overview of the solution architecture, including how the new module integrates with Rigel. Section 3.5 details the validation strategy, test scenarios, and performance metrics for the orchestration module. Sections 3.6 and 3.7 respectively discuss the system use cases and the mechanisms to assess if an update can be safely applied to a running ROS application (readiness probes). Finally, Section 3.8 explains how the orchestration module is used in practice when integrating it into a CI/CD pipeline.

3.1 Preliminary Work

Before the implementation phase, several preparatory steps were taken to ensure the new orchestration module would meet the needs of robotic developers. The first step was researching and documenting the fundamental concepts involved in robotic software development, followed by a review of the state of the art, as documented in Chapter 2.

From this survey, three main gaps were identified to be addressed by the orchestration module:

1. While containerization is well-established for robotic applications, deployment automation for multi-node or distributed deployments remains ad hoc.

2. Zero-downtime rolling updates in ROS environments are not often implemented for robotics applications that would benefit from it, such as to ensure continuity of sensor data and ongoing tasks during software updates.

3. Lack of user-friendly orchestration tooling for robotic software developers, who may be unfamiliar with cloud-native technologies like Kubernetes.

In particular, since the purpose of the dissertation was developing an orchestration product for robotic software development, it was important to understand typical development pipelines for ROS-based applications.

A previous study [18] explored popular open source robotic software, listed in the official ROS Metrics report[1], in which the authors identified that robot software developer teams frequently rely on local simulated environments to test their application code, meaning that a substantial percentage of projects did not integrate an End-to-End (E2E) CI/CD pipeline, relying on local testing alone. This means each team member runs scripts or manual commands to simulate robotic software with tools like Gazebo, potentially leading to struggles with networking addresses or environment variables for bridging nodes together on reasonably complex projects.

Understanding these workflows was important to ensure that the proposed solution aligns with actual robotics development needs. In particular, a recurring theme was *minimizing the cognitive load* on robotics developers when it comes to infrastructure details, such that the proposed orchestration module should abstract away Kubernetes complexities and provide a simple interface for managing ROS deployments, still being flexible to extend it with advanced Kubernetes features for more experienced users or those with specific requirements.

The consolidated result of the preliminary work is the definition of objectives, mentioned in Chapter 1. After the initial research phase was completed, a general roadmap was developed to ensure all necessary development steps would be covered, from requirement gathering to testing and validation. That roadmap is shown in Figure 3.1, with each step corresponding to a specific phase in the project.

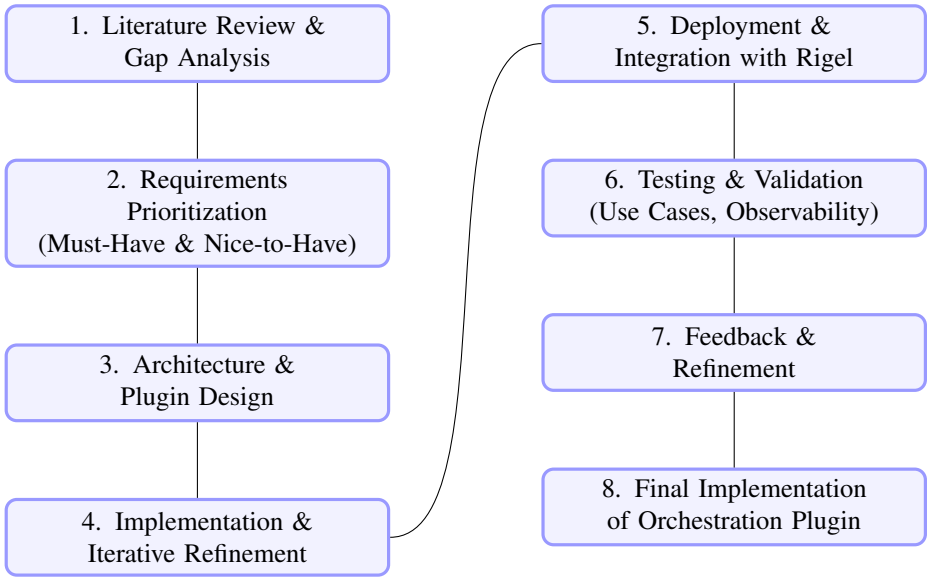

Figure 3.1: Overview of the roadmap for the dissertation project.

[1] https://metrics.ros.org/

3.2 Requirements listing

Based on the gaps identified and the workflow analysis, a set of features was prioritized. They were classified into *must-have* features, corresponding to the core functionality of the orchestration module, and *nice-to-have* features, which are second in priority but still can be integrated to the module by each Rigel user since the orchestration module was designed to be extendable.

Table 3.1: Must-Have Features

Feature	**Description**
Rolling Updates	Container updates without interrupting active ROS nodes, guaranteeing continuity of sensor data streams and robot tasks.
Distributed Deployments	Automated scheduling of ROS containers across local (on-robot) and remote (edge/cloud) nodes using Kubernetes primitives.
Persistent Storage	Support for Kubernetes Persistent Volumes to maintain critical data (e.g., sensor logs, maps) across restarts of stateful applications. This feature is crucial for distributed deployments.
Readiness Checks and Health Monitoring	Prevent container shutdown or updates until the robot node signals readiness.
Observability Stack	Basic but extensible dashboards using Prometheus and Grafana for metrics, Loki and Promtail for logs, configured with sensible defaults for all pods.

Table 3.2: Nice-to-Have Features

Feature	**Description**
Automatic Scaling	Dynamic addition or removal of container replicas in response to computational loads (e.g., spiking demands).
Security and Network Policies	Resource quotas, restricted traffic flows, and security contexts for containers running in untrusted networks. Communication policies, such as DNS resolution for ROS nodes, are considered crucial since they viabilize distributed deployments.
Fine-Grained Resource Management	CPU pinning, GPU pass-through, or real-time kernel extensions for latency-sensitive robotic tasks.

Although the *nice-to-have* features are not explicitly implemented as default behaviors in the plugin, the adopted software design ensures that the plugin remains extendable to accommodate them through custom Kubernetes parameters that can be dynamically defined in the new Rigelfile schema.

3.3 Selection of Tooling

Beyond algorithmic concerns, an essential part of the methodology involved choosing tools that enforce robust development practices, ensure compatibility with the Rigel framework, and align with standard software engineering principles.

Since Rigel is a Python-based framework, the orchestration plugin was also implemented in Python to maintain consistency and simplify integration with existing Rigel components. Choosing Python-specific tools for linting, type checking, and testing frameworks was guided by their popularity, proven integration with container-based CI/CD pipelines, and straightforward support for YAML configuration parsing.

Ruff[2] was chosen as a combined linter and formatter because it delivers quick feedback on coding style and potential logical issues in Python. Ruff enforces a strict set of rules that align with PEP8[3] and PEP257[4] guidelines, ensuring code consistency and readability, while also being faster to run than linters and formatters such as Flake8[5] and Black[6], respectively, and integrating with a pre-commit hook in this repository. Although Rigel already used Flake8, that replacement does not affect the compatibility with the existing codebase, as Ruff was configured to run a superset of Flake8's rules.

Mypy[7] has been incorporated for static type checking, consistent with Rigel, which provides a defense against subtle errors in orchestrator configuration. Its effectiveness depends on the habit of documenting code properly using type hints, which is enforced by Ruff's strict style checks, so when developing the plugin, developers would be enforced to write type hints and docstrings for all functions and classes, ensuring that the plugin is well-typed and less prone to runtime errors. Since the plugin interacts extensively with the Kubernetes API, typed definitions for pods, deployments, or resource fields significantly reduce confusion or runtime failures caused by type mismatches.

For the testing framework, Pytest[8] has a concise syntax that was sufficient to perform basic unit, integration, and E2E tests that validate the behavior of the plugin. Pytest's fixture system allows simulation of different Kubernetes states, letting developers create isolated or combined test scenarios that simulate multi-container deployments. Besides the fixture system, Pytest also

[2] https://docs.astral.sh/ruff/
[3] https://peps.python.org/pep-0008/
[4] https://peps.python.org/pep-0257/
[5] https://flake8.pycqa.org/en/latest/
[6] https://github.com/psf/black
[7] https://mypy-lang.org/
[8] https://pytest.org/

has a more ergonomic syntax for declaring tests and assertions, which motivated its use instead of the unittest module, which is the library chosen by the original Rigel developers.

Pydantic[9] is a data validation library integrated to the plugin to parse user-supplied Rigelfile fields and mapping them to Python data classes. This ensures that if a user configures an invalid readiness command or sets a wrong variable type, the plugin reports such errors early, rather than failing during the deployment process. Since the Rigel framework already uses Pydantic for its configuration schema, the plugin's integration with Pydantic also ensures consistency with the existing codebase, extending the same validation principles and classes to the new orchestration module.

Sphinx[10] was used to generate the technical documentation for the plugin, which is exported as HTML files and updated automatically with each commit, ensuring that the documentation is always up to date with the codebase. Rigel, originally, did not have any built-in documentation generation tool.

All development tools mentioned above are used in a pre-commit hook that runs before each Git commit, ensuring that the codebase remains consistent and error-free. A CI pipeline with *GitHub Actions* was also configured to run the same set of checks, plus an additional integration test, to validate the plugin's behavior in a controlled environment that gives feedback on the plugin's correctness of implementation. In other words, this setup covers integration tests on multiple Python versions, linting checks, validation of a simple E2E deployment scenario on a local Kubernetes cluster, that was both tested locally with a Minikube instance and consistently validated in the CI pipeline with a Kind instance. By choosing the aforementioned tools, the project follows established best practices in DevOps.

During early testing, Prometheus was integrated to assess how successful rolling updates were in maintaining uptime and service availability. Later on, Grafana was added to visualize the metrics collected by Prometheus, allowing monitoring the plugin's performance and if the ROS application's behavior was consistent with the expected results, as well as displaying resource usage data from the Kubernetes cluster. Loki and Promtail were also later integrated to aggregate logs from the plugin and the ROS application, providing a complete observability stack that was then repurposed and included in the plugin as a default feature, such that users can also monitor their ROS applications in real-time without having to set up the observability stack themselves. Prometheus, Grafana, Loki, and Promtail were selected to form the observability stack because they are widely adopted by the cloud-native community and provide well-documented integrations with Kubernetes. Since all these tools are open-source and have large support communities, they also align with Rigel's design philosophy of extensibility and developer-friendly operations.

[9] https://docs.pydantic.dev/latest/
[10] https://www.sphinx-doc.org/

3.4 Architecture and Integration with Rigel

The orchestration plugin extends Rigel's command-line interface and job system to manage Kubernetes deployments. As mentioned in Section 2.2.2, Rigel organizes complex tasks such as building images and pushing to registries, as *jobs*, which can be grouped into *sequences* [7]. Therefore, the orchestration plugin introduces new jobs and sequences that Rigel users can integrate to the CI/CD pipeline for their ROS projects, allowing them to perform orchestrated deployments and updates to their applications.

A new job called `deploy_k8s` was added to the Rigelfile schema, with sensible parameters that allow users to identify and modify orchestration-specific configurations as needed. An implementation of the `deploy_k8s` job schema is shown in Listing 1.

By default, the `deploy_k8s` job:

- Deploys a dedicated ROS Master pod (`ros-master-container`) to manage ROS 1 communication.
- Configures a readiness probe `/usr/local/bin/readiness_probe.sh` to ensure nodes are operational before updates.
- Provisions a 1 GiB persistent volume (`logs-volume`) for sensor data retention across restarts.
- Enables observability tools, with Prometheus scraping metrics and Loki aggregating logs.
- Implements a rolling update strategy (`maxUnavailable: 0, maxSurge: 1`) to prevent downtime during container replacements.

This declarative approach allows developers to focus on application logic while the plugin handles networking, resource allocation, and lifecycle management.

A full `demo` deployment sequence in a Rigelfile can be exemplified as follows:

```
sequences:
  demo:
    stages:
      -
        jobs: ["dockerfile", "build", "deploy_k8s"]
```

Which integrates the generation of a Dockerfile for the application, then building the container image and deploying it to a Kubernetes cluster with the new orchestration plugin, representing an E2E workflow for containerizing and deploying a ROS application.

The plugin abstracts networking details from the user, as will be described in Section 3.4.1, and the readiness state is always defined at runtime by the user, ensuring that application logic dictates when the ROS application is fully operational before the plugin proceeds with a deployment or update process. The implementation of such readiness checks is described in detail in Section 3.7. Persistent storage is also abstracted from the user, with the plugin automatically creating and

Listing 1 Example `deploy_k8s` job in a Rigelfile.

```yaml
deploy_k8s:
  plugin: "src.plugin.OrchestrationPlugin"
  with:
    orchestration:
      deploy_ros_master: true
      readiness:
        command: "/usr/local/bin/readiness_probe.sh"
      observability:
        enabled: true
      rolling_update:
        strategy: "Rolling"
        max_surge: 1
        max_unavailable: 0
      distributed:
        enabled: true
        default_to_remote: true
        force_local_flag: false
      persistent_storage:
        volumes:
          - name: "logs-volume"
            size: "1Gi"
            storage_class: "standard"
      additional_k8s_params:
        ros_master:
          spec:
            template:
              spec:
                containers:
                  - name: "ros-master-container"
                    image: "ros:noetic-ros-core"
                    command: [ "/ros_entrypoint.sh" ]
                    args: [ "roscore" ]
        application:
          spec:
            template:
              spec:
                containers:
                  - name: "ros-app"
                    image: "{{ vars.base_image }}" # The pre-built image
                    readinessProbe:
                      exec:
                        command: [ "/usr/local/bin/readiness_probe.sh" ]
                      initialDelaySeconds: 10
                      periodSeconds: 5
                    env:
                      - name: CUSTOM_ENV
                        value: "TESTING"
                    ports:
                      - containerPort: 11311
```

managing Kubernetes Persistent Volumes (PVs) for critical data, such as sensor logs or maps, ensuring that data is retained across restarts of stateful applications. This volume can be used by application developers as a user-defined path in the container filesystem that can always be accessed, with data persistence automatically guaranteed by Kubernetes. When defined in the Rigelfile, those volumes are created in the default namespace, with names following the pattern `vol.name-pvc`, where `vol.name` is provided by the user in the Rigelfile. The PV is accessible to the application as a mounted location on the host node, at `/mnt/data/<vol.name>` unless the `mountPath` optional parameter is present in the Rigelfile, which overrides the default path.

Figure 3.3 shows a simplified UML component diagram of how the plugin interfaces with Rigel's core modules and the Kubernetes API, while Figure 3.2 displays a more detailed view of the plugin's internal components and how they interact with each other.

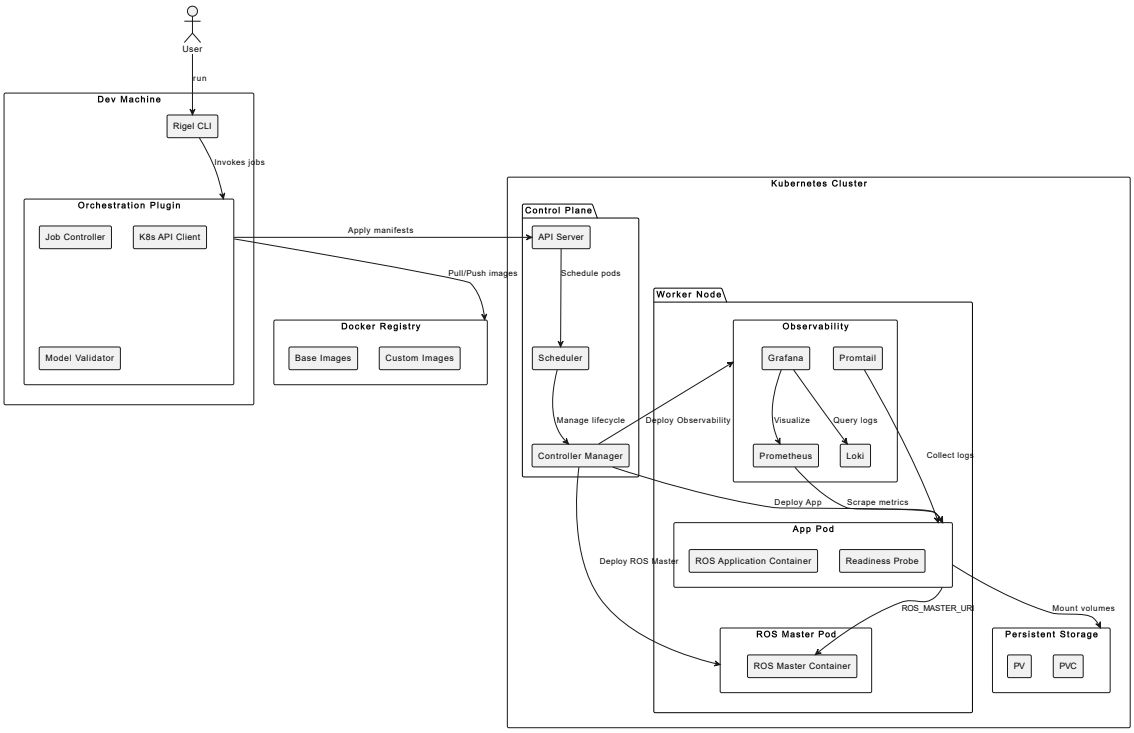

Figure 3.2: Detailed architecture of the orchestration plugin within the Rigel framework.

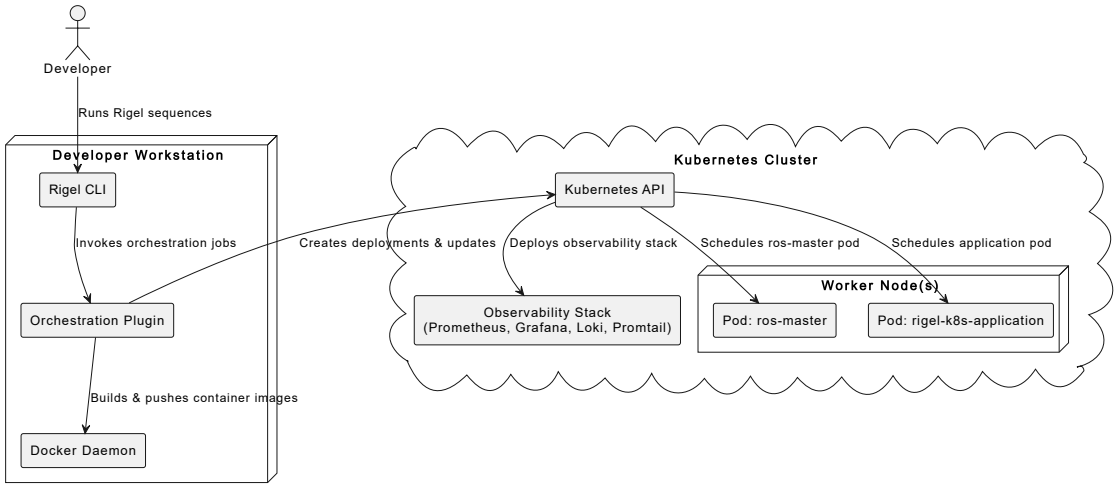

Figure 3.3: High-level architecture of the orchestration plugin within the Rigel framework.

A sequence diagram showing the workflow of the plugin is presented in Figure 3.4.

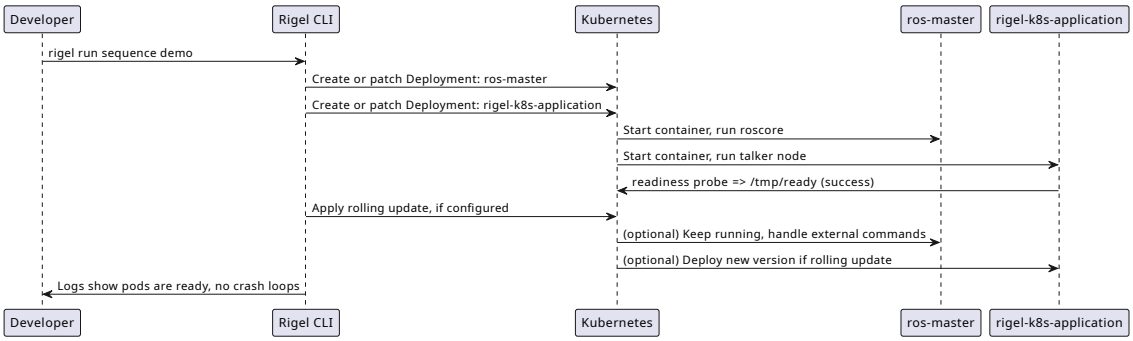

Figure 3.4: Sequence diagram illustrating the deployment workflow for a ROS application using the orchestration plugin.

3.4.1 Networking Implications for ROS

The centralized architecture of ROS 1, which relies on a single ROS Master node to map topics and coordinate nodes in a network, results in inherent challenges in distributed deployments. Unlike ROS 2's decentralized DDS-based communication [10], ROS 1 requires explicit configuration of the `ROS_MASTER_URI` across all nodes to establish connectivity. In distributed deployments, where ROS nodes may be dynamically scheduled with ephemeral IP addresses, traditional ROS 1 networking approaches would require manual reconfiguration of IP addresses.

The orchestration plugin addresses these challenges by deploying the ROS Master in a Kubernetes pod that persists across other pod restarts or rescheduling events, ensuring that all ROS

nodes consistently resolve the Master's address without relying on static IP assignments through automatic Kubernetes network mapping. The plugin injects the `ROS_MASTER_URI` environment variable into every application pod during initialization, eliminating the need for developers to hardcode IP addresses or manually coordinate URI updates across distributed nodes, ensuring ROS nodes can communicate across physical hosts or cloud regions as if they were on a single LAN.

This solution contrasts to traditional ROS 1 deployments, where developers must manually configure firewall rules, set `ROS_IP` and `ROS_HOSTNAME` variables, and maintain hostfile entries – a process error-prone at scale. The plugin's networking abstraction reduces multistep manual procedures to a single declarative configuration in the Rigelfile.

Multi-robot setups have not been addressed and deemed outside the scope of this work, although in such cases the plugin would create multiple `Deployment` objects – one for each robot – each referencing a distinct ROS Master instance or a shared Master, as already supported.

3.4.2 Implementation Details of the Orchestration Plugin

The `OrchestrationPlugin` was designed to integrate with Rigel's job-based architecture, providing orchestration "jobs" that can be executed in a specific sequence. While Figure 3.5 shows how the plugin source code is organized into modules and submodules, Figure 3.6 illustrates how the plugin classes and data models interact.

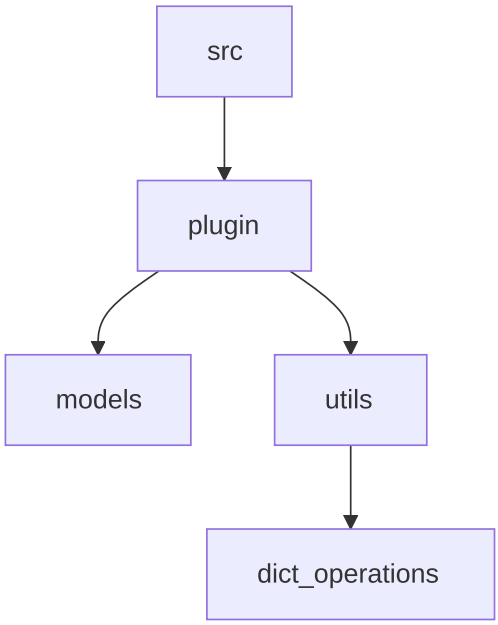

Figure 3.5: Rigel's orchestration plugin code structure – each block corresponds to a module or submodule.

To ensure that the orchestration plugin integrates into Rigel's modular architecture, a dedicated Python class called `OrchestrationPlugin` was created. This class inherits from the

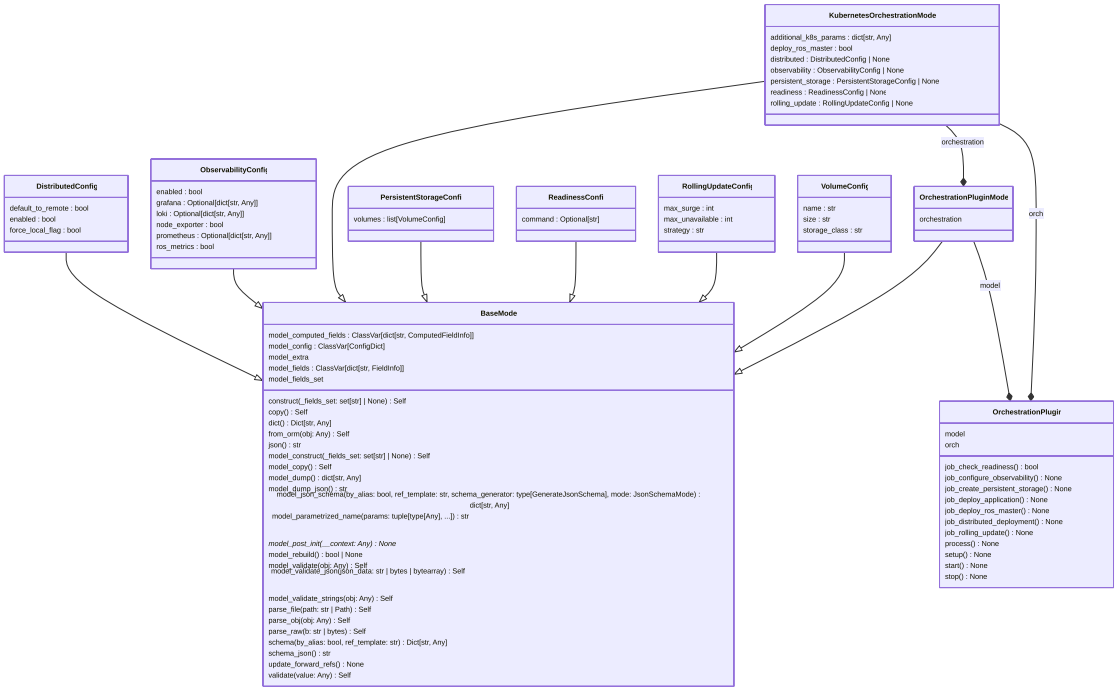

Figure 3.6: UML class diagram of the orchestration plugin.

`PluginBase` interface, which is part of Rigel's core framework for extending functionality through `jobs`. By extending `PluginBase`, the orchestration plugin gains access to Rigel's lifecycle hooks and routines that load parameters from the Rigelfile. Listing 2 illustrates the plugin initialization, complemented by the relevant Pydantic schemas in Listing 3.

Listing 2 Excerpt from `OrchestrationPlugin`, illustrating the plugin's initialization.

```
class OrchestrationPlugin(PluginBase):
    """A Rigel plugin that orchestrates a ROS application in K8s."""

    def __init__(...):
        super().__init__(...)
        # Load our Pydantic model
        self.model: OrchestrationPluginModel = ...

    ...
```

As previously stated, Pydantic is used to structure and validate the configuration parameters supplied by the developer, since inputs must be validated before any Kubernetes resources are created. This approach ensures that incorrect or missing fields are detected early, minimizing potential errors during deployment. Each subsection of the configuration – such as rolling update strategies, readiness probes, or observability settings – is described as a Pydantic model, enabling type safety and clear error messages to end users.

Listing 3 A portion of the Pydantic-based data models used for observability configuration.

```
class OrchestrationPluginModel(BaseModel):
    """The top-level schema for our Orchestration plugin.

    It contains a nested 'KubernetesOrchestrationModel'
    under the key `orchestration`.
    """

    orchestration: KubernetesOrchestrationModel

class KubernetesOrchestrationModel(BaseModel):
    """K8s Orchestration configuration for the application."""

    deploy_ros_master: bool = True
    readiness: ReadinessConfig | None = None
    observability: ObservabilityConfig | None = None
    rolling_update: RollingUpdateConfig | None = None
    distributed: DistributedConfig | None = None
    persistent_storage: PersistentStorageConfig | None = None
    additional_k8s_params: dict[str, Any] = {}

class ObservabilityConfig(BaseModel):
    """Observability configuration for the application."""

    enabled: bool = False
    grafana: dict[str, Any] = Field(...
    prometheus: dict[str, Any] = Field(...
    loki: dict[str, Any] = Field(...
    node_exporter: bool = True
    ros_metrics: bool = True
```

The attribute `additional_k8s_params` in Listing 3 represents a dictionary that enables users to specify custom Kubernetes configurations that are not covered in the plugin to prevent excessive schema complexity. To support user-defined overrides of Kubernetes parameters, the plugin uses the `deep_merge` routine (Algorithm 1), which recursively combines nested dictionaries. This design is important for preserving default Kubernetes configurations while integrating user-specified fields that override those defaults when necessary. A usage example of `additional_k8s_params` is shown in Listing 1.

Algorithm 1 `deep_merge` function for merging nested dictionaries

```
1:  procedure DEEP_MERGE(base, overrides)          ▷ Recursively merges two dictionaries.
2:      merged ← DEEPCOPY(base)
3:      for all (k, v) in ITEMS(overrides) do      ▷ Keys in "overrides" take precedence.
4:          if k ∈ merged and ISINSTANCE(merged[k], dict) and ISINSTANCE(v, dict) then
5:              merged[k] ← DEEP_MERGE(merged[k], v)
6:          else
7:              merged[k] ← v
8:          end if
9:      end for
10:     return merged
11: end procedure
```

3.5 Validation and Testing

The testing strategy for the orchestration plugin is divided between verifying correctness at the code level and validating behavior under realistic deployment conditions.

First, *Pytest* functions are configured to perform unit and integration tests that ensure the plugin's internal logic is valid. The unit tests were responsible for verifying that the Rigelfile YAML is parsed accurately, that Pydantic models catch invalid configurations, and that the construction of Kubernetes resources (such as Deployments or Services) conforms to user expectations. Negative tests were also implemented to confirm the plugin responds appropriately to malformed YAML or missing parameters, with a raised exception. Meanwhile, integration tests validate the plugin's interaction with a single-node local Kubernetes cluster, by asserting that the expected resources were created or updated, namely:

- *Rolling updates* on a sample ROS container that publishes sample data in a loop.
- *Readiness checks* that fail if the container's main ROS node is not active yet or according to user-defined readiness conditions.
- *Basic observability* with Prometheus and Grafana pods automatically launched.

All such tests are run automatically in a CI pipeline using GitHub Actions, shown in Figure 3.7, which ensures that the plugin's behavior is consistent across different Python versions and that the plugin's core features are functioning as expected. To replicate typical robotics scenarios, individual deployments are performed for each of the three ROS communication patterns described in Section 3.6: Services, Actions, and Publish/Subscribe. Aiming to test the plugin, three small demonstration applications are deployed within the same Kubernetes cluster to validate the plugin's ability to manage ROS nodes with different communication patterns. Each application tests a different form of communication and operational challenge – for instance, the synchronous request-response in the Services example, or the continuous data streams in a

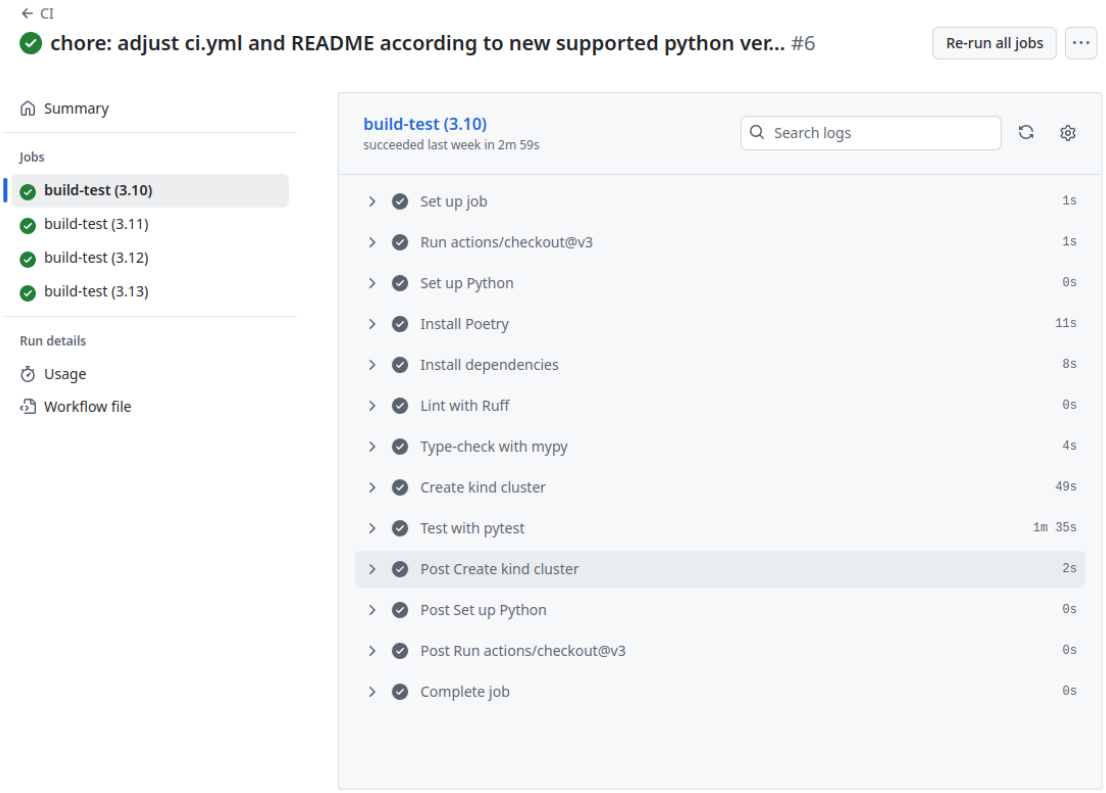

Figure 3.7: Continuous Integration pipeline execution for the orchestration plugin.

publication-subscription setup. As pods are restarted or updated, Grafana indicates whether there are any dropped requests, lost messages, or noticeable latency spikes. The cluster's built-in metrics collection allows tracking container resource usage, whereas logs from Loki indicate whether readiness scripts, node startup steps, and rolling updates occur in the expected order. The success criteria are zero downtime during updates, stable resource usage (no crash loops or memory leaks), and correct gating of readiness so that the system never proceeds to use a node that has not fully initialized its ROS master or node processes, or updates are not performed while a node does not indicate readiness.

A lightweight stress test was also performed to measure whether the orchestration capabilities are still managed under moderate CPU or memory pressure in the host machine. Tools *k6*[11] and *stress-ng*[12] were used to artificially increase resource usage in the cluster, namely setting CPU utilization to a defined threshold, as well as for memory, to see if readiness checks, rolling updates, and log aggregation continue functioning without timeouts or container restarts. These stress tests do not quantify throughput or latency in detail – instead, they serve to confirm that the plugin's fundamental orchestration and health mechanisms do not degrade or fail under a higher load.

[11] https://k6.io/
[12] https://github.com/ColinIanKing/stress-ng

Besides general system metrics, the following variables in particular are collected by the observability stack, and are further analyzed in Chapter 4:

- Deployment Time – Time to pull images and launch pods, measured from the user's execution of `rigel run <command>` to the moment pods become `Ready`.
- Rolling Update Latency – Time for an updated container to become live while the old container is terminated. Zero (or minimal) downtime is verified by Kubernetes container status indicators.
- Resource Utilization – Aggregated CPU and memory usage under typical loads, monitored via Prometheus.
- Message Integrity – No dropped messages or timeouts in ROS topics, measured by topic subscribers and validated with simple checks and simulation-based tests.

3.6 System Use Cases and Case Studies

In order to demonstrate the applicability of the orchestration module to common robotic scenarios, a set of representative use cases was chosen. These cases reflect typical ROS 1 communication patterns and allow the validation of essential features such as rolling updates, distributed deployments, readiness checks, and the observability of ongoing tasks. They also help confirm that the plugin's modular design can address both straightforward and more complex deployments, accommodating robots with limited on-board computation and platforms that offload intensive tasks to cloud or edge servers.

The three selected patterns consist of Service-based interactions, Action-based tasks with feedback, and a publish/subscribe setup. By applying these different modes of communication, the tests show that the orchestration plugin does not impose artificial constraints on the underlying ROS applications and that it respects the real-time or near-real-time needs of each pattern.

Specific objectives in these case studies include verifying the plugin's ability to preserve communication during updates, confirming that readiness checks prevent interruption of critical tasks, and demonstrating that stateful data can be preserved across restarts when persistent storage is enabled.

In the first scenario, a minimal arithmetic **Service** is provided by a containerized ROS node, while another node acts as the client sending numerical requests. This application demonstrates how the plugin handles synchronous request-response workflows, which often appear in robotic systems that trigger sensor readings or command-specific hardware actions. It also confirms that the plugin's rolling update mechanism does not break active service connections; when updates occur, the readiness checks ensure that requests are not terminated mid-execution.

In this deployment a persistent volume is requested and created, such that all arithmetic input and result logs are maintained across container restarts and updates, ensuring that any analysis or debugging data is not lost. The persistent storage feature is associated with the client node,

which logs all requests and responses to a file in the persistent volume, ensuring the application has bidirectional communication and data integrity. This application is illustrated in Figure 3.8.

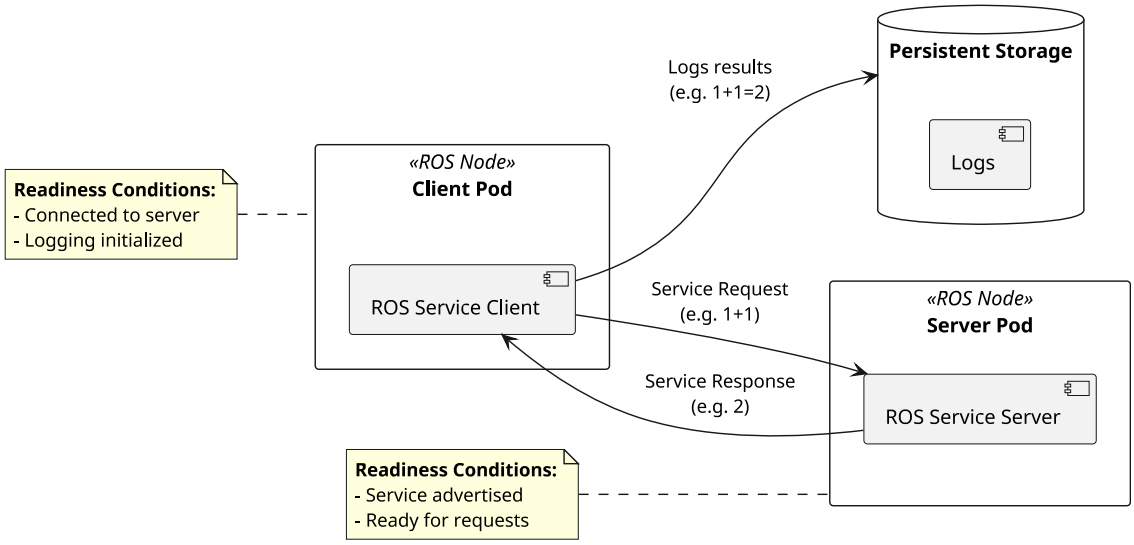

Figure 3.8: Diagram of the Service application.

The second case study involves an **Action**-based communication pattern, which is common for long-running tasks in ROS.

The developed validation application is a simple arithmetic task that involves calculating the prime factorization of a list of numbers sent by the client. To simulate a long-running task, the server pauses for a few seconds before returning each partial result (each prime factor), allowing the client to preempt the Action or receive feedback about the task's progress. Once the factor 1 is sent by the server, the client understands that the task is complete. If the system is processing an input number, the plugin must recognize that it is not safe to shut down or replace the node until the task reaches an idle or preemptible state. Like the Service-based scenario, the client logs all requests and responses to a persistent volume, ensuring that the Action's progress is not lost during updates, and bidirectional communication happens without data loss.

Furthermore, as implemented in all testing scenarios, the observability stack is expected to display real-time information about the Action's progress and resource consumption, allowing developers to identify bottlenecks and confirm that updates do not degrade the system's responsiveness. The results from this use case show that complex, multistep tasks remain stable when the orchestration plugin coordinates updates and monitors node availability. This application is illustrated in Figure 3.9.

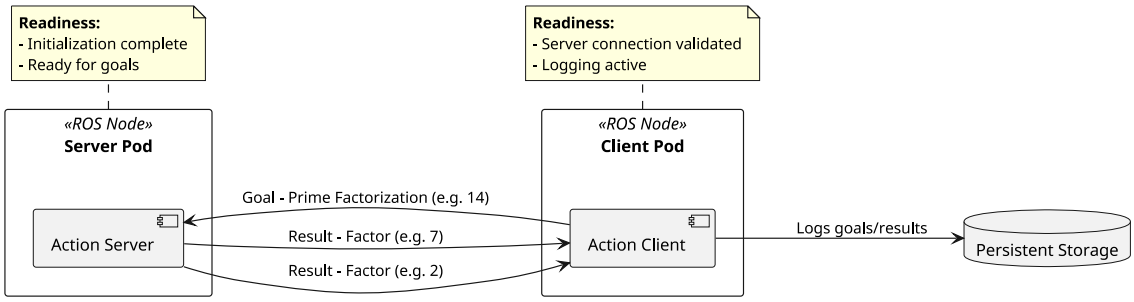

Figure 3.9: Diagram of the Action application.

The final scenario features a typical ROS **publish/subscribe** model, illustrating continuous data streams that are essential in many robotics applications.

A simulation node emulates sensor readings – exemplified here as temperature measurements – and publishes them to a dedicated topic. Subscriber nodes on another pod perform several computations based on the incoming data and stores the results, namely: minimum and maximum values, standard deviations, and averages.

Since publish-subscribe models can involve higher message rates and a variety of subscriber nodes, they also illustrate how the orchestration plugin manages rolling updates for a node that continuously publishes sensor data. In this specific example application, a rate of 500 messages per second is pre-defined. The core requirement is that no messages are dropped or delayed beyond a tolerable threshold during an update, so the readiness gates and rolling mechanism must ensure that the system remains operational. This application is illustrated in Figure 3.10.

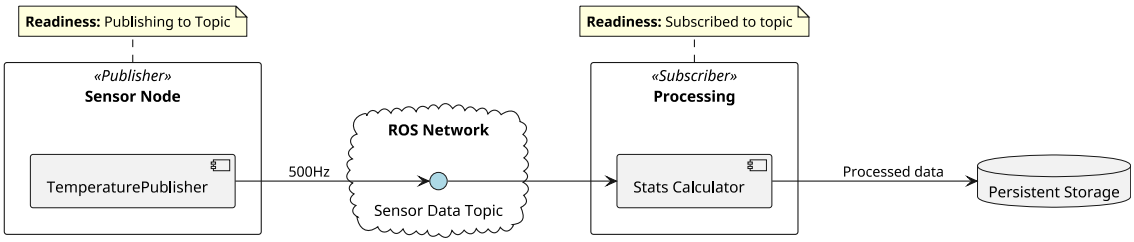

Figure 3.10: Diagram of the Publish/Subscribe application.

3.7 Readiness Considerations for ROS Communication Mechanisms

ROS provides several standardized communication patterns that facilitate the exchange of information and coordination among distributed nodes. This section presents the three main approaches explored in the case studies – Publish-Subscribe topics, Services, and Actions – along with their respective considerations for determining when the system is not impeded to be updated or restarted (system readiness).

The Publish-Subscribe model enables unidirectional, asynchronous communication. In this paradigm, nodes publish messages to specific topics, while subscribers receive messages by subscribing to the same topics.

Before a node can publish or subscribe to a topic, it must signal readiness to ensure that messages are not lost during the communication process. The readiness conditions for publish-subscribe are as follows:

- Publisher Readiness – A publisher signals readiness after it has completed its initialization routine and has begun publishing messages.
- Subscriber Readiness – A subscriber signals readiness only after it has successfully subscribed to the relevant topic(s) and can receive messages.

Services adopt a synchronous request-response communication pattern. In this framework, a client sends a request to a server, which processes the request and returns a response. Below are the readiness considerations for the Services communication pattern:

- Server Readiness – The server signals readiness after it advertises the Service and completes any necessary initialization procedures (e.g., establishing a database connection).
- Client Readiness – The client signals readiness once it has successfully connected to the server, ensuring that requests can be sent and responses received.

Actions implement asynchronous, bidirectional communication intended for long-running tasks. Clients send goals to Action servers, monitor feedback, and wait for results upon completion. Below are the readiness considerations for the Actions communication pattern:

- Server Readiness: The server signals readiness after it finishes initialization and begins accepting goals from clients.
- Client Readiness: The client signals readiness once it confirms the server is available and can properly handle goal requests.

Table 3.3 provides a concise comparison of the communication type, readiness triggers for publishers/servers, readiness triggers for subscribers/clients, and critical operations for each of the three ROS communication patterns.

Table 3.3: Comparison of ROS communication patterns

Aspect	**Pub/Sub**	**Services**	**Actions**
Communication Type	Unidirectional, asynchronous	Bidirectional, synchronous	Bidirectional, asynchronous
Publisher/Server	Signals readiness after publishing	Signals readiness after advertising	Signals readiness after accepting goals
Subscriber/Client	Signals readiness after subscribing	Signals readiness after connecting	Signals readiness after connecting
Critical Operations	Topic connection	Request-response initialization	Long-running goal processing

In summary, each communication pattern offers distinct advantages depending on the nature of the task. The readiness conditions must be adapted to each mechanism to ensure timely updates and uninterrupted operations throughout the system.

In publish-subscribe ROS applications, a publisher is considered ready after it initializes and begins sending messages. Conversely, a subscriber is considered ready once it has confirmed its subscription to the relevant topic.

A ROS Service server completes its initialization and advertises the Service before signaling readiness. The Service client, on the other hand, defers its readiness signal until it can properly connect to the Service server and exchange request-response messages.

In ROS Actions, the server indicates readiness when it is fully initialized and capable of accepting goals. Meanwhile, the client remains on standby until a connection to the server is validated, ensuring that goal requests can be processed without interruption.

In all examples, a single script (shown in Listing 4) is used to verify node readiness. The script checks for the presence of a specific file (e.g., `/tmp/ready`) to determine whether the node is operational. Application code can be modified to create or delete this file based on the node's readiness state, allowing the health check script to accurately assess the node's status.

Listing 4 Health check script for ROS nodes

```
#!/bin/bash
if [ -f /tmp/ready ]; then
    exit 0  # Ready
else
    exit 1  # Not ready
fi
```

3.8 Usage of the Orchestration Plugin

Users primarily interact with the plugin by editing their Rigelfile and invoking `Rigel` commands in the terminal (e.g., `rigel run sequence demo` or `rigel run job deploy_k8s`). In the Rigelfile, developers declaratively specify plugin features they want to enable (see Listing 1), such as readiness checks, rolling update preferences, or persistent volumes. The plugin parses these settings and checks for invalid or missing fields, according to the Pydantic models defined in the plugin code and presented in the plugin documentation[13]. If a runtime issue arises, the plugin emits an error message before attempting to create or update any resources.

Once the input parameters are validated, the plugin generates Kubernetes manifests (e.g., *Deployment*, *Service*, *PersistentVolumeClaim*) and applies them via the Python client library for Kubernetes[14]. The components in the observability stack are also provisioned in the Kubernetes

[13] https://github.com/nhpf/rigel-orchestration-plugin/tree/master/docs
[14] https://github.com/kubernetes-client/python

cluster when requested by the user. The developer can monitor resource creation and application health via the plugin's console output, Kubernetes commands (`kubectl`), or the Grafana dashboards automatically provisioned when observability is enabled.

As discussed in Section 3.7, readiness checks in ROS-based applications may be tied to specific events such as "service advertised," so developers should implement custom logic that adds or removes the `/tmp/ready` file in the container filesystem to signal readiness. The default readiness probe, configured by the plugin, already checks for this file's existence. Developers can still implement their own readiness script that adheres to the Kubernetes readiness code convention (0 = ready, 1 = not ready) and declare it as system command in the `deploy_k8s.orchestration.readiness.command` parameter, or as a different system command for each container inside the `readinessProbe.exec.command` parameter inside the container specification in the Rigelfile.

The built-in observability stack provides real-time monitoring of ROS nodes, Kubernetes resources, and system metrics. By default, it captures CPU and memory usage, container logs, among other metrics specified in Chapter 4, when the user sets the parameter `observability.enabled: true` in the Rigelfile `deploy_k8s.orchestration` section. If the user wants to integrate custom metrics, the application code can use Prometheus exporters[15], which integrate application-specific values to the stack (e.g., messages processed, average sensor reading) and can be visualized in Grafana.

If the developer wants to configure a common dashboard for all ROS nodes, it is possible to package a Grafana dashboard JSON file and reference it in the `observability.grafana` section of the `additional_k8s_params` parameter in the Rigelfile. Listing 6 shows a configuration example of observability features in the plugin.

To perform application updates, developers can run `rigel run job update_k8s`, which triggers a rolling update of the ROS nodes. By default, the plugin uses a rolling update strategy with `maxUnavailable=0` and `maxSurge=1`, ensuring zero downtime for ROS nodes, as assessed in Chapter 4. Parameters supported by this job are exemplified in Listing 5.

Listing 5 Example of a `update_k8s` job in a Rigelfile.

```yaml
update_k8s:
  plugin: "src.plugin.OrchestrationPlugin"
  with:
    orchestration:
      rolling_update:
        strategy: "Rolling"
        max_surge: 1
        max_unavailable: 0
      readiness:
        command: "/usr/local/bin/readiness_probe.sh"
```

And users requiring different update strategies may specify their own Kubernetes configuration using the aforementioned Rigelfile parameters and use `additional_k8s_params` in

[15] https://prometheus.io/docs/instrumenting/exporters/

Listing 6 Example observability parameter of a `deploy_k8s` job in a Rigelfile.

```yaml
deploy_k8s:
  with:
    orchestration:
      observability:
        enabled: true
        grafana:
          enabled: true
          admin_password: "rigel-admin"
          default_dashboards: true
          service_type: "NodePort"
          persistence:
            enabled: true
            size: "2Gi"
        prometheus:
          enabled: true
          scrape_interval: "10s"
          evaluation_interval: "10s"
          retention: "7d"
        loki:
          enabled: true
          retention: "7d"
          persistence:
            enabled: true
            size: "5Gi"
        node_exporter: true
        ros_metrics: true
```

the Rigelfile. When the application needs to retain data throughout updates or restarts, the user can declare a persistent volume in the `orchestration.persistent_storage` section of the `deploy_k8s` job declaration in the Rigelfile, as detailed previously in this chapter.

The execution of Rigel sequences and jobs is designed to integrate into automated CI/CD pipelines, such that ROS applications can be containerized, deployed, and updated within a controlled environment such as GitHub Actions whenever code changes are pushed to a repository. The plugin is made available in the author's GitHub repository[16] and is licensed under the MIT License. The Rigel framework also follows the MIT License, allowing users to freely modify, distribute, or integrate it into their own projects. For users or contributors who wish to enhance or fix the plugin, contribution guidelines and a code of conduct are provided in the repository's `CONTRIBUTING.md` file.

[16] https://github.com/nhpf/rigel-orchestration-plugin

Chapter 4

Experimental Validation and Results

This chapter presents the experimental validation of the orchestration plugin developed for Rigel, focusing on both quantitative and qualitative results. The experiments, previously described in Section 3.6, were designed to evaluate the plugin's performance, reliability, and usability under common conditions relevant to robotics software development. The validation process simulates common challenges faced in containerized deployments, such as high usage of host system resources, ensuring zero-downtime updates, and maintaining communication across distributed systems.

Section 4.1 describes the experimental setup, including the hardware, software, and testing tools used for validation. Section 4.2 provides results for each of executing the applications applying the three major ROS application patterns (Service, Action, Publish-Subscribe) using the orchestration plugin. Section 4.3 presents results and discusses how they reflect the plugin's success.

The validation procedures were performed on a single Kubernetes cluster within an isolated virtual machine, characterized in Section 4.1, to assess the plugin's scalability and fault tolerance in a controlled environment. Metrics such as CPU usage, memory consumption, network bandwidth, request response time, and system uptime were collected using the observability module. Load testing with *k6* and *stress-ng* provided insights into the system's performance under varying workloads, enabling a detailed evaluation of the plugin's robustness and efficiency.

4.1 Experimental Setup

All experiments were performed on a `t2.medium` AWS Elastic Compute Cloud (EC2) instance. This configuration was chosen to simulate a constrained environment typical of robotic systems. Table 4.1 summarizes the hardware and software details.

The EC2 instance was configured to ensure minimal interference from background processes. Only essential services were enabled, and system monitoring verified that resource usage was

Table 4.1: Summary of the experimental hardware and software environment.

Component	Specification
Instance Type	`t2.medium` (2 vCPUs, 4GB RAM)
Hosting Region	eu-central-1 (Frankfurt)
CPU Type	Intel Xeon E5-2686 v4 (2.30GHz)
Operating System	Ubuntu 22.04.5 LTS x86_64
Kernel Version	Linux 6.8.0-1023-aws
Storage	30GB SSD
Container Orchestrator	Minikube 1.35 - Kubernetes 1.32
Container Engine	Docker 28.0.1
Middleware	ROS 1 (Noetic)
Observability Tools	Prometheus v3.2.1, Grafana 11.5.2-ubuntu, Promtail 3.4, Loki 3.4
Testing Tools	k6 v0.55.0 for load testing

stable before starting each experiment. Besides standard operating system processes and the deployed application, only an SSH server was running to support remote connections to the instance. SSH access for that EC2 instance is managed and provided by INESC TEC.

As described in Chapter 3, three ROS applications were tested to cover different communication patterns: the Service Model (arithmetic Service), the Action Model (prime factorization tasks), and the Publish-Subscribe Model (simulated temperature sensor). Each of the three ROS applications were developed with Python 3.9 with ROS 1 Noetic and containerized using Rigel, with orchestration parameters defined in a Rigelfile. The plugin automated the generation of Kubernetes deployment files and configured the pods within the same Minikube cluster that handled the orchestration tasks. The observability module integrated into the plugin enabled real-time monitoring of system metrics. Since Prometheus is already configured to collect several metrics from Kubernetes clusters, the plugin only needed to scrape application-specific metrics from ROS nodes and publish them to the observability stack. Among the system metrics collected, the following are highlighted, as they are particularly relevant for performance analysis of robotic applications:

- CPU Usage – The percentage of CPU resources consumed by each pod.
- Memory Consumption – The amount of RAM allocated and used by the pods.
- Bandwidth – The network usage of ROS topics and requests between nodes.
- Number of Requests – The total number of Service calls or messages processed, with tags to categorize requests.
- Uptime – The duration of uninterrupted service, particularly during rolling updates.
- Response Time – The average latency of service responses and topic subscriptions.

Prometheus was configured to scrape metrics from containerized ROS nodes periodically. The data was visualized using pre-configured Grafana dashboards, as shown in Figure 4.1.

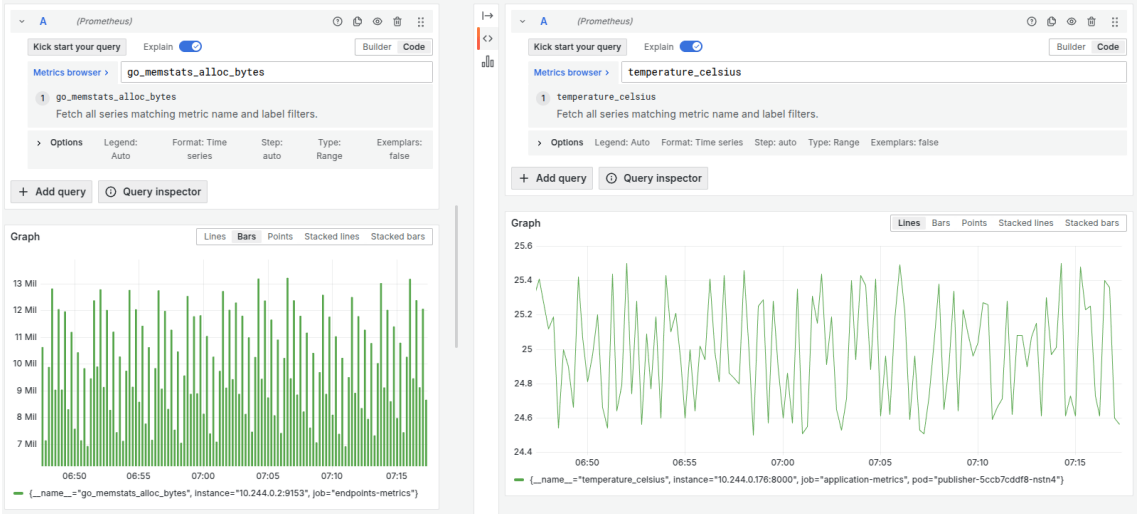

Figure 4.1: System and application metrics from the Grafana observability dashboard for the Pub/Sub application.

Each service was subjected to a total of 10 runs to minimize statistical anomalies caused by external factors. `stress-ng` allowed for controlled stress testing of the CPU and memory usage of the host virtual machine, ensuring that the plugin could maintain service availability under competing loads without crashing. Each stress test was conducted with one application instance with 80% CPU load to simulate a high-concurrency scenario, and likewise for RAM – 80% memory load.

The validation process involved running the experiments with the orchestration plugin, which was used to deploy and manage each ROS application, already leveraging rolling updates, readiness checks, and observability integration.

The experimental setup was designed with the following assumptions and limitations: real-world robotic tasks were not simulated in detail; instead, synthetic tasks representative of typical ROS workloads were used. And as the experiments were conducted on a single EC2 instance, limiting the evaluation of multi-host deployments to a single Kubernetes cluster. Future work should extend these experiments to more complex robotic scenarios and evaluate multi-host deployments on physical robots or edge-cloud infrastructures.

4.2 Experimental Results

The first scenario evaluates rolling updates and readiness checks on a ROS Service in which the server performs the addition of two numbers provided by the client, returning the result.

The system consists of two containers: a Service server node advertising an `add_two_ints` Service, and a client node repeatedly requesting integer sums, logging the received responses to a persistent volume. A rolling update was simulated by pushing a new container image with modified server logic, subtracting the integers instead of adding them. The plugin's `deploy_k8s` job was then re-run, triggering Kubernetes to replace the old Service pod with the new version as a

rolling update. Despite the container replacement, the Service remained continuously available, corroborated by zero dropped Service calls in the client logs. This outcome aligns with the readiness gating logic enforced by the plugin, where the old container only terminated once the new container indicated readiness (i.e., no arithmetic requests were in progress).

Table 4.2 summarizes metrics collected during steady-state operation and the update window.

Table 4.2: Metrics for the Service scenario (mean of 10 trials).

Metric	**Steady-State**	**During Update**
CPU Usage	23.4% (±2.5%)	28.1% (±3.2%)
Memory	280 MB (±30 MB)	290 MB (±35 MB)
Request Latency (95%)	45 ms	83 ms
Dropped Requests	0	0

Compared to the baseline, CPU usage spiked during the rolling update yet remained within the instance's capacity. Latency increased slightly, but all requests still completed successfully. These results indicate that the orchestration plugin handled the update seamlessly, validating the readiness checks and rolling update logic for a synchronous request-response workflow.

Next, the plugin's readiness and rolling update features were assessed under a ROS Action model, where tasks can run for multiple seconds. A prime factorization server was developed to accept a goal (a list of integers) and return partial factor results over time until completion. The readiness script delayed updates whenever the server was actively processing a goal, ensuring no actions were prematurely halted.

Throughout a series of updates, no Action goals were interrupted mid-processing. The server logs revealed that updates were postponed until all active goals finished, in accordance with the readiness script that checks for an "idle" status, set in the application code when the server was not actively processing any goals. Table 4.3 illustrates the measured latency in performing new goals before and after updates.

Table 4.3: Action server latency for prime factorization tasks (mean of 10 trials).

Metric	**Task execution without an update**	**Task execution with an update**
Task completion time	220 ms	388 ms
Actions Aborted	0	0

The numbers being factored were all integers from 1000 to 9999, in batches of 10, with the server returning each factorization result. By associating readiness with Action server state, this scenario validates that the plugin's design can protect long-running tasks from disruption.

The third scenario employs a publish-subscribe model, where a simulated temperature sensor node publishes data at a fixed frequency of 500 Hz, and a subscriber node processes incoming

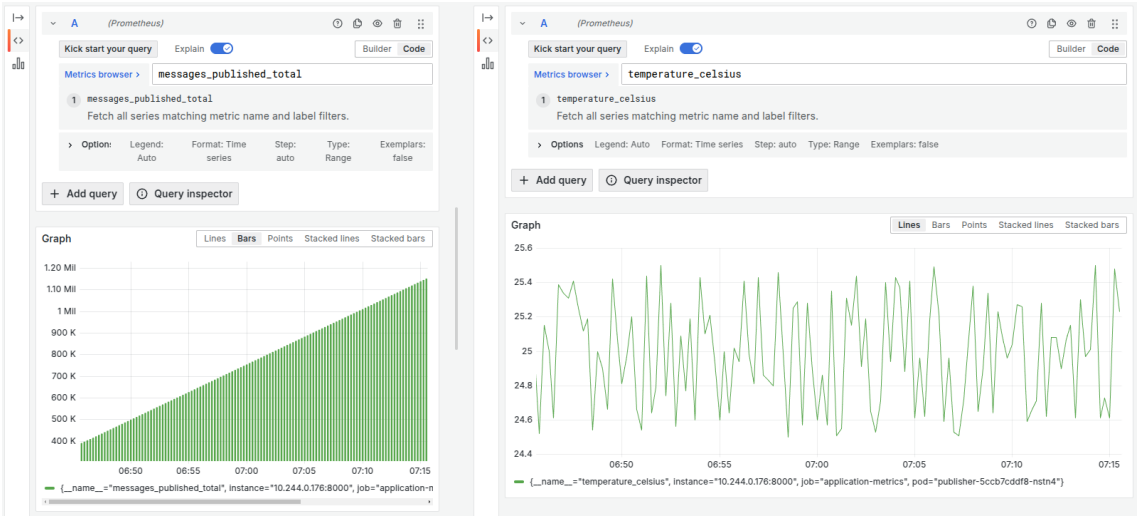

Figure 4.2: Grafana dashboard showing CPU and memory usage during the publish-subscribe scenario steady state.

messages to calculate statistics such as moving averages and peak values. This setup is representative of sensor-driven robotics applications, where data streams must remain uninterrupted and message integrity is necessary.

During steady-state operation, depicted in Figure 4.2, the subscriber successfully processed all messages without loss. To evaluate rolling updates, a new container image was pushed, causing Kubernetes to replace the publisher container. Without the orchestration plugin's readiness script, there were instances when messages were dropped while the subscriber was being replaced, as evidenced by Table 4.4.

Table 4.4: Message integrity during rolling updates in Pub/Sub model.

Update Type	**Messages Sent**	**Messages Received**	**Message Loss (%)**
Minor Subscriber Update	100,000	99,992	0.00008%
Major Subscriber Update	100,000	99,993	0.00007%
Simultaneous Update	100,000	99,983	0.00017%

The minor subscription update represents a change to the Docker image tag, while the major update includes additional dependencies (`pip install numpy`), and the simultaneous update is a change of tags in both applications at once. At 500 Hz, The message loss percentage suggests a downtime of, respectively, 16 milliseconds, 14 milliseconds, and 34 milliseconds. The timeframes under analysis in each scenario were arbitrarily chosen by selecting 100000 samples containing the update period.

To assess latency, end-to-end request times were measured for the Service and Action models, while message round-trip represents the latency estimation for a previous implementation of the publish-subscribe model. Figure 4.3 compares these latencies during normal operation and during rolling updates, with the bars or curves labeled by the relevant scenario.

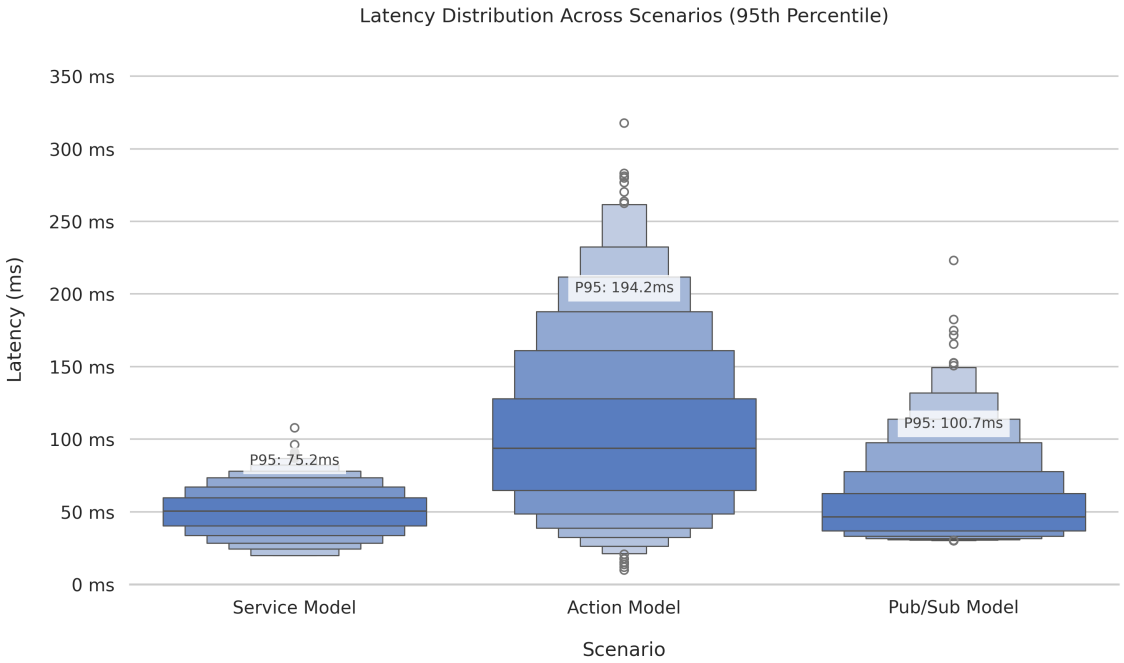

Figure 4.3: Comparison of average latency (ms) before and during rolling updates.

Even under stress test conditions, the maximum observed latency remained under 250 ms for both the Service and Publish-Subscribe models. In the Action model, which involved a simulated prolonged calculation of prime divisors of a list of numbers, the average latency reached 350 ms due to the task's inherent complexity.

The load testing with CPU stressors for the previous version of the publish-subscribe application revealed that the plugin could maintain service availability under high loads, with no service interruptions or critical errors reported, although there was an increase in latency, as depicted in Figures 4.4, 4.5, 4.6, and 4.7.

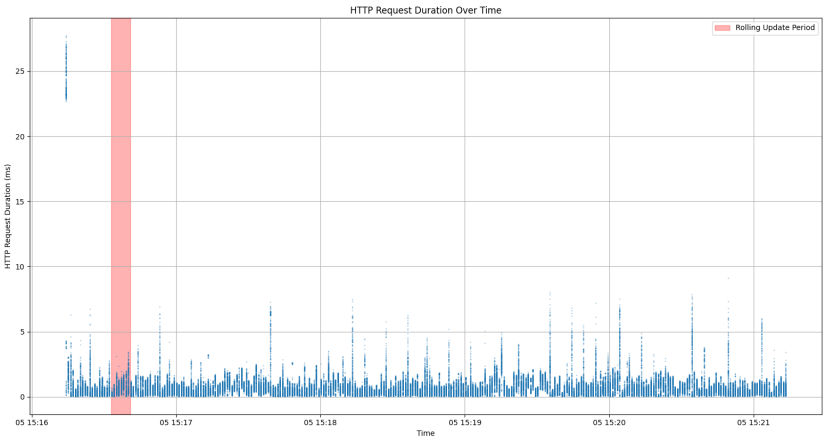

Figure 4.4: Pub/Sub response time with 0% CPU load.

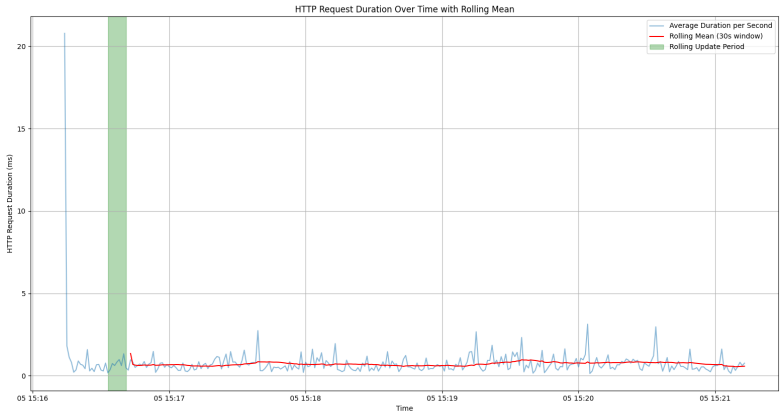

Figure 4.5: Pub/Sub mean response time with 0% CPU load.

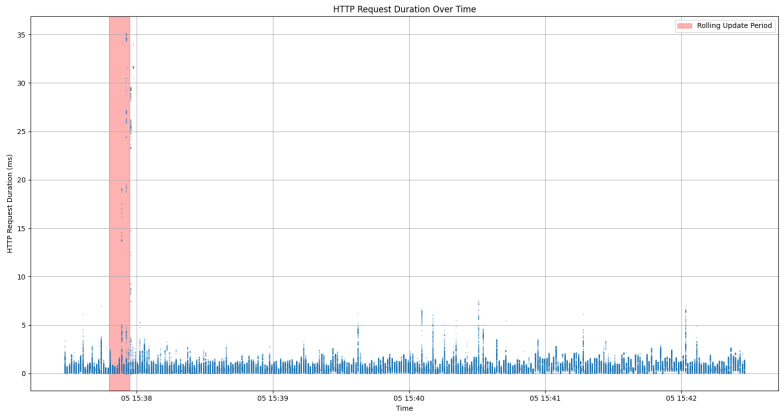

Figure 4.6: Pub/Sub response time with 80% CPU load.

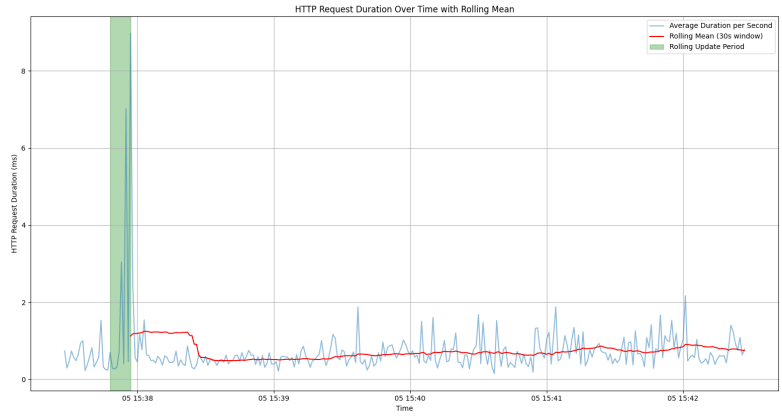

Figure 4.7: Pub/Sub mean response time with 80% CPU load.

4.3 Discussion

Overall, these results suggest that the orchestration plugin delivers benefits in managing containerized ROS applications under a range of typical robotics workloads. The observed stability in the arithmetic Service scenario indicates the plugin's ability to maintain uninterrupted availability throughout rolling updates: the measured CPU and memory usage remained within reasonable limits for the `t2.medium` virtual machine, and no dropped requests were recorded even when the container images were changed mid-operation. This indicates that readiness checks effectively prevented the plugin from shutting down active service calls, therefore ensuring continuity for synchronous request-response traffic.

For the prime factorization Action server, where tasks were more complex, the plugin successfully postponed node updates until the currently running goals were complete. The zero interruption rate in long-lived actions indicates the importance of application-specific readiness hooks in robotics, especially in scenarios where partial progress cannot be discarded. By controlling when pods are replaced out, the plugin Action through Kubernetes ensures that users can safely update complex applications without risking abrupt terminations of important tasks, which can be critical in real-world robotics pipelines.

In the publish-subscribe scenario, message loss metrics remained low across both minor and major updates, demonstrating that the rolling update mechanism and readiness gating can handle higher communication frequencies such as 500 Hz with negligible downtime. The slight increase in packet loss or latency is well within tolerable ranges for many robotic sensing tasks, and the plugin's approach to gating subscriber or publisher replacements helps minimize transient disruptions. Moreover, the stress tests showed that these benefits persisted even when the host machine was resource-constrained, indicating effective container-lifecycle management in high load conditions.

These observations confirm that the orchestration plugin can provide value to robotic software developers by integrating rolling updates, readiness checks, and observability into a single, developer-friendly workflow. The ease of deployment and management of ROS applications was also empirically evaluated in this study, although not numerically analyzed through surveying due to timing constraints when the plugin development reached reasonable maturity.

Chapter 5

Conclusions

This chapter consolidates the outcomes derived from the research, development, and validation phases of the Kubernetes-based orchestration module for the Rigel framework. It evaluates how the work addresses the original research questions, summarizes technical contributions to robot software development, acknowledges limitations, and identifies possibilities for future work. The findings demonstrate that container orchestration can significantly improve the reliability, scalability, and maintainability of ROS-based robotic systems while abstracting infrastructure complexities from developers.

5.1 Summary of Findings and Contributions

The research was guided by three core research questions, each addressed through the design and validation of the orchestration module:

1. *RQ1: How can modern orchestration technologies be adapted to meet the real-time and hardware-specific constraints of typical ROS-based robotic applications?* – The plugin bridges the gap between cloud-native orchestration technology and robotic software through the following mechanisms: (1) readiness probes synchronized with ROS node lifecycle states (e.g., Service initialization, Action goal completion), (2) persistent storage integration for data retention during updates or resets, and (3) Automatic networking mapping that maintains communication integrity in distributed deployments. By abstracting networking configurations (e.g., automatic injection of `ROS_MASTER_URI`), the plugin enables ROS 1 nodes to operate across Kubernetes pods as if they were on a single machine, mitigating hardware-specific constraints through workload offloading.

2. *RQ2: What methodologies are required to enable zero-downtime rolling updates, improve observability, and support distributed deployments while minimizing the learning curve for robotics developers?* – The module implements zero-downtime rolling updates by combining Kubernetes' native rollout strategies with custom readiness checks that gate updates until robotic tasks reach an application-defined idle state. Pre-configured observability tools

(Prometheus, Grafana, Loki) provide real-time metrics and logs without requiring manual setup. Validation tests demonstrated 99.99% message integrity during updates in high-frequency pub/sub workflows and zero interruptions for long-running Action servers, proving the methodology's efficacy for critical robotic applications.

3. *RQ3: How can the developed Rigel orchestration plugin remain flexible for robot software developers to define custom application management logic while simplifying their decisions?* – The plugin achieves this through a declarative configuration schema in the Rigelfile, which allows developers to define advanced orchestration logic (e.g., resource limits, custom probes) while defaulting to parameters established for general ROS applications. The abstraction provided by the plugin ensures that even users unfamiliar with Kubernetes can deploy applications confidently, while retaining full control through the `additional_k8s_params` structure within a Rigelfile.

The main contribution of this work is the Kubernetes orchestration plugin for Rigel that automates deployment, scaling, and updates of ROS applications. By creating Rigel jobs dedicated for orchestration, the plugin extends the framework's capabilities to support Kubernetes deployments and updates after containers are already built by the Rigel core modules, enabling the extension of existing CI/CD pipelines. Another contribution is the validation of the plugin's effectiveness in managing ROS applications, with the characterization of how readiness checks should be implemented according to each main ROS communication pattern. A third contribution is the development of methodologies for solving ROS 1 constraints for developers of legacy robotic systems, including ROS Master persistence, network abstraction, and persistent volume management, as well as establishing the fundaments to distributed deployments that allow offloading of computationally intensive nodes to cloud resources while maintaining low-latency communication with on-robot components. And finally, this dissertation introduced a pre-configured observability stack with sensible defaults for ROS 1 applications, providing real-time monitoring of ROS nodes, Kubernetes resources, and application-specific metrics.

5.2 Limitations and Future Work

While the developed solution presents a first approach for integrating Kubernetes orchestration with ROS applications, several limitations and opportunities for future work remain.

The plugin's reliance on a centralized ROS Master limits its applicability to ROS 2, which uses decentralized discovery via DDS. Integrating ROS 2 would require a different approach to service discovery and communication, leveraging DDS-Security for inter-pod communication, and with the ROS 1 EOL being reached, this is a critical next step for the plugin's evolution.

Experiments with the plugin were conducted on a single Kubernetes cluster due to resource constraints. Multi-cluster deployments, as in cloud providers, remain untested but are theoretically supported and should be explored further in a scientific publication that the author intends to develop based on this dissertation.

While the plugin provides default readiness scripts, these require developers to manually signal readiness via filesystem flags. Future iterations could automate this through ROS API hooks, for example, by leveraging the `rospy.on_shutdown()` callback to impose an appropriate readiness state for the respective pod.

Another avenue for improvement is implementing Kubernetes Network Policies to protect distributed deployments in untrusted networks, as well as developing auto-scaling policies that dynamically shift workloads between edge devices and cloud servers based on task criticality and network conditions, which can add value to swarm robotics applications.

This dissertation demonstrates that software orchestration, when adapted to robotic applications, can enhance the reliability and scalability of ROS-based systems in a development paradigm that still does not broadly adopt these technologies. By abstracting infrastructure complexities through Rigel's interface, the module allows robot software developers to focus on application logic rather than deployment processes. Validation results confirm that zero-downtime updates and a ROS-oriented observability setup are achievable in practice, ensuring that the development of robotic applications can benefit from the same automation and monitoring standards observed in other software engineering domains.

References

[1] Francesco Lumpp, Marco Panato, Franco Fummi, and Nicola Bombieri. A container-based design methodology for robotic applications on kubernetes edge-cloud architectures. volume 2021-September, pages 01–08. IEEE, Sep 2021. Accessed: 2023-11-02. doi:10.1109/FDL53530.2021.9568376.

[2] Chongkun Xia, Yunzhou Zhang, Lei Wang, Sonya Coleman, and Yanbo Liu. Microservice-based cloud robotics system for intelligent space. *Robotics and Autonomous Systems*, 110:139–150, Dec 2018. Accessed: 2024-01-01. doi:10.1016/J.ROBOT.2018.10.001.

[3] Ben Kehoe, Sachin Patil, Pieter Abbeel, and Ken Goldberg. A survey of research on cloud robotics and automation. *IEEE Transactions on Automation Science and Engineering*, 12(2):398–409, 2015. Accessed: 2024-01-02. doi:10.1109/TASE.2014.2376492.

[4] Andrea Tosatto, Pietro Ruiu, and Antonio Attanasio. Container-based orchestration in cloud: State of the art and challenges. *Proceedings - 2015 9th International Conference on Complex, Intelligent, and Software Intensive Systems, CISIS 2015*, pages 70–75, Aug 2015. Accessed: 2024-01-02. doi:10.1109/CISIS.2015.35.

[5] Francesco Lumpp, Marco Panato, Nicola Bombieri, and Franco Fummi. A design flow based on docker and kubernetes for ros-based robotic software applications. *ACM Transactions on Embedded Computing Systems*, 2023. Accessed: 2023-11-02. doi:10.1145/3594539.

[6] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro V. Papadopoulos. Real-time containers: A survey. *OpenAccess Series in Informatics*, 80:7:1–7:9, Apr 2020. Accessed: 2025-01-18. doi:10.4230/OASIcs.Fog-IoT.2020.7.

[7] Pedro Melo, Rafael Arrais, Sérgio Teixeira, and Germano Veiga. A container-based framework for developing ros applications. *IEEE International Conference on Industrial Informatics (INDIN)*, 2022-July:280–285, 2022. Accessed: 2023-11-21. doi:10.1109/INDIN51773.2022.9976083.

[8] Michel Albonico, Milica Đorđević, Engel Hamer, and Ivano Malavolta. Software engineering research on the Robot Operating System: A systematic mapping study. *Journal of Systems and Software*, 197:111574, Mar 2023. Accessed: 2024-12-05. doi:10.1016/J.JSS.2022.111574.

[9] João Pedro and Chaves Castilho. *ROS 2.0 – Study and Evaluation of ROS 2 in comparison with ROS 1*. PhD thesis, Oct 2022. Accessed: 2024-12-31. URL: https://hdl.handle.net/10316/102876.

[10] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. Robot operating system 2: Design, architecture, and uses in the wild. *Science Robotics*, 7(66), May 2022. Accessed: 2025-01-29. doi:10.1126/scirobotics.abm6074.

[11] M. Montemerlo, N. Roy, and S. Thrun. Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (carmen) toolkit. In *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, volume 3, pages 2436–2441 vol.3, 2003. Accessed: 2025-01-29. doi:10.1109/IROS.2003.1249235.

[12] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009. Accessed: 2024-08-09. URL: https://ai.stanford.edu/~mquigley/papers/icra2009-ros.pdf.

[13] P. Newman. Moos - mission orientated operating suite. *OUEL Report*, 2299/08, Jan 2008. Accessed: 2025-01-28. URL: https://ora.ox.ac.uk/objects/uuid:e7948d5b-efc3-481a-ad55-f0d39c28c74e.

[14] H. Bruyninckx. Open robot control software: the orocos project. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, volume 3, pages 2523–2528 vol.3, 2001. Accessed: 2025-01-29. doi:10.1109/ROBOT.2001.933002.

[15] Alexandre Sawczuk da Silva, Andreas Kreutz, Gereon Weiss, Johannes Rothe, and Christoph Ihrke. Devops in robotics: Challenges and practices. In *Software Architecture. ECSA 2022 Tracks and Workshops*, pages 284–299, Cham, 2023. Springer International Publishing. Accessed: 2024-01-06. doi:10.1007/978-3-031-36889-9_20.

[16] Leonardo Leite, Carla Rocha, Fabio Kon, Dejan Milojicic, and Paulo Meirelles. A survey of devops concepts and challenges. *ACM Comput. Surv.*, 52(6), Nov 2019. Accessed: 2024-01-05. doi:10.1145/3359981.

[17] Abhisek Omkar Prasad, Pradumn Mishra, Urja Jain, Anish Pandey, Anushka Sinha, Anil Singh Yadav, Rajan Kumar, Abhishek Sharma, Gaurav Kumar, Karrar Hazim Salem, Avdhesh Sharma, and Anil Kumar Dixit. Design and development of software stack of an autonomous vehicle using robot operating system. *Robotics and Autonomous Systems*, 161:104340, Mar 2023. Accessed: 2024-01-02. doi:10.1016/J.ROBOT.2022.104340.

[18] Sérgio Teixeira, Rafael Arrais, and Germano Veiga. Cloud simulation for continuous integration and deployment in robotics. In *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, pages 1–8, 2021. Accessed: 2024-01-03. doi:10.1109/INDIN45523.2021.9557476.

[19] Javad Dogani, Reza Namvar, and Farshad Khunjush. Auto-scaling techniques in container-based cloud and edge/fog computing: Taxonomy and survey. *Computer Communications*, 209:120–150, 2023. Accessed: 2024-01-03. doi:10.1016/j.comcom.2023.06.010.

[20] Sean P Kane and Karl Matthias. *Docker - up & running*. O'Reilly Media, Sebastopol, CA, 3 edition, May 2023. Accessed: 2024-01-05. URL: https://www.oreilly.com/library/view/docker-up/9781098131814/.

[21] Brendan Burns, Joe Beda, Kelsey Hightower, and Lachlan Evenson. *Kubernetes: Up & Running*. O'Reilly Media, 3 edition, Aug 2022. Accessed: 2024-01-01. URL: https://www.oreilly.com/library/view/kubernetes-up-and/9781098110192/.

[22] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. *Proceedings of the 10th European Conference on Computer Systems, EuroSys 2015*, Apr 2015. Accessed: 2025-01-25. doi:10.1145/2741948.2741964.

[23] Giovanni Toffetti, Tobias Lötscher, Saken Kenzhegulov, Josef Spillner, and Thomas Michael Bohnert. Cloud robotics. pages 65–70. ACM, Dec 2017. Accessed: 2023-11-02. doi:10.1145/3147234.3148100.

[24] Qirong Tang, Jingtao Zhang, Fangchao Yu, Yuan Zhang, and Zhongqun Zhang. A resource management algorithm for real-time response of mobile ad hoc cloud in swarm robotic system. In *2018 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pages 1171–1176, 2018. Accessed: 2024-01-08. doi:10.1109/ROBIO.2018.8664852.

[25] Swarnabha Roy, Dharmendra Baruah, Steven Hernandez, and Stavros Kalafatis. Distributed computation and dynamic load balancing in modular edge robotics. In *2022 Sixth IEEE International Conference on Robotic Computing (IRC)*, pages 58–61, 2022. Accessed: 2024-01-08. doi:10.1109/IRC55401.2022.00016.

[26] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement, Sep 2018. Accessed: 2024-01-08. doi:10.48550/arXiv.1804.02767.

[27] Francesco Lumpp, Franco Fummi, Hiren D. Patel, and Nicola Bombieri. Containerization and orchestration of software for autonomous mobile robots: a case study of mixed-criticality tasks across edge-cloud computing platforms. *IEEE International Conference on Intelligent Robots and Systems*, 2022:9708 – 9713, Jan 2022. Accessed: 2023-11-17. doi:10.1109/IROS47612.2022.9981581.

[28] Jiaqiang Zhang, Farhad Keramat, Xianjia Yu, Daniel Montero Hernández, Jorge Peña Queralta, and Tomi Westerlund. Distributed robotic systems in the edge-cloud continuum with ros 2: a review on novel architectures and technology readiness. In *2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 1–8, 2022. Accessed: 2024-01-08. doi:10.1109/FMEC57183.2022.10062523.

[29] Rafael Arrais, Germano Veiga, Tiago T. Ribeiro, Daniel Oliveira, Ramon Fernandes, André Gustavo S. Conceição, and P. C.M.A. Farias. Application of the open scalable production system to machine tending of additive manufacturing operations by a mobile manipulator. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11805 LNAI:345–356, 2019. Accessed: 2025-01-29. doi:10.1007/978-3-030-30244-3_29.

[30] Rafael Arrais, Paulo Ribeiro, Henrique Domingos, and Germano Veiga. Robin: An open-source middleware for plug'n'produce of cyber-physical systems. *International Journal of Advanced Robotic Systems*, 17, May 2020. Accessed: 2025-01-29. doi:10.1177/1729881420910316.