

# Multi-Layer Distributed Robotics Platform for Autonomous Mobile Robots

by

Cédric MELANÇON

MANUSCRIPT-BASED THESIS PRESENTED TO ÉCOLE DE  
TECHNOLOGIE SUPÉRIEURE  
IN PARTIAL FULFILLMENT FOR THE DEGREE OF DOCTOR IN  
ELECTRICAL ENGINEERING  
Ph.D.

MONTREAL, AUGUST 6 2026

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE  
UNIVERSITÉ DU QUÉBEC



Cédric Melançon, 2026



This Creative Commons license allows readers to download this work and share it with others as long as the author is credited. The content of this work cannot be modified in any way or used commercially.

## **BOARD OF EXAMINERS**

**THIS THESIS HAS BEEN EVALUATED**

**BY THE FOLLOWING BOARD OF EXAMINERS**

M. Maarouf Saad, Thesis supervisor  
Department of Electrical Engineering, École de technologies supérieure

Mrs. Kuljeet Kaur, Thesis Co-Supervisor  
Department of Electrical Engineering, École de technologies supérieure

M. Julien Gascon-Samson, Thesis Co-Supervisor  
Department of Software and Information Technology Engineering, École de technologie supérieure

M. Michel Rioux, Chair, Board of Examiners  
Department of System Engineering, École de technologie supérieure

M. David St-Onge, Member of the Jury  
Department of Mechanical Engineering, École de technologie supérieure

M. Aris Leivadeas, Member of the Jury  
Department of Software and Information Technology Engineering, École de technologie supérieure

M. Pierre-Yves Lajoie, External Examiner  
Department of Computer Engineering and Software Engineering, Polytechnique de Montréal

**THIS THESIS WAS PRESENTED AND DEFENDED**

**IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC**

**ON "JULY 16 2026"**

**AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE**



## ACKNOWLEDGEMENTS

This thesis was made possible by the involvement and support of several people. I would therefore like to thank and highlight the significant contributions of those who, in any way, have contributed to the smooth running and success of this research project.

First of all, I would like to thank my thesis director, Professor Maarouf Saad, who always believed in me and knew that, even 15 years later, I would return to do a PhD. Your support, both personal and academic, has always been greatly appreciated. In addition, I would like to thank my two co-supervisors. First, Professor Kuljeet Kaur, who always pushed me to excel and was invaluable in helping me understand the ins and outs of writing scientific papers. Then, Professor Julien Gascon-Samson, who, thanks to his financial support and guidance in the wonderful world of distributed systems, has enabled me to get where I am today and has always helped me understand the academic environment.

Thanks also to Professor David St-Onge for kindly lending us a Gen3-Lite for the project. I would have liked to use the manipulator more, but circumstances unfortunately prevented me from doing so.

I would like to express my gratitude to the members of the jury for the time they devoted to all stages of evaluating this thesis, which has contributed to enriching and finalizing this work.

I wish to express my recognition to André Zalzal and Patrice Dion. André spent a lot of time with me converting my good-old Pioneer 3-AT robot into a modern, solid mobile robot. It has been a long and difficult journey, but we did it! I consider myself the not-so-young Padawan of Patrice in the understanding of network configuration. I now know that there is no magic, and, above all, networking is not easy, but it is not an untamable monster.

I would also like to thank my colleagues at the Groupe de Recherche en Électronique de Puissance et Commande Industrielle (GREPCI) robotics laboratory and the Cloud-to-Edge Lab, with whom I have had the honor of sharing this wonderful adventure over the past five years. A

big thank you also to my friends who believed in me and always supported me through the good moments and the more difficult ones.

Finally, I would like to thank my family for their support and encouragement throughout my academic journey. Thanks to my three kids for their patience with me. I've always tried to shield you from my decision to return to school, but sometimes it wasn't as easy as I wished. A particular cheerful thank you to my wife, who knew how difficult it would be for me to pursue my PhD but always encouraged me to achieve my dream. Thank you for believing in me, for your patience, your sacrifices, and your support during the six difficult years going through a pandemic, multiple job challenges, and the graduation of your four men!

# **Robotique Distribuée Multicouche pour les Robots Mobiles Autonomes dans les Environnements de Santé Intelligents**

Cédric MELANÇON

## **RÉSUMÉ**

Les robots mobiles autonomes s'appuient de plus en plus sur des modèles d'apprentissage profond et sur des infrastructures informatiques distribuées couvrant les couches périphériques, de brouillard et infonuagiques. Dans des environnements tels que le secteur de la santé, ces systèmes doivent respecter des contraintes strictes de latence, de consommation d'énergie et de gestion opérationnelle, tout en demeurant sûrs et robustes. Cette thèse étudie la mise en œuvre de robots mobiles autonomes basés sur l'IA sur des plateformes hétérogènes multicouches en abordant conjointement l'exécution embarquée de modèles d'apprentissage profond, la distribution de données en temps réel et l'orchestration dynamique de services robotiques distribués. Le travail s'appuie sur un banc d'essai réaliste, combinant un robot mobile modernisé et trois clusters Kubernetes représentant les différentes couches distribuées. Un modèle d'odométrie visuelle par apprentissage profond est d'abord optimisé pour une exécution en temps réel, montrant que ces performances peuvent être atteintes sans dégrader la précision de localisation, tout en quantifiant les compromis en termes de ressources, d'énergie et de température. Un middleware de communication multicouche, BlazeFlow, est ensuite introduit pour unifier les flux de données entre des protocoles de publication-abonnement hétérogènes et assurer une communication à faible latence pour des flux de capteurs à haut débit entre la périphérie, le brouillard et le nuage. Sur cette base, BlazeAIoT est proposé comme plateforme modulaire multicouche intégrant la communication, la configuration, la surveillance et le déploiement de services robotiques, et validé au moyen de scénarios de navigation et d'arrêt d'urgence, où les services sont répartis entre des clusters périphériques de type fog et cloud, tout en maintenant des performances en temps réel. Enfin, BlazeMaestro est présenté comme un jumeau numérique qui associe un simulateur à un déploiement multi-clusters pour évaluer des stratégies d'orchestration tenant compte de contraintes de ressources, de latence et d'énergie, et pour comparer des orchestrateurs métaheuristiques, d'apprentissage par renforcement profond, gloutons et basés sur des autoencodeurs. Dans l'ensemble, cette recherche fournit des outils et des preuves expérimentales pour guider le déploiement de robots mobiles autonomes sur des infrastructures multicouches hétérogènes et clarifie comment équilibrer la latence, l'énergie et la complexité opérationnelle dans les environnements de santé intelligents.

**Mots-clés:** Distribution de données, informatique en périphérie, informatique en brouillard, infonuagique, intelligence artificielle, Internet des Objets, jumeau numérique, optimisation multi-objectifs, orchestration dynamique, robotique distribuée, et robotique mobile autonome



# Multi-Layer Distributed Robotics Platform for Autonomous Mobile Robots

Cédric MELANÇON

## ABSTRACT

Autonomous mobile robots increasingly rely on deep learning models and distributed computing infrastructures spanning edge, fog, and cloud layers. In healthcare environments, these systems must meet strict constraints regarding latency, power consumption, and operational management, while remaining safe and robust. This thesis investigates the implementation of Artificial Intelligence (AI)-based autonomous mobile robots on heterogeneous multi-layer platforms by jointly addressing the on-board execution of deep learning models, real-time data distribution, and the dynamic orchestration of distributed robotic services. The work relies on a realistic testbed, combining a modernized mobile robot and three Kubernetes clusters representing the different distributed layers. A deep learning-based visual odometry model is first optimized for real-time execution, demonstrating that such performance can be achieved without degrading localization accuracy, while quantifying the trade-offs in terms of resources, energy, and temperature. A multi-layer communication middleware, BlazeFlow, is then introduced to unify data flows between heterogeneous publish-subscribe protocols and ensure low-latency communication for high-throughput sensor streams between the edge, the fog, and the cloud. Building on this, BlazeAIoT is proposed as a modular, multi-layer platform integrating communication, configuration, monitoring, and deployment of robotic services, and validated through navigation and emergency stop scenarios, where services are distributed across fog and cloud edge clusters while maintaining real-time performance. Finally, BlazeMaestro is presented as a digital twin that combines a simulator with a multi-cluster deployment to evaluate orchestration strategies that account for resource, latency, and energy constraints, and to compare metaheuristic, deep reinforcement learning, greedy, and autoencoder-based orchestrators. Overall, this research provides tools and experimental evidence to guide the deployment of autonomous mobile robots on heterogeneous multi-layer infrastructures and clarifies how to balance latency, energy, and operational complexity in smart healthcare environments.

**Keywords:** Artificial intelligence, autonomous mobile robotics, data distribution, distributed robotics, dynamic orchestration, cloud computing, digital twin, edge computing, fog computing, Internet of Things, and multi-constraint optimization



## TABLE OF CONTENTS

|                                                                                            | Page |
|--------------------------------------------------------------------------------------------|------|
| CHAPTER 1 INTRODUCTION .....                                                               | 1    |
| 1.1 Motivation .....                                                                       | 2    |
| 1.2 Objectives .....                                                                       | 3    |
| 1.3 Methodology .....                                                                      | 4    |
| 1.3.1 Methodology for Objective 1: AI operationalization under real-time constraints ..... | 4    |
| 1.3.2 Methodology for Objective 2: Multi-layer communication middleware ....               | 5    |
| 1.3.3 Methodology for Objective 3: Distributed robotics platform .....                     | 6    |
| 1.3.4 Methodology for Objective 4: Dynamic multi-layer orchestration .....                 | 7    |
| 1.4 Contributions .....                                                                    | 8    |
| 1.5 Organization .....                                                                     | 9    |
| 1.6 Related Publications .....                                                             | 9    |
| CHAPTER 2 LITERATURE REVIEW .....                                                          | 13   |
| 2.1 Strategy .....                                                                         | 13   |
| 2.2 Autonomous Robots .....                                                                | 14   |
| 2.2.1 Use Cases .....                                                                      | 15   |
| 2.2.1.1 Home Service and Social Robots .....                                               | 15   |
| 2.2.1.2 Autonomous Vehicles .....                                                          | 15   |
| 2.2.1.3 Industrial and Logistics Robots .....                                              | 16   |
| 2.2.1.4 Assistive and Healthcare Robots .....                                              | 16   |
| 2.2.2 Concepts and Requirements .....                                                      | 17   |
| 2.2.2.1 Core Functional Capabilities .....                                                 | 17   |
| 2.2.2.2 Non-functional Requirements .....                                                  | 19   |
| 2.2.3 Synthesis and Implications .....                                                     | 19   |
| 2.3 Artificial Intelligence in Robotics .....                                              | 20   |
| 2.3.1 Localization and Sensor Fusion .....                                                 | 20   |
| 2.3.2 Perception, Prediction, and Navigation .....                                         | 21   |
| 2.3.2.1 Navigation and Path Planning .....                                                 | 21   |
| 2.3.2.2 Motion Prediction and Intent Modeling .....                                        | 23   |
| 2.3.3 Deep Reinforcement Learning and Learning Dynamics .....                              | 23   |
| 2.3.4 Real-time Operationalization .....                                                   | 24   |
| 2.3.4.1 Processing Pipeline Optimization .....                                             | 24   |
| 2.3.4.2 Model Optimization .....                                                           | 25   |
| 2.3.4.3 Distributed Deployment .....                                                       | 26   |
| 2.3.4.4 Open Challenges for Real-time AI in Robotics .....                                 | 26   |
| 2.3.5 Synthesis and Implications .....                                                     | 27   |
| 2.4 Distributed Robotics .....                                                             | 28   |
| 2.4.1 Models and Challenges .....                                                          | 28   |
| 2.4.1.1 Cloud Robotics .....                                                               | 28   |

|           |                                                                                                                         |    |
|-----------|-------------------------------------------------------------------------------------------------------------------------|----|
| 2.4.1.2   | Edge and Fog Robotics .....                                                                                             | 29 |
| 2.4.1.3   | Internet of Robotics Things .....                                                                                       | 29 |
| 2.4.1.4   | Key Challenges in Distributed Robotics .....                                                                            | 30 |
| 2.4.2     | Data Distribution and Dynamic Configuration .....                                                                       | 31 |
| 2.4.2.1   | Communication Paradigms and Middlewares .....                                                                           | 31 |
| 2.4.2.2   | Multi-layer Data Distribution .....                                                                                     | 31 |
| 2.4.2.3   | Dynamic Configuration and Adaptation .....                                                                              | 32 |
| 2.4.3     | Multi-layer Orchestration .....                                                                                         | 33 |
| 2.4.3.1   | Orchestration Concepts .....                                                                                            | 33 |
| 2.4.3.2   | Orchestration Approaches .....                                                                                          | 33 |
| 2.4.3.3   | Orchestration in Robotics .....                                                                                         | 35 |
| 2.4.3.4   | Digital Twin and Simulation for Orchestration .....                                                                     | 36 |
| 2.4.4     | Synthesis and Implications .....                                                                                        | 37 |
| 2.5       | Conclusion .....                                                                                                        | 37 |
| CHAPTER 3 | CONTEXT AND PROJECT DETAILS .....                                                                                       | 41 |
| 3.1       | Robot Modernization .....                                                                                               | 41 |
| 3.1.1     | Robot Platform .....                                                                                                    | 41 |
| 3.1.2     | Robot ROS Packages .....                                                                                                | 45 |
| 3.2       | Multi-Layer Topology .....                                                                                              | 46 |
| 3.2.1     | Edge Layer .....                                                                                                        | 47 |
| 3.2.2     | Fog Cluster .....                                                                                                       | 48 |
| 3.2.3     | Cloud Layer .....                                                                                                       | 50 |
| 3.3       | Related Capstone Projects Supervision .....                                                                             | 50 |
| 3.3.1     | Preliminary Sensor Board Extension .....                                                                                | 51 |
| 3.3.2     | Digital Twin Simulator .....                                                                                            | 52 |
| 3.3.3     | Management Application for a Dynamic Distributed Platform .....                                                         | 55 |
| 3.4       | Research Project .....                                                                                                  | 56 |
| 3.4.1     | Real-time AI .....                                                                                                      | 56 |
| 3.4.2     | Multi-Layer Data Distribution .....                                                                                     | 57 |
| 3.4.3     | Dynamic Configuration .....                                                                                             | 58 |
| 3.4.4     | Service Placement .....                                                                                                 | 58 |
| 3.5       | Conclusion .....                                                                                                        | 59 |
| CHAPTER 4 | IMPACT ANALYSIS OF ALGORITHM OPTIMIZATION ON ROBOT<br>DEEP LEARNING LOCALIZATION MODEL FOR REAL-TIME<br>EXECUTION ..... | 61 |
| 4.1       | Abstract .....                                                                                                          | 61 |
| 4.2       | Introduction .....                                                                                                      | 62 |
| 4.2.1     | Motivation .....                                                                                                        | 64 |
| 4.2.2     | Related Work .....                                                                                                      | 65 |
| 4.2.3     | Contributions .....                                                                                                     | 68 |
| 4.2.4     | Organization .....                                                                                                      | 69 |
| 4.3       | Visual Odometry Model .....                                                                                             | 70 |

|           |                                                                                                                                      |     |
|-----------|--------------------------------------------------------------------------------------------------------------------------------------|-----|
| 4.3.1     | Deployment .....                                                                                                                     | 73  |
| 4.3.2     | Model training .....                                                                                                                 | 73  |
| 4.4       | Model Operationalization .....                                                                                                       | 74  |
| 4.4.1     | Phase 1: ROS conversion .....                                                                                                        | 74  |
| 4.4.2     | Phase 2: Parallelization .....                                                                                                       | 76  |
| 4.4.3     | Phase 3: Loop unrolling .....                                                                                                        | 77  |
| 4.5       | Evaluation .....                                                                                                                     | 79  |
| 4.5.1     | Methodology .....                                                                                                                    | 79  |
| 4.5.2     | Frame Rate .....                                                                                                                     | 81  |
| 4.5.3     | Phase 1 .....                                                                                                                        | 82  |
| 4.5.4     | Phase 2 .....                                                                                                                        | 83  |
| 4.5.5     | Phase 3 .....                                                                                                                        | 84  |
| 4.5.6     | Resource Usage .....                                                                                                                 | 86  |
| 4.5.7     | Power and Temperature .....                                                                                                          | 87  |
| 4.5.8     | Discussions .....                                                                                                                    | 89  |
|           |                                                                                                                                      |     |
| CHAPTER 5 | BLAZEFLOW: A MULTI-LAYER COMMUNICATION<br>MIDDLEWARE FOR REAL-TIME DISTRIBUTED IOT<br>APPLICATIONS .....                             | 93  |
| 5.1       | Abstract .....                                                                                                                       | 93  |
| 5.2       | Introduction .....                                                                                                                   | 94  |
| 5.3       | Motivating Use Case .....                                                                                                            | 95  |
| 5.4       | System Design .....                                                                                                                  | 97  |
| 5.4.1     | Design Goals .....                                                                                                                   | 97  |
| 5.4.2     | Architecture .....                                                                                                                   | 98  |
| 5.4.2.1   | Inter-Layer Communications .....                                                                                                     | 98  |
| 5.4.2.2   | Intra-Layer Communications .....                                                                                                     | 100 |
| 5.4.2.3   | Intra-Node Communications .....                                                                                                      | 100 |
| 5.4.2.4   | General Considerations .....                                                                                                         | 100 |
| 5.4.2.5   | Local Storage .....                                                                                                                  | 101 |
| 5.5       | Evaluation .....                                                                                                                     | 101 |
| 5.5.1     | Implementation and Deployment .....                                                                                                  | 101 |
| 5.5.2     | Experimental Results .....                                                                                                           | 103 |
| 5.5.2.1   | Methodology .....                                                                                                                    | 103 |
| 5.5.2.2   | Latency .....                                                                                                                        | 106 |
| 5.5.2.3   | Bandwidth Utilization .....                                                                                                          | 107 |
| 5.5.2.4   | CPU and Memory Utilization .....                                                                                                     | 108 |
| 5.6       | Related Work .....                                                                                                                   | 109 |
| 5.7       | Conclusion .....                                                                                                                     | 110 |
|           |                                                                                                                                      |     |
| CHAPTER 6 | BLAZEAIOT: A MODULAR MULTI-LAYER PLATFORM FOR<br>REAL-TIME DISTRIBUTED ROBOTICS ACROSS EDGE, FOG,<br>AND CLOUD INFRASTRUCTURES ..... | 111 |
| 6.1       | Abstract .....                                                                                                                       | 111 |

|           |                                                                                                |     |
|-----------|------------------------------------------------------------------------------------------------|-----|
| 6.2       | Introduction .....                                                                             | 112 |
| 6.2.1     | Motivation .....                                                                               | 112 |
| 6.2.2     | Contributions .....                                                                            | 113 |
| 6.2.3     | Organization .....                                                                             | 114 |
| 6.3       | Related Work .....                                                                             | 114 |
| 6.4       | Proposed Solution .....                                                                        | 117 |
| 6.4.1     | System Overview .....                                                                          | 117 |
| 6.4.2     | BlazeAIoT SDK: A standard interface for every service .....                                    | 119 |
| 6.4.3     | BlazeConfig: Multi-layer Configuration .....                                                   | 120 |
| 6.4.4     | BlazeFlow: Multi-layer communication .....                                                     | 120 |
| 6.4.5     | BlazeMonitoring: System Monitoring .....                                                       | 121 |
| 6.5       | BlazeFlow as a Dynamic Data Bridge Solution .....                                              | 122 |
| 6.5.1     | Dynamic Data Distribution .....                                                                | 122 |
| 6.5.1.1   | Bridge Creation Example .....                                                                  | 123 |
| 6.5.1.2   | One-to-many Scenario Example .....                                                             | 124 |
| 6.5.2     | Network Considerations .....                                                                   | 126 |
| 6.5.3     | Adaptive Data Distribution .....                                                               | 126 |
| 6.6       | Evaluation Testbed .....                                                                       | 128 |
| 6.6.1     | Infrastructure .....                                                                           | 128 |
| 6.6.1.1   | Cloud cluster .....                                                                            | 129 |
| 6.6.1.2   | Fog cluster .....                                                                              | 130 |
| 6.6.1.3   | Edge cluster .....                                                                             | 131 |
| 6.6.2     | Robot .....                                                                                    | 132 |
| 6.6.2.1   | Robot Localization Service .....                                                               | 132 |
| 6.6.2.2   | Pure Pursuit Service .....                                                                     | 134 |
| 6.6.2.3   | Vision-Based Safety Stop Service .....                                                         | 134 |
| 6.7       | Experimental Results .....                                                                     | 135 |
| 6.7.1     | Scenario 1: Navigation .....                                                                   | 135 |
| 6.7.2     | Scenario 2: Vision-based safety stop .....                                                     | 139 |
| 6.8       | Discussion .....                                                                               | 141 |
| 6.8.1     | BlazeAIoT vs. Objectives .....                                                                 | 142 |
| 6.9       | Conclusion .....                                                                               | 143 |
|           |                                                                                                |     |
| CHAPTER 7 | BLAZEMAESTRO: A DIGITAL-TWIN FRAMEWORK FOR<br>DISTRIBUTED ROBOTICS DYNAMIC ORCHESTRATION ..... | 145 |
| 7.1       | Abstract .....                                                                                 | 145 |
| 7.2       | Introduction .....                                                                             | 146 |
| 7.3       | Environments .....                                                                             | 150 |
| 7.3.1     | Digital Twin Overview .....                                                                    | 151 |
| 7.3.2     | Simulation Environment .....                                                                   | 152 |
| 7.3.2.1   | Network jitter .....                                                                           | 153 |
| 7.3.2.2   | Network Congestion .....                                                                       | 153 |
| 7.3.2.3   | Energy .....                                                                                   | 154 |

|     |         |                                         |     |
|-----|---------|-----------------------------------------|-----|
|     | 7.3.2.4 | WiFi Mobility .....                     | 155 |
|     | 7.3.3   | Real Environment .....                  | 155 |
| 7.4 |         | Dynamic Orchestration .....             | 157 |
|     | 7.4.1   | Constraints .....                       | 157 |
|     | 7.4.1.1 | CPU, RAM, and GPU Capacity .....        | 158 |
|     | 7.4.1.2 | Period and Dependency Constraint .....  | 159 |
|     | 7.4.1.3 | Service Locks and Soft Objectives ..... | 160 |
|     | 7.4.2   | Orchestrator Justification .....        | 161 |
|     | 7.4.2.1 | NSGA-II Orchestrator .....              | 162 |
|     | 7.4.2.2 | DQN Orchestrator .....                  | 163 |
|     | 7.4.2.3 | Greedy Weighted Orchestrator .....      | 165 |
|     | 7.4.2.4 | Autoencoder Orchestrator .....          | 167 |
|     | 7.4.3   | Reallocation Process .....              | 170 |
| 7.5 |         | Evaluation .....                        | 171 |
|     | 7.5.1   | Training Scenarios .....                | 172 |
|     | 7.5.2   | Simulation Evaluation .....             | 172 |
|     | 7.5.3   | Experimental Evaluation .....           | 174 |
| 7.6 |         | Discussion .....                        | 177 |
| 7.7 |         | Conclusion .....                        | 180 |
|     |         | CONCLUSION AND RECOMMENDATIONS .....    | 181 |
| 8.1 |         | Contributions .....                     | 181 |
| 8.2 |         | Recommendations .....                   | 183 |
| 8.3 |         | Future Research .....                   | 185 |
| 8.4 |         | Synthesis .....                         | 186 |
|     |         | BIBLIOGRAPHY .....                      | 189 |



## LIST OF TABLES

|           | Page                                                                                        |
|-----------|---------------------------------------------------------------------------------------------|
| Table 2.1 | Summary of representative works reviewed in Chapter 2 ..... 39                              |
| Table 3.1 | Edge cluster nodes ..... 48                                                                 |
| Table 3.2 | Fog cluster nodes ..... 49                                                                  |
| Table 3.3 | Cloud cluster nodes ..... 51                                                                |
| Table 4.1 | Real-time model references ..... 66                                                         |
| Table 4.2 | Number of frames executed over the sequence by configuration and development phase ..... 82 |
| Table 4.3 | Submodels execution times (in ms) per configuration on development phase 3 ..... 84         |
| Table 4.4 | Jetson power profile configuration ..... 86                                                 |
| Table 4.5 | Resource usage of each configuration by development phase ..... 86                          |
| Table 4.6 | GPU Temperature in °C for each configuration and development phase .. 88                    |
| Table 4.7 | GPU Power in Watts for each configuration and development phase ..... 89                    |
| Table 4.8 | Total power for each configuration and development phase ..... 90                           |
| Table 4.9 | Execution Time Improvement Across Optimization Phases ..... 90                              |
| Table 5.1 | Data definition of the robot ..... 96                                                       |
| Table 6.1 | Cloud cluster nodes ..... 130                                                               |
| Table 6.2 | Fog cluster nodes ..... 131                                                                 |
| Table 6.3 | Edge cluster nodes ..... 132                                                                |
| Table 6.4 | ROS topic latency ..... 137                                                                 |
| Table 6.5 | Emergency stop metrics ..... 140                                                            |
| Table 7.1 | Constraint Definition for Dynamic Orchestration ..... 158                                   |
| Table 7.2 | Simulation Test Results ..... 173                                                           |

|           |                                                  |     |
|-----------|--------------------------------------------------|-----|
| Table 7.3 | Experimental Test Results (Hysteresis: 0%) ..... | 176 |
|-----------|--------------------------------------------------|-----|

## LIST OF FIGURES

|                                                                                    | Page |
|------------------------------------------------------------------------------------|------|
| Figure 3.1      Robot Hardware Architecture .....                                  | 42   |
| Figure 3.2      P3AT control board .....                                           | 43   |
| Figure 3.3      Modernized P3AT .....                                              | 44   |
| Figure 3.4      P3AT Software Components .....                                     | 45   |
| Figure 3.5      High-level architecture of the Distributed Robotics Solution ..... | 46   |
| Figure 3.6      High-level Network Topology .....                                  | 47   |
| Figure 3.7      Edge Layer Topology .....                                          | 48   |
| Figure 3.8      Fog Layer Topology .....                                           | 49   |
| Figure 3.9      Cloud Layer Topology .....                                         | 50   |
| Figure 3.10     Altium PCB Layout .....                                            | 51   |
| Figure 3.11     Preliminary Robot Modernization .....                              | 52   |
| Figure 3.12     Robot Simulation with Vortex and Unity .....                       | 53   |
| Figure 3.13     Scene Creation from Robot Sensor Data .....                        | 54   |
| Figure 3.14     Scene with Textures .....                                          | 54   |
| Figure 3.15     BlazeAIoT Management User Interface .....                          | 55   |
| Figure 3.16     Scene Creation from Robot Sensor Data .....                        | 56   |
| Figure 4.1      Deep Learning Visual Odometry Model .....                          | 70   |
| Figure 4.2      Encoder Model .....                                                | 71   |
| Figure 4.3      Transition Model .....                                             | 71   |
| Figure 4.4      Pose Model .....                                                   | 72   |
| Figure 4.5      Phase 1 Sequence Diagram .....                                     | 75   |
| Figure 4.6      Model Execution Sequence Diagram .....                             | 77   |

|             |                                                             |     |
|-------------|-------------------------------------------------------------|-----|
| Figure 4.7  | Phase 3 Sequence Diagram .....                              | 78  |
| Figure 4.8  | Model Execution Time for Each Configuration and Phase ..... | 83  |
| Figure 4.9  | Phase 3 Execution Time for Two Jetson Power Profiles .....  | 85  |
| Figure 5.1  | Data Distribution Framework Architecture .....              | 99  |
| Figure 5.2  | Data Distribution Evaluation Solution .....                 | 102 |
| Figure 5.3  | Redis Message Latency .....                                 | 103 |
| Figure 5.4  | MQTT Message Latency .....                                  | 103 |
| Figure 5.5  | Kafka Message Latency .....                                 | 104 |
| Figure 5.6  | Edge Incoming and Outgoing Network Bandwidth .....          | 104 |
| Figure 5.7  | Fog Incoming Network Bandwidth .....                        | 105 |
| Figure 5.8  | Fog Outgoing Network Bandwidth .....                        | 105 |
| Figure 5.9  | Edge CPU/Memory Metrics .....                               | 106 |
| Figure 5.10 | Fog CPU Usage .....                                         | 106 |
| Figure 5.11 | Fog Memory Usage .....                                      | 107 |
| Figure 6.1  | BlazeAIoT High-level Architecture .....                     | 118 |
| Figure 6.2  | Multi-layer Handshaking Sequence .....                      | 123 |
| Figure 6.3  | Multi-bridge Scenario Example .....                         | 125 |
| Figure 6.4  | Experimental Server Infrastructure .....                    | 129 |
| Figure 6.5  | Modified Pioneer 3-AT .....                                 | 133 |
| Figure 6.6  | Safety Stop Detection Model .....                           | 135 |
| Figure 6.7  | Latency Distribution .....                                  | 136 |
| Figure 6.8  | Spatial map .....                                           | 138 |
| Figure 6.9  | Total Distance Error .....                                  | 139 |
| Figure 7.1  | BlazeMaestro Digital Twin Architecture .....                | 151 |

|            |                                                        |     |
|------------|--------------------------------------------------------|-----|
| Figure 7.2 | BlazeMaestro Architecture Within BlazeAIoT .....       | 157 |
| Figure 7.3 | Simulation Latency and Energy over Scenarios I-V ..... | 175 |
| Figure 7.4 | Experimental Instantaneous Energy Consumption .....    | 178 |



## LIST OF ALGORITHMS

|                                                            | Page |
|------------------------------------------------------------|------|
| Algorithm 7.1      WiFi Handover for Mobile Robot .....    | 156  |
| Algorithm 7.2      Greedy Weighted Service Placement ..... | 166  |



## LIST OF ABBREVIATIONS

|        |                                                                           |
|--------|---------------------------------------------------------------------------|
| AI     | Artificial Intelligence                                                   |
| AIoT   | Augmented Intelligence with Internet of Things                            |
| CNN    | Convolutional Neural Network                                              |
| CPS    | Cyber-Physical Systems                                                    |
| CPU    | Central Processing Unit                                                   |
| DDS    | Data Distribution Service                                                 |
| DL     | Deep Learning                                                             |
| DRL    | Deep Reinforcement Learning                                               |
| E2C    | Edge-to-Cloud                                                             |
| EKF    | Extended Kalman Filter                                                    |
| E2C    | Edge-to-Cloud                                                             |
| ETS    | École de Technologie Supérieure                                           |
| GPS    | Global Positioning System                                                 |
| GPIO   | General-Purpose Input/Output                                              |
| GPU    | Graphics Processing Unit                                                  |
| GREPCI | Groupe de Recherche en Électronique de Puissance et Commande Industrielle |
| GRU    | Gated Recurrent Units                                                     |
|        | IaaS Infrastructure-as-a-Service                                          |
| IMU    | Inertial Measurement Unit                                                 |

|         |                                            |
|---------|--------------------------------------------|
| IoRT    | Internet of Robotic Things                 |
| IoT     | Internet of Things                         |
| LiDAR   | Light Detection and Ranging                |
| LSTM    | Long Short Term Memory                     |
| MLP     | Multilayer Perceptron                      |
| MQTT    | Message Queuing Telemetry Transport        |
| NSGA-II | Non-dominated Sorting Genetic Algorithm II |
| NTP     | Network Time Protocol                      |
| ORB     | Oriented Fast and Rotated Brief            |
| P3AT    | Pioneer 3AT                                |
| PaaS    | Platform-as-a-Service                      |
| PCB     | Printed Circuit Board                      |
| PID     | Proportional-Integral-Derivative           |
| QoS     | Quality of Service                         |
| RaaS    | Robot-as-a-Service                         |
| RGB     | Red Green Blue                             |
| RAM     | Random-Access Memory                       |
| RPC     | Remote Procedure Calls                     |
| RNN     | Recurrent Neural Networks                  |
| ROS     | Robotics Operating System                  |

|      |                                       |
|------|---------------------------------------|
| SLAM | Simultaneous Localization and Mapping |
| SNR  | Signal-to-Noise Ratio                 |
| SPI  | Serial Peripheral Interface           |
| SSH  | Secure Shell                          |
| USB  | Universal Serial Bus                  |
| UWB  | Ultra-Wideband                        |
| V2X  | Vehicle-to-Everything                 |
| YOLO | You Only Look Once                    |



## LIST OF SYMBOLS AND UNITS OF MEASUREMENTS

|                        |                                                                 |
|------------------------|-----------------------------------------------------------------|
| $\alpha_m$             | Message overhead factor                                         |
| $\alpha_r$             | Active power scaling coefficient for the CPU/GPU usage term (W) |
| $\alpha_q$             | Energy efficiency factor $1/(1 + 0.1 E_q)$                      |
| $\beta_b$              | Per-message maximum bandwidth (%)                               |
| $\beta$                | Path-loss exponent                                              |
| $\Delta t$             | Elapsed time (s)                                                |
| $\delta_s$             | Dependency indicator for service $s$                            |
| $\gamma$               | Discount factor                                                 |
| $\lambda$              | Latent regularization coefficient                               |
| $\sigma_j$             | Base jitter amplitude (%)                                       |
| $\tau_i$               | Period slack $T_i - t_{\text{exec},i}$ (s)                      |
| $\rho_i$               | Normalized slack $\tau_i/T_i$                                   |
| $\rho_{ij}(t)$         | Network load on link $(i, j)$ at time $t$                       |
| $\mathcal{A}_s$        | Feasible actions (candidate nodes) for service $s$ at state $o$ |
| $b_{n_j}$              | Battery flag for node $n_j$                                     |
| $B_{\text{remaining}}$ | Remaining bandwidth (bytes/s)                                   |
| $B_{\text{avail}}$     | Available bandwidth (bytes/s)                                   |
| $B_{s,n}$              | CPU/RAM balance cost on node $n$                                |
| $B_t$                  | Battery capacity at time $t$ (J)                                |

XXX

|                                                  |                                                                                                                         |
|--------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------|
| $C$                                              | Token capacity                                                                                                          |
| $c_s$                                            | Real-time criticality flag for service $s$                                                                              |
| $C_n^r$                                          | Capacity of node $n$ for resource $r$ (CPU cores for CPU, GB for RAM/GPU)                                               |
| $C_r(n)$                                         | Resource $r$ capacity of node $n$ (CPU cores for CPU, GB for RAM/GPU)                                                   |
| $C_{\text{tot}}, R_{\text{tot}}, G_{\text{tot}}$ | Total CPU/RAM/GPU capacity across all nodes (CPU cores for $C_{\text{tot}}$ , GB for $R_{\text{tot}}, G_{\text{tot}}$ ) |
| $\bar{C}, \bar{R}, \bar{G}$                      | Mean CPU/RAM/GPU capacity per node (CPU cores for $\bar{C}$ , GB for $\bar{R}, \bar{G}$ )                               |
| $\text{CPU}_{\text{eff}}(s)$                     | Effective CPU utilization of service $s$ (CPU cores)                                                                    |
| $\text{CPU}_s, \text{RAM}_s, \text{GPU}_s$       | Resource demands of service $s$ (CPU cores, GB, GB)                                                                     |
| $d, d_{\text{max}}$                              | Distance; maximum WiFi range (m)                                                                                        |
| $D_i$                                            | Bandwidth demand (bytes/s)                                                                                              |
| $D(s)$                                           | Direct dependencies of service $s$                                                                                      |
| $E_{\text{drain}}(t)$                            | Energy drain over time step $\Delta t$ (J)                                                                              |
| $E_{s,n}$                                        | Incremental energy cost of placing $s$ on node $n$ (J)                                                                  |
| $E_q$                                            | Total estimated energy cost of training sample $q$ (J)                                                                  |
| $f_s^{\text{svc}}$                               | Feature vector for service $s$                                                                                          |
| $f^{\text{net}}$                                 | Aggregated network feature vector                                                                                       |
| $f^{\text{energy}}$                              | Per-node energy features (battery flag, power)                                                                          |
| $\mathcal{H}(U, y)$                              | Per-service cross-entropy reconstruction loss                                                                           |

|                                                          |                                                                  |
|----------------------------------------------------------|------------------------------------------------------------------|
| $J(i, j, t)$                                             | Jitter factor on link $(i, j)$ at time $t$                       |
| $\ell_{s,n}^{\min}, \ell_{s,n}^{\max}, \bar{\ell}_{s,n}$ | Min/max/mean dependency latency for service $s$ at node $n$ (s)  |
| $L_{\text{client}}$                                      | Client Bandwidth limit (Mbps)                                    |
| $L_{\text{base}}(i, j)$                                  | Base latency on link $(i, j)$ (s)                                |
| $L_{\text{actual}}(i, j, t)$                             | Link latency with jitter (s)                                     |
| $L_{\text{total}}(i, j, t)$                              | Total link latency (jitter + congestion) (s)                     |
| $L_{\text{cong}}(\rho)$                                  | Congestion latency multiplier                                    |
| $L_{\text{dep}}(s, n)$                                   | Transitive dependency latency (s)                                |
| $L_{\text{edge}}(d, n)$                                  | Link latency from dependency $d$ to node $n$ (s)                 |
| $L_{\text{crit}}$                                        | Critical-path latency through the service graph (s)              |
| $L_{\text{deps},n}$                                      | Transitive dependency latency for $s$ if placed on $n$ (s)       |
| $\mathcal{L}_{\text{AE}}$                                | Autoencoder training loss                                        |
| $\bar{L}, L_{\text{max}}$                                | Mean and maximum inter-node latency (s)                          |
| $\mathcal{L}$                                            | DQN training loss                                                |
| $m_{\text{link}}$                                        | Link-type jitter multiplier                                      |
| $n, i, j$                                                | Node or link endpoint indices                                    |
| $n^{\star}$                                              | Selected node for service $s$                                    |
| $\mathcal{N}_s$                                          | Feasible nodes for service $s$ (satisfying all hard constraints) |
| $\mathcal{N}, \mathcal{S}$                               | Set of all available nodes/services                              |
| $ \mathcal{N} ,  \mathcal{S} $                           | Number of nodes/services                                         |

|                                    |                                                                    |
|------------------------------------|--------------------------------------------------------------------|
| $L_s$                              | Node logit vector for service $s$ (decoder output)                 |
| $M_i$                              | Message size (bytes)                                               |
| $o$                                | Environment state (features used by the DQN at each decision step) |
| $p_{n_j}$                          | Normalized active power of node $n_j$                              |
| $P_{\text{idle}}$                  | Node idle power (W)                                                |
| $P_{\text{active}}$                | Node active power (W)                                              |
| $P(o' \mid o, a)$                  | Transition kernel                                                  |
| $\text{place}(d)$                  | Node where dependency $d$ is placed                                |
| $Q_\theta(o, a)$                   | Q-value, main network parameters $\theta$                          |
| $Q_{\bar{\theta}}(o, a)$           | Q-value, target network parameters $\bar{\theta}$                  |
| $r$                                | Resource list CPU, GPU                                             |
| $r_i^{\text{alloc}}$               | Advertised rate (Hz)                                               |
| $r_{\text{step}}, r_{\text{term}}$ | Step and terminal reward components                                |
| $R(o, a, o')$                      | Reward function                                                    |
| $s, s_i$                           | Service index or identifier                                        |
| $S, S_{\text{max}}$                | Signal strength/maximum signal strength (dBm)                      |
| $S_n$                              | Services assigned to node $n$                                      |
| $\text{score}_{s,n}$               | Placement score for service $s$ on node $n$                        |
| $\hat{t}_{s,n}$                    | Estimated execution time of service $s$ on node $n$ (s)            |
| $T(t)$                             | Token count at time $t$                                            |

|                                                                    |                                                                                   |
|--------------------------------------------------------------------|-----------------------------------------------------------------------------------|
| $t, \Delta t$                                                      | Simulation time/time step (s)                                                     |
| $t_{\text{exec}}(s)$                                               | Execution time of service $s$ (s)                                                 |
| $T_{\text{period}}(s)$                                             | Period or deadline of service $s$ (s)                                             |
| $T_s$                                                              | Period of service $i$ (s)                                                         |
| $u(t)$                                                             | CPU utilization at time $t$                                                       |
| $w_{\text{res}}, w_{\text{per}}, w_{\text{dep}}, w_{\text{slack}}$ | Step reward weights                                                               |
| $v_{\text{res}}$                                                   | Number of resource constraint violations                                          |
| $\mathcal{V}$                                                      | Set of remaining hard-constraint violations in a final placement                  |
| $w_{\text{viol}}, w_{\text{lat}}$                                  | Terminal reward weights                                                           |
| $w_E, w_L, w_B$                                                    | Weights for energy, latency, and balance terms                                    |
| $w_q$                                                              | Sample weight (energy-efficiency and feasibility-aware)                           |
| $x$                                                                | Input state vector (services, network, energy features)                           |
| $y$                                                                | Target Q-value (DQN) / Ground-truth placement encoding (one-hot)<br>(Autoencoder) |
| $z_s$                                                              | Per-service latent representation                                                 |



## CHAPTER 1

### INTRODUCTION

Since the COVID-19 pandemic, the labor shortage and demographic shifts have worsened across many sectors, including manufacturing, logistics, and public services, with particularly severe effects in Quebec's public healthcare system as exposed by Legault Thuot (2024). Employees frequently work overtime to maintain service quality, and based on Institut de la statistique du Québec (2026), demographic trends suggest that demand for care and other essential services will continue to increase while the working-age population shrinks. This structural imbalance motivates technological solutions that can support workers, improve efficiency, and help maintain quality and safety in smart environments.

Over the past years, autonomous robots have emerged as a promising approach to alleviating labor shortages and supporting human workers in healthcare, industrial production, logistics, and service robotics, as studied by Christoforou *et al.* (2020) and Chinmay & Nisha (2024). Nursing and assistive robots can provide socially-assistive support, physically-assistive capabilities, and telerobotics, while other systems can perform invasive or non-invasive tasks such as phlebotomy, logistics, guidance, and interaction in hospitals and long-term care facilities Blechar *et al.* (2019). In parallel, autonomous mobile robots are increasingly deployed for indoor logistics, warehouse operations, and smart manufacturing, where they transport goods, inspect equipment, and operate in close proximity to humans and other cyber-physical systems, as exposed by Hachijo *et al.* (2024) and Kaneko *et al.* (2020). Realizing a fully autonomous robot capable of safely executing such tasks in these heterogeneous environments requires multiple Artificial Intelligence (AI) models, including perception, localization, navigation, and human-robot interaction components, which in turn demand substantial computational resources and energy.

Battery-powered mobile robots have limited onboard resources and energy autonomy, making it difficult to host all computationally demanding workloads locally without compromising mission

duration or increasing cost, weight, and thermal complexity (Hu *et al.* (2012)). Distributed robotics has therefore emerged as a way to offload parts of the computation to external infrastructures. Cloud robotics, introduced by Hu *et al.* (2012), first exploited centralized cloud resources for intensive processing and data storage, but introduced latency, reliability, and trust issues for time-critical control loops. Subsequent work on edge (Dey & Mukherjee (2016)) and fog (Leela *et al.* (2017)) robotics sought to balance latency, compute capacity, and privacy by distributing workloads across edge, fog, and cloud layers.

## 1.1 Motivation

Despite these advances, current distributed robotics solutions exhibit several limitations when applied to autonomous mobile robots operating in safety-critical and time-sensitive smart environments, such as hospitals, factories, and warehouses. In such contexts, robots must satisfy strict real-time control-loop requirements, respect energy and resource constraints, and integrate with heterogeneous communication protocols and computing platforms, while remaining robust to dynamic wireless links and variable workloads (Samara *et al.* (2022); Dordevic *et al.* (2023)).

This thesis identifies three gaps that present important limitations for these scenarios:

- **Gap I:** Many AI models used for perception and localization remain difficult to deploy under real-time constraints on resource-limited platforms, especially when considering energy, temperature, and multi-hardware deployment trade-offs.
- **Gap II:** Existing multi-layer communication middlewares often lack unified support for heterogeneous protocols and do not systematically address high-frequency, high-volume data flows across edge, fog, and cloud while preserving latency and throughput.
- **Gap III:** While orchestration frameworks exist for cloud-to-thing continuum, few provide digital-twin-driven, constraint-aware multi-objective orchestration tailored to mobile robots operating over dynamic wireless links and under strict timing and energy constraints.

These gaps motivate an integrated approach to distributed autonomous robotics that jointly considers the following aspects:

1. AI model operationalization under real-time and energy constraints.
2. Multi-layer communication and data distribution.
3. Dynamic orchestration of services across edge, fog, and cloud infrastructures.

The general goal of this thesis is to design, implement, and evaluate a complete platform and methodology that enables safe and efficient deployment of AI-enabled autonomous robots over heterogeneous distributed resources, with a particular emphasis on smart environments such as healthcare facilities, industrial plants, and indoor logistics hubs.

## 1.2 Objectives

This thesis addresses the following objectives:

- **AI operationalization under real-time constraints:** Evaluate and optimize a deep learning (DL) visual odometry model for robot localization to meet strict real-time control-loop requirements without degrading accuracy, and quantify its impact on energy consumption, temperature, and resource usage across edge, fog, and cloud hardware.
- **Multi-layer communication middleware:** Design and implement a multi-layer communication middleware that bridges heterogeneous protocols (e.g., ROS2, DDS, MQTT, Kafka, Redis) and sustains high-frequency, high-volume robotic data across edge, fog, and cloud layers, enabling flexible service placement and migration.
- **Distributed robotics platform:** Develop a modular platform that unifies edge, fog, and cloud robotics in a single service-oriented architecture, providing dynamic data distribution, configuration, and monitoring to support real-time distributed robotics applications over heterogeneous infrastructures.

- **Dynamic multi-layer orchestration:** Propose and evaluate a digital-twin-based orchestration framework that formulates multi-objective service placement for autonomous mobile robots under resource, timing, and energy constraints, and compare multiple orchestration paradigms (metaheuristics, deep reinforcement learning, heuristics, and autoencoder-based methods)

### 1.3 Methodology

To address these objectives, the thesis adopts an experimental methodology based on a realistic distributed robotics setup. A mobile robot equipped with multiple sensors (*e.g.*, Light Detection and Ranging (LiDAR), cameras, Inertial Measurement Unit (IMU)) operates within the GREPCI Robotics laboratory at École de technologie supérieure, while three Kubernetes-based clusters represent the edge, fog, and cloud layers. The infrastructure supports deploying AI-based localization models, communication middlewares, and orchestration services across layers, and includes tooling for collecting resource usage, timing, energy, and temperature metrics.

The following subsections describe how each of the four objectives introduced in Section 1.2 is addressed by the proposed methodology.

#### 1.3.1 Methodology for Objective 1: AI operationalization under real-time constraints

Objective 1 aims to evaluate and optimize a DL visual odometry model for robot localization so that it satisfies strict real-time control-loop requirements without degrading accuracy, while quantifying its impact on energy consumption, temperature, and resource usage across heterogeneous hardware platforms.

To address this objective, a state-of-the-art DL visual odometry model is selected and integrated into the robot software stack for on-board execution on GPU-enabled platforms. The model is first ported to the ROS ecosystem and encapsulated as a reusable localization service, enabling

seamless deployment on the edge cluster and on embedded devices hosted on the robot. The methodology then follows a three-phase optimization process:

1. ROS integration and baseline deployment
2. Parallelization of critical submodules
3. Loop unrolling and implementation-level optimizations targeting the GPU.

For each phase and hardware configuration, detailed measurements are collected, including end-to-end execution time, per-submodule latency, frame rate, CPU and GPU utilization, memory usage, temperature, and power consumption. Accuracy is evaluated using standard trajectory error metrics over representative sequences. The resulting analysis quantifies how implementation changes affect real-time performance and resource/energy trade-offs across embedded and server-grade platforms. This methodology is detailed in Chapter 4, which corresponds to the visual odometry optimization study.

### **1.3.2 Methodology for Objective 2: Multi-layer communication middleware**

Objective 2 focuses on designing and implementing a multi-layer communication middleware that bridges heterogeneous protocols (*e.g.*, ROS2, DDS, MQTT, Kafka, Redis) and sustains high-frequency, high-volume robotic data streams across edge, fog, and cloud layers, thereby enabling flexible service placement and migration.

To address this objective, a set of representative data flows is derived from the mobile robot use case, including high-rate LiDAR and camera topics, localization outputs, and control-related messages. Based on these flows, the proposed middleware is designed around a common bridging abstraction that unifies intra-node, intra-layer, and inter-layer communications. The architecture specifies how bridges are instantiated, how topics are mapped across protocols, and how data paths can be dynamically reconfigured to support different deployment scenarios.

The middleware is then implemented and deployed on the three Kubernetes clusters and on the robot. Experimental scenarios are defined to stress the system with varying message rates, payload sizes, and communication paths (intra-node, intra-layer, and inter-layer). For each scenario, the evaluation collects end-to-end latency, throughput, bandwidth utilization, CPU and memory usage, and message loss under different loads. These experiments characterize the ability of the middleware to sustain high-frequency data streams with low latency and controlled resource overhead. The methodology and results for this objective are presented in Chapter 5, which reproduces the BlazeFlow study.

### **1.3.3 Methodology for Objective 3: Distributed robotics platform**

Objective 3 aims to develop a modular platform that unifies edge, fog, and cloud robotics in a single service-oriented architecture, providing dynamic data distribution, configuration, and monitoring to support real-time distributed robotics applications over heterogeneous infrastructures.

To this end, a Kubernetes-based multi-layer platform is designed that integrates several core services:

1. A standardized SDK for packaging robotic services;
2. A configuration subsystem for declaring service graphs and deployment constraints;
3. A communication component based on the multi-layer middleware from Objective 2
4. A monitoring stack that continuously collects metrics from the robot and the infrastructure.

The mobile robot is integrated as an edge node exposing localization, navigation (pure pursuit), and vision-based safety stop services, which interact with fog and cloud services for higher-level processing.

The methodology validates the platform using end-to-end scenarios inspired by autonomous navigation in a smart healthcare environment. These scenarios include navigation tasks where

the robot follows predefined trajectories while offloading parts of the computation to fog and cloud nodes, as well as safety stop situations triggered by AI-based perception modules. For each scenario, the platform is exercised under realistic loads and network conditions, and metrics such as latency between components, service availability, fault tolerance, and responsiveness of the safety mechanisms are recorded. This methodology demonstrates how the platform supports modular deployment and reconfiguration of distributed robotic applications and is detailed in Chapter 6, which presents the BlazeAIoT platform.

#### **1.3.4 Methodology for Objective 4: Dynamic multi-layer orchestration**

Objective 4 proposes and evaluates a digital-twin-based orchestration framework that formulates multi-objective service placement for autonomous mobile robots under resource, timing, and energy constraints, and compares multiple orchestration paradigms, including metaheuristics, deep reinforcement learning, greedy heuristics, and autoencoder-based methods.

To address this objective, a digital twin is constructed that couples a realistic network and energy simulator with the live multi-cluster Kubernetes deployment used for the previous objectives. The twin models the robot, its service pipeline (localization, perception, navigation, safety services), the edge-fog-cloud infrastructure, and dynamic wireless links subject to jitter, congestion, and mobility-induced variations. Scenarios are defined that combine different robot trajectories, background traffic patterns, and battery levels, thereby exposing the orchestration algorithms to a wide range of operating conditions.

Within this environment, several orchestrators are implemented: a multi-objective NSGA-II optimizer, a Deep Q-learning (DQN) agent, a greedy weighted heuristic, and an autoencoder-based approach that learns compact service representations. The orchestration problem is formulated with hard constraints on CPU, RAM, GPU capacity, service periods, and dependencies, and soft objectives on latency, energy consumption, and resource balance. During experiments,

the orchestrators repeatedly reallocate services in response to changes in network conditions, load, and energy state. The evaluation collects metrics such as end-to-end latency along the critical service paths, total energy consumption, number and cost of reallocations, and constraint violation rates across simulation and real-world trials. This methodology is detailed in Chapter 7, which presents the BlazeMaestro digital twin and orchestration framework.

## 1.4 Contributions

The main contributions of this thesis are:

- A thorough experimental analysis of algorithmic optimization for a DL visual odometry model, demonstrating that targeted implementation changes can achieve real-time execution while preserving model structure and accuracy, and quantifying the impact on energy, temperature, and resource usage across heterogeneous hardware.
- BlazeFlow, a multi-layer communication middleware that unifies intra-node, intra-layer, and inter-layer data flows through a common bridging abstraction, enabling low-latency, high-throughput communication across heterogeneous publish/subscribe protocols for autonomous robotics workloads.
- BlazeAIoT, a modular, Kubernetes-based multi-layer platform for distributed robotics that supports dynamic data transfer, configurable services, integrated monitoring, and robust deployment across edge-fog-cloud topologies, validated on navigation and AI-driven scenarios.
- BlazeMaestro, a digital-twin framework that couples a simulator with a real multi-cluster Kubernetes deployment to support constraint-aware, multi-objective orchestration of robotic service pipelines, and a comparative evaluation of metaheuristic, deep reinforcement learning, greedy, and autoencoder-based orchestrators under realistic dynamics.

## 1.5 Organization

Chapter 2 presents the literature review supporting the thesis, covering autonomous robots and AI models, distributed robotics architectures, multi-layer communication, and orchestration for edge-fog-cloud systems. Chapter 3 provides an overview of the project and explains how the four research papers contribute to the thesis’s objectives and relate to one another. Chapters 4-7 reproduce the four core papers with minor adaptations. Chapter 4 investigates the operationalization and optimization of a DL visual odometry model for real-time robot localization, following a three-phase optimization process and evaluating execution time, resource usage, temperature, and energy across multiple hardware configurations. Chapter 5 presents BlazeFlow, a multi-layer communication middleware that unifies intra-node, intra-layer, and inter-layer data flows through a common bridging abstraction designed to sustain high-rate sensor streams across edge, fog, and cloud with low latency and controlled resource overhead. Chapter 6 introduces BlazeAIoT, a modular Kubernetes-based multi-layer platform that unifies communication, configuration, monitoring, and service management for distributed robotics, and validates it through navigation and AI-driven message-processing scenarios that span edge, fog, and cloud components. Chapter 7 presents BlazeMaestro, a digital-twin framework that couples a network and energy simulator with a live multi-cluster Kubernetes deployment to study dynamic orchestration of robotic services as a multi-objective placement problem under resource, timing, and energy constraints.

## 1.6 Related Publications

This research resulted in the following published and accepted research papers. Journal publications are prefixed with J, while conference proceedings are prefixed with C.

J1: **C Melancon**, M Saad, K Kaur, J Gascon-Samson, "Impact Analysis of Algorithm Optimization on Robot Deep Learning Localization Model for Real-Time Execution" *IEEE Access*.

C1: **C Melançon**, G Simard, M Saad, K Kaur, J Gascon-Samson, "BlazeFlow: a Multi-Layer Communication Middleware for Real-Time Distributed IoT Applications" *Mid4CC 2023 - Proceedings of the 2023 1st International Workshop on Middleware for the Computing Continuum, Part of: Middleware 2023*.

The following paper was submitted to the Elsevier Robotics and Autonomous Systems journal on december 2025 and published as a pre-print on arXiv in January 2026.

J2: **C Melancon**, J Gascon-Samson, M Saad, K Kaur, S Savard, "BlazeAIoT: A Modular Multi-Layer Platform for Real-Time Distributed Robotics Across Edge, Fog, and Cloud Infrastructures".

The following paper was submitted to the IEEE Transactions on Parallel and Distributed Systems Journal on April 2026.

J3: **C Melancon**, J Gascon-Samson, M Saad, K Kaur, S Savard, "BlazeAIoT: A Modular Multi-Layer Platform for Real-Time Distributed Robotics Across Edge, Fog, and Cloud Infrastructures."BlazeMaestro: A Digital-Twin Framework for Distributed Robotics Dynamic Orchestration".

Other complementary research papers were also published during the project, but are not explicitly included in this thesis.

C2: **C Melancon**, K Kaur, J Gascon-Samson, M Saad, "Towards Smart Distributed Robotics Solution using Digital Twin" *Proceedings of the International Conference on Power Electronics and Drive Systems*, 2023.

C3: G Simard, **C Melançon**, P Cardinal, J Gascon-Samson, "Performance Characterization of MQTT Brokers in a Device-Local Edge Deployment" *MiddleWEdge 2023 - Proceedings of the 2nd International Workshop on Middleware for the Edge, Part of: ACM/IFIP Middleware 2023*.



## **CHAPTER 2**

### **LITERATURE REVIEW**

Robotics is becoming increasingly prevalent across industrial, service, and healthcare sectors, where autonomous systems perform tasks such as logistics, inspection, surgery, and patient support. These robots rely on multiple perception, localization, navigation, and decision-making modules, which require considerable computing power and generate a large volume of sensor data. Battery-powered platforms must therefore balance strict real-time constraints with limited onboard resources and battery life, making it difficult to execute all workloads locally and justifying the use of distributed robotics architectures across edge, fog, and cloud infrastructures.

The aim of this literature review is to examine the state of the art in autonomous robotics and distributed computing to identify requirements and gaps that motivate the contributions of this thesis. The chapter focuses on four main aspects:

1. Autonomous robots and their application domains, with an emphasis on healthcare-inspired scenarios;
2. AI models for perception and localization under real-time constraints;
3. Distributed robotics concepts, including edge, fog, cloud, and the Internet of Robotic Things;
4. Multi-layer orchestration techniques for highly distributed systems.

The synthesis at the end of the chapter highlights open challenges related to real-time AI execution, multi-layer communication, and dynamic orchestration for autonomous robots.

#### **2.1 Strategy**

The literature review covers multiple aspects of autonomous robots and distributed systems, including application use cases, AI models, distributed architectures, communication middlewares, and orchestration mechanisms. The primary databases used for the paper search are Compendex, Inspec, Knovel, and IEEE Xplore. The main keywords used in various combinations include:

- autonomous Robot
- distributed Robotics

- edge Robotics
- fog Robotics
- cloud Robotics
- Internet of Robotic Things
- robot Healthcare
- multi-layer data distribution
- distributed system platform
- artificial intelligence optimization and operationalization
- real-time inference
- dynamic orchestration

The search primarily targets work from the last decade, complemented by seminal earlier contributions when necessary. Relevant survey articles and framework papers serve as starting points for examining their reference lists and subsequent citations to identify additional contributions in robotics, edge/fog/cloud computing, and IoT. The subsequent sections synthesize the selected works along the four dimensions outlined above, starting with autonomous robot applications and their requirements.

## **2.2 Autonomous Robots**

Autonomous robots are increasingly visible in everyday environments, ranging from domestic and assistive robots to autonomous vehicles and industrial manipulators as described by Chinmay & Nisha (2024) and Zachiotis *et al.* (2018). These systems combine sensing, perception, localization, planning, and control to operate with limited human supervision, often in close proximity to people or in safety-critical contexts. This section surveys representative application domains to derive common functional requirements and constraints that will later guide the discussion on AI models and distributed architectures.

### **2.2.1 Use Cases**

Autonomous robots span several application domains that differ in their tasks, environments, and interaction patterns with humans. Understanding these use cases is important because it reveals recurring capabilities and constraints that any realistic platform must support. The following subsections summarize four major categories that are particularly relevant to this thesis.

According to Chinmay & Nisha (2024), autonomous robots can be categorized into assistive robots, autonomous vehicles, and autonomous manipulators. Assistive robots support humans in daily living or social interaction, for example by simplifying personal chores, providing companionship, or assisting with mobility. These applications involve rich human-robot interaction and require safe navigation in shared spaces, robust perception of human behavior, and ethically acceptable motion patterns (Kruse *et al.* (2013)).

#### **2.2.1.1 Home Service and Social Robots**

Home service robotics is a related field that has gained momentum with the emergence of domestic robots according to Zachiotis *et al.* (2018). The authors identify five application areas: educational, entertainment, household, social, and gaming. Educational and entertainment robots act as autonomous or teleoperated agents for language learning, storytelling, dance, or music. Household and social robots perform tasks such as floor cleaning or dishwashing, or provide emotional support by detecting user feedback and adapting their behavior. These scenarios typically involve relatively unstructured environments, a variety of sensors, and continuous operation subject to energy and cost constraints.

#### **2.2.1.2 Autonomous Vehicles**

Autonomous vehicles are described by Chinmay & Nisha (2024) as another prominent class of autonomous robots. Here, the human is embedded in the robot, and the design is guided by objectives such as emissions reduction, energy efficiency, affordability, safety, and traffic optimization. Autonomous driving workloads combine high-bandwidth sensing (*e.g.*, cameras,

LiDAR, radar) with complex perception and planning models that must meet strict real-time deadlines. Concepts and techniques from autonomous driving carry over directly to mobile service robots and are therefore highly relevant for this thesis.

#### **2.2.1.3 Industrial and Logistics Robots**

Industrial manipulators are primarily used for hazardous operations or repetitive tasks that pose physical or ergonomic risks to humans (Chinmay & Nisha (2024)). Examples include handling chemicals, operating in extreme environments, or executing high-precision assembly tasks. While many industrial robots remain stationary and rely on fixed infrastructure, collaborative and mobile manipulators introduce additional challenges related to human-robot interaction, workspace sharing, and safety monitoring.

#### **2.2.1.4 Assistive and Healthcare Robots**

Healthcare is another sector that increasingly leverages robotics. An early survey by Hockstein *et al.* (2007) documented the exponential growth of robotics in healthcare, focusing on automated and teleoperated surgeries. More recent work by Khan & Anwar (2020) reviews therapy robots used as companions to reduce anxiety or loneliness, assist patients with mental health issues, and support rehabilitation. Surgical robots have evolved towards surgical assistance platforms that augment the surgeon's dexterity, as well as miniaturized robots that can be injected or inserted into the body to access otherwise invasive regions. Assisted telesurgery is also described, aiming to deliver surgical capabilities to remote or dangerous locations. Exoskeletons can provide support for rehabilitation and mobility assistance as proposed in Bedolla-Martinez *et al.* (2023). Finally, Shilpa *et al.* (2023) discuss integrating robotics with IoT technologies to build end-to-end healthcare solutions that span diagnosis, treatment, and patient monitoring.

Beyond ground robots, Unmanned Aerial Vehicles (UAVs) represent another important class of autonomous robots capable of operating in remote or hard-to-reach areas as discussed in Jiang *et al.* (2023). UAVs can be used for inspection, delivery, environmental monitoring, and

emergency response. They typically face tight payload and energy constraints, strong safety and regulatory requirements, and dynamic environmental conditions.

## **2.2.2 Concepts and Requirements**

Beyond their concrete applications, autonomous robots share a set of fundamental concepts and requirements that shape how their software and hardware are designed. On the functional side, core capabilities such as localization, mapping, navigation, and interaction must operate reliably under diverse conditions. On the non-functional side, robots must satisfy strict constraints related to safety, real-time performance, energy, and robustness in order to be viable outside controlled environments. This subsection reviews these aspects and highlights their implications for system architecture.

### **2.2.2.1 Core Functional Capabilities**

Localization and navigation are central concepts in almost all of these applications. Nimura *et al.* (2023) present a LiDAR-based Simultaneous Localization and Mapping (SLAM) approach for indoor navigation, which demonstrates how mapping and localization can be computed in real time with distributed processing. Mendes *et al.* (2016) propose an Iterative Closest Points (ICP)-based SLAM implementation that also relies on LiDAR data to build the map. Moore & Stouch (2016) describe how Kalman filtering can be used to fuse multiple sensors such as IMU, encoders, and Global Positioning System (GPS) to estimate the robot pose in outdoor environments, and Geiger *et al.* (2012) provide a dataset that has become a reference for such tasks. For indoor scenarios where GPS is unavailable, Bach *et al.* (2024) investigate the use of UWB as an alternative for localization. Zhang *et al.* (2022) and Clark *et al.* (2017) explore visual odometry approaches that combine camera images with DL models, showing that these methods can outperform classical filtering when properly trained.

Navigation and obstacle avoidance also require careful consideration. Gunasekaran *et al.* (2018) propose a pipeline that builds an occupancy grid from LiDAR data and uses a genetic algorithm

to compute paths, which are then followed by a trajectory controller. Zhang (2021a) investigate reinforcement learning techniques for navigation without a prior map and compare them with SLAM-based methods. Massoud *et al.* (2022) distinguish between static and dynamic obstacles and between known and unknown static obstacles, proposing a LiDAR-based genetic algorithm for path planning in indoor environments. Melancon *et al.* (2009) describe a simple dynamic obstacle avoidance strategy that changes from navigation to avoidance mode, controlling its position with respect to the obstacle until the obstacle is no longer in the path of the original trajectory. Authors in Sanchez-Lopez *et al.* (2019) present an artificial potential field approach where the obstacles are acting as repulsive magnets and the destination as an attractive magnet. One challenge of this method occurs when two obstacles are near and create a repulsive field, making it impossible for the algorithm to navigate between them. The solution proposed by Lin *et al.* (2025) addresses this limitation by introducing a rotating repulsive field to enable navigation through narrow passages.

Occupancy grids were originally introduced by Elfes (1989) as a way to represent the environment as a grid of occupied and free cells. Zheng *et al.* (2022) show how occupancy grid maps from multiple vehicles can be fused to improve navigation in shared environments. Thomas *et al.* (2022) propose predictive occupancy grids using reinforcement learning to handle dynamic obstacles, and Messaoud (2021) use DL models to predict the trajectories of dynamic objects. These works illustrate how perception and prediction components are becoming increasingly intensive on data and compute.

When robots operate in human-populated environments, navigation must be human-aware. Li *et al.* (2019) present a Deep Reinforcement Learning (DRL) approach for human-aware navigation in crowded spaces. Kruse *et al.* (2013) survey human-aware navigation and highlight aspects such as comfort, naturalness of motion, and social acceptability. Rudenko *et al.* (2020) provide an extensive survey on human motion trajectory prediction and distinguish between model-based and learning-based approaches. These studies underscore the need for robots to reason about human behavior over short and long time horizons.

### 2.2.2.2 Non-functional Requirements

Across all these works, several non-functional requirements recur regularly. The robots must handle high-rate sensor streams, such as IMU data at tens or hundreds of hertz and camera data at tens of frames per second (Lu *et al.* (2026)). They must execute control and perception loops within deadlines ranging from tens to a few hundred milliseconds, depending on the task (Kramer *et al.* (2025)). At the same time, they typically rely on battery power, which limits their energy and thermal budgets (Loukatos *et al.* (2024)). Taken together, these constraints make it challenging to host all perception, localization, and decision-making workloads solely on the robot.

### 2.2.3 Synthesis and Implications

The literature on assistive, domestic, industrial, transportation, and healthcare robots shows that these systems share common architectural building blocks and strict constraints. They process large volumes of sensor data, rely on multiple AI and optimization components, and must satisfy real-time and safety requirements while operating on platforms constrained in resource energy. The examples discussed above also highlight the growing use of learning models for perception, localization, prediction, and human-aware behaviors.

These characteristics indicate that purely onboard computation quickly becomes a bottleneck, particularly when multiple AI models are deployed simultaneously. They also suggest that integrating external compute resources, such as edge, fog, and cloud infrastructures, can help meet computational demand but may introduce new challenges related to latency, reliability, and orchestration. The next sections build on these observations by reviewing AI models for autonomous robotics, distributed robotics architectures, and orchestration mechanisms that aim to address these challenges.

## 2.3 Artificial Intelligence in Robotics

As seen in the previous section, autonomous robots are taking an increasing role in industry and public spaces and require multiple algorithms to accomplish their tasks (Pareigis *et al.* (2023)). When robots share space with humans, these algorithms become safety-critical and demand particular attention to ensure that the robot neither harms people nor damages the environment. This section reviews key AI components used in robotics, focusing on localization and sensor fusion, perception and navigation, deep reinforcement learning, and techniques for real-time operationalization, before synthesizing their implications for this thesis.

### 2.3.1 Localization and Sensor Fusion

The basis of autonomous robotics is the ability to localize the robot in its environment. Sensor fusion is a common use of AI in robotics, where vision data is added to conventional sensor models. Han *et al.* (2022) propose a localization model based on camera images and LiDAR data fused in a Convolutional Neural Network (CNN) to estimate the robot's linear and angular velocity. Barreto-Cubero *et al.* (2022) extend this idea by adding sonar data to a fully connected neural network, limiting camera data to the LiDAR field of view to reduce the number of parameters.

Another approach, explored by Chitta *et al.* (2023), uses a multi-modal fusion transformer to obtain localization from camera images and LiDAR data. By leveraging attention mechanisms, the model can focus on prominent landmarks to improve performance. Visual odometry is also widely studied. Clark *et al.* (2017) propose VINet, which builds on FlowNet (Dosovitskiy *et al.* (2015)) for image feature extraction and uses a recurrent neural network with IMU data to exploit temporal memory and improve precision. Wang *et al.* (2017) present DeepVO, a similar visual odometry approach without the use of IMU measurements. In Krishnan & Sahin (2019), the authors describe Oriented Fast and Rotated Brief (ORB)DeepOdometry, which combines a CNN and Long Short Term Memory (LSTM) with ORB features as the feature extraction method.

Indoor robotics scenarios often cannot rely on GPS. Authors in Poulouse & Han (2021) propose a sensor fusion model based on UWB and an LSTM network that reduces localization error from 10 cm to 5 cm.

These works illustrate how AI-based sensor fusion methods can significantly improve localization quality compared with single-sensor approaches, but they also increase computational demands and introduce tight real-time constraints on inference.

To support training and evaluation of such models, several datasets provide access to sequences of LiDAR, camera images, and other sensor data. The KITTI dataset, introduced in Geiger *et al.* (2012), is widely used for outdoor scenarios and provides synchronized images, LiDAR, and GPS/IMU measurements. For indoor robotics, the MIT Stata Center dataset from Fallon *et al.* (2013) offers complete sensor data in ROS bag format, which facilitates realistic replay and experimentation for visual odometry and localization.

### **2.3.2 Perception, Prediction, and Navigation**

Perception, prediction, and navigation collectively determine how a robot interprets its surroundings and decides where to move. Perception modules detect and track relevant objects and structures, prediction modules forecast the future motion of agents, and navigation components plan and execute safe trajectories. The following subsections first review AI techniques for navigation and path planning, then focus on motion prediction and intent modeling in dynamic, human-populated environments.

#### **2.3.2.1 Navigation and Path Planning**

For navigation, robots rely on AI models for multiple tasks. Traffic-sign and landmark recognition is a common requirement in autonomous driving and service robotics. Chen *et al.* (2022a) investigate real-time detection of traffic signs using CNN, demonstrating that high accuracy can be achieved at real-time speeds when models are properly optimized. Object detection for manipulation is another key capability explored in Chen *et al.* (2023b). The authors present

a design for an autonomous harvesting robot that detects fruits to be picked in agricultural scenarios. Peszor *et al.* (2018) use optical flow to detect obstacles from camera images, enabling robots or vehicles to avoid collisions based on the observed motion patterns.

Beyond perception and prediction, AI models are also used for motion planning. In Xiao *et al.* (2022), the authors survey the use of machine learning for motion planning in robotics and decompose the navigation pipeline into global planning (*e.g.*, SLAM and occupancy-grid-based planning) and local planning for control. They outline how DRL and LSTM-based architectures can be used to capture temporal dependencies and improve local decision-making. Janji & Kliks (2022) investigate neural network variants of classical planning methods, including adaptations of rapidly exploring random trees with CNN-based feature extraction and generative adversarial networks to improve robustness under noisy map data. Other neural network-based techniques are explored for scenarios with moving obstacles, where planners must account for dynamic constraints and interactions.

Path tracking and control can also benefit from DRL. Joglekar *et al.* (2022) adapt the pure-pursuit path tracking algorithm (Coulter (1992)) using DRL to learn both the look-ahead distance and linear velocity based on previous commands and IMU data. This approach improves robustness to environmental variations compared with fixed-parameter controllers. The work in Srikonda *et al.* (2022) uses DRL for trajectory tracking based on top-view images. While this method shows promising performance, its applicability is limited by the requirement for infrastructure that provides a top-down view of the robot's environment.

SLAM itself is increasingly enhanced by neural networks. Liu *et al.* (2019) integrate CNN and RNN components into SLAM to exploit RGB-D images and improve point-cloud registration and pose learning. The authors in Liu *et al.* (2022b) also propose a bio-inspired navigation framework that builds an online experience map from grid-cell and place-cell-based pose encoding as well as visual cues, then learns a predictive map to guide trajectory planning. This approach supports robust navigation in static and dynamic environments with compact maps that can be reused without retraining after local changes. In Thomas *et al.* (2021), the authors address the challenge

of separating static and dynamic elements by introducing a self-supervised LiDAR segmentation pipeline based on SLAM and ray tracing. Their method labels ground, permanent structures, and moving objects without human annotation, enabling downstream localization and planning components to focus on relevant static geometry.

### **2.3.2.2 Motion Prediction and Intent Modeling**

When navigating crowded environments, robots must predict human motion to avoid accidents and behave in a socially acceptable manner. In Wang *et al.* (2021), the authors use transformers to predict human pose evolution, showing that attention-based architectures can effectively capture temporal dependencies in body movement. Amirloo *et al.* (2022) propose LatentFormer, a multi-agent transformer model for vehicle trajectory prediction that splits attention between the static map and the dynamic agents, then fuses both to forecast future motion. Recurrent Neural Network (RNN) is leveraged in Messaoud *et al.* (2019) to predict vehicle trajectories from a top-view drone perspective, without direct vehicle sensor data, by modeling agent interactions from the scene. These models highlight the trend toward increasingly complex predictive components integrated into robot navigation pipelines.

### **2.3.3 Deep Reinforcement Learning and Learning Dynamics**

Deep reinforcement learning is increasingly used in robotics to learn complex behaviors and decision policies. Matsumoto *et al.* (2020) propose a simulation-based DRL approach to learn multiple robotic tasks, emphasizing the benefits of simulation for accelerating training that requires many repetitions of similar scenarios. In Park *et al.* (2024), the authors apply DRL with Hindsight Experience Replay to navigation tasks, showing that replay mechanisms can improve both training efficiency and inference performance. Nagabandi (2020) use DRL to model the robot's surroundings and adapt to environmental variations, and further propose a meta-learning algorithm to quickly learn new tasks from limited data. Transfer learning strategies, for reusing policies across different robotic tasks and environments, reducing the need for training from scratch is studied in Ozturk *et al.* (2023).

A comprehensive survey of reinforcement learning in robotics is proposed in Singh *et al.* (2022), covering actor-critic methods, human-centered approaches, DRL, and multi-agent scenarios, as well as different neural topologies such as neuro-evolution, spiking neural networks, and cognitive architectures. Their survey highlights both the diversity of RL approaches and the growing complexity of robotic tasks tackled by these methods.

A crucial enabler for DRL in robotics is high-fidelity simulation. Kainova (2023) present an overview of NVIDIA ISAAC Sim, a simulator that provides multiple toolboxes and integration capabilities for robotics and enables large-scale, realistic training of perception and control policies. Such simulators reduce the need for repeated physical experiments, which can be costly, time-consuming, or risky, and allow for the exploration of a wide range of scenarios before transferring policies to real robots.

### **2.3.4 Real-time Operationalization**

Many AI models used in robotics and related domains must run under strict real-time constraints on resource-limited platforms. In practice, this means that designers often need to adapt the execution pipeline, the model, or the deployment hardware to meet timing, energy, and memory requirements. Several authors also emphasize that this problem should be viewed in a broader AI lifecycle perspective, where training, deployment, monitoring, and updating are tightly coupled. Carastan-Santos & Pham (2022) discuss how understanding the full AI lifecycle is essential to move from proof-of-concept models to reliable intelligent systems in the field. In Rajendran & Shyam Shankaran (2021), the authors present big-data and learning pipelines that highlight the need for scalable data management and processing when deploying learning-based services at scale.

#### **2.3.4.1 Processing Pipeline Optimization**

Some work focuses on optimizing the processing pipeline around existing algorithms. Josyula *et al.* (2019) present a fast point cloud segmentation pipeline implemented in ROS and the

Point Cloud Library, where each LiDAR scan is divided into radial slices that are processed in parallel stages. Their approach reduces run time, while preserving segmentation accuracy, which makes it suitable for safety-critical obstacle avoidance. Barodi *et al.* (2022) propose an improved implementation of a safety-critical perception pipeline for automated driving that reduces end-to-end latency while maintaining detection performance, further illustrating how refactoring and parallelization of existing components can help satisfy timing constraints. A similar idea appears in crowd enumeration systems introduced in Kounte *et al.* (2023), where MobileNet-SSD is selected as a lightweight detector and scheduled to run at high frame rates on an embedded GPU such as a Jetson Nano to raise alarms when a region becomes overcrowded. In all these cases, the main contribution is to restructure processing steps and to choose light models so that end-to-end latency becomes compatible with real-time use.

This subsection reviews work on real-time operationalization of AI in robotics, focusing on pipeline optimization, model optimization, distributed deployment, and remaining open challenges.

#### **2.3.4.2 Model Optimization**

Other works compare different AI models and accelerators to identify options that are better suited for low-power devices. Munim & Rahman Kohinoor (2023) evaluate several pre-trained face recognition models (FaceNet, OpenFace, DeepFace, VGGFace) on a constrained mobile environment. The authors show that FaceNet achieves the best compromise between accuracy and inference time, and the lowest memory usage, making it a strong candidate for real-time deployment on low-end devices. In Kang & Park (2026), the authors analyze the energy consumption and performance of multiple accelerators, including a Coral Edge TPU, an Intel Movidius NCS2, and embedded GPUs on Jetson boards. They report that quantized accelerators can offer much better energy efficiency than CPUs or GPUs for inference, but also note that idle power and model size have a large impact on overall energy consumption. These results highlight that model choice and hardware selection are both crucial for real-time and energy-aware AI. In Belkahla *et al.* (2025), the authors perform detailed energy and timing measurements for

deep neural networks running on heterogeneous embedded platforms and report how different hardware and framework configurations affect inference latency and power usage. Bültemann *et al.* (2023) present an energy study of AI workloads on edge devices and highlight trade-offs between model complexity, batch size, and device utilization. These works show that model choice and hardware selection are both crucial for real-time and energy-aware AI.

#### **2.3.4.3 Distributed Deployment**

Real-time constraints also affect how AI workloads are split between edge, fog, and cloud. Suneel *et al.* (2025) propose an edge intelligence hybrid framework with three layers, where lightweight models and preprocessing run at the edge, aggregation and synchronization run at the fog, and complex training and analytics execute in the cloud. They show how this hierarchy can balance latency and accuracy for time-critical IoT tasks. Barbosa *et al.* (2023) introduce LightSpaN, a sparse CNN variant for vehicle detection in intelligent traffic systems. Their model is designed to run fast on IoT devices and is integrated with a traffic light control algorithm that reduces waiting and travel time for priority vehicles, showing that model simplification and system-level integration are both important for timely decisions. The energy-efficiency of orchestration strategies used for AI services across distributed infrastructures is studied in Dinar *et al.* (2023), showing that placement and scaling decisions have a strong impact on latency and energy consumption. Their findings connect real-time AI operationalization with the broader orchestration topics discussed later in this chapter.

#### **2.3.4.4 Open Challenges for Real-time AI in Robotics**

Human-robot interaction provides additional examples where real-time AI is essential. Vu *et al.* (2024) present a collision detection system for collaborative robots that uses a single RGB camera and a MediaPipe-based human detection model to detect body parts entering the robot workspace. Their system runs on a laptop and stops a robot arm when a person approaches, improving safety at the cost of increased assembly time. In another context, Shah *et al.* (2023) use real-time human pose estimation from RGB video to support gesture-based interaction

and tracking. Chen *et al.* (2022a) propose a real-time traffic sign detection network tailored to embedded platforms and report inference times compatible with automotive control-loop requirements. These works underline that real-time AI operationalization must consider not only latency and throughput, but also how the AI component influences the overall task duration and user experience.

From a model-design perspective, there is also interest in making modern architectures such as transformers more efficient for deployment. Zhuang *et al.* (2023) survey acceleration methods for transformers, including pruning, quantization, lightweight attention mechanisms, and efficient training and inference schemes. Their analysis shows that significant speed-ups are possible with limited loss in accuracy, which is highly relevant for perception and prediction models in robotics that increasingly rely on transformer-based backbones.

These observations motivate a more systematic study of AI operationalization for autonomous robots. In this thesis, the focus is on DL models for localization and perception, on quantifying execution time, resource usage, temperature, and energy across heterogeneous hardware, and on using these measurements to guide deployment choices in distributed edge-fog-cloud robotics platforms.

### **2.3.5 Synthesis and Implications**

The reviewed literature shows that AI is now deeply embedded in most stages of the robotic pipeline, from localization and mapping to perception, human motion prediction, planning, and control. CNN, RNN, transformers, and DRL agents are all used to improve accuracy, robustness, and autonomy. However, these models come with substantial computational and energy requirements and must often operate under strict real-time constraints, especially for safety-critical functions such as localization, obstacle avoidance, and human-aware navigation.

At the same time, training typically occurs on powerful servers while inference is executed on resource-constrained platforms, which raises questions about deployment strategies, model optimization, and hardware selection. Some works provide detailed measurements of execution

time and energy consumption on specific devices, but only a few systematically analyze these aspects across different hardware configurations and in relation to concrete control-loop requirements. These gaps motivate the focus of this thesis on operationalizing AI models for autonomous robots under real-time and energy constraints and on exploiting distributed robotics architectures to offload computation when necessary.

## **2.4 Distributed Robotics**

Distributed robotics investigates how autonomous robots cooperate with each other and with external compute resources to perform sensing, perception, and control tasks under resource and real-time constraints. In contrast with standalone robots, these systems explicitly rely on communication networks and distributed computing layers such as edge, fog, and cloud to meet increasing demands for computation, storage, and data sharing. This section focuses on architectural models, data distribution, and configuration mechanisms that are relevant for autonomous robots operating over multi-layer infrastructures derived from the provided literature.

### **2.4.1 Models and Challenges**

Several architectural models have been proposed to structure how robots interact with external computing resources and with each other. Cloud robotics, edge and fog robotics, and the Internet of Robotic Things (IoRT) represent complementary perspectives on where computation, storage, and intelligence should reside. This subsection summarizes these models and highlights key challenges that arise when deploying robotic workloads in distributed environments.

#### **2.4.1.1 Cloud Robotics**

Early work on cloud robotics emphasized extending networked robots with cloud computing to overcome onboard resource limitations and to share information and learned skills across a fleet. In the architecture proposed by Hu *et al.* (2012), robots communicate via machine-to-machine

links to form an interconnected team of robots and reach infrastructure clouds through machine-to-cloud connections, enabling task offloading and access to shared map or skill repositories while introducing new dependencies on network security and Quality of Service (QoS), such as latency and throughput. Quintas *et al.* (2011) further adopt a service-oriented approach where mobile robots and smart-room infrastructures expose capabilities as services and use cloud-based repositories to store and retrieve robot skills, which supports assisted living scenarios but raises questions about how to partition functionality between onboard and remote components.

#### **2.4.1.2 Edge and Fog Robotics**

Fog and edge robotics refine these ideas by introducing intermediate layers that bring computation and storage closer to the robots. In Ivanov (2020), the authors consider fog robotics for groups of mobile robots performing environmental monitoring and shows how distributing computation within the robot group reduces its load and balances energy consumption while still relying on remote servers for integration and aggregating the data coming from the robots into the map. Krishna Chand Gudi *et al.* (2018) focus on human-robot interaction and propose a fog robotics architecture where fog servers handle low-latency perception and dialogue tasks near social robots, while only residual or aggregated data is sent to the cloud. This approach improves responsiveness and robustness to connectivity issues. These works highlight the responsibilities of the different layers: fog and edge nodes to pre-process and perform low-compute tasks, whereas cloud resources are reserved for heavier analytics or cross-site knowledge sharing.

#### **2.4.1.3 Internet of Robotics Things**

Concrete frameworks illustrate how these models can be realized in field robotics and IoT settings. Muratore & Tsagarakis (2023) introduce XBot2D, a hybrid architecture that dynamically allocates robotic skills between local computers and cloud instances based on network conditions, demonstrating that perception and planning components can migrate between edge and cloud while maintaining acceptable control performance. Bédard *et al.* (2023) propose tracing-based analysis of distributed ROS2 applications, providing developers with visibility into message flows,

end-to-end latencies, and causal relationships across nodes and machines, which is essential to reason about the impact of distribution choices on timing and reliability. In Samara *et al.* (2022), the authors survey the Internet of Robotic Things and outline a three-layer architecture where robots act as smart IoT nodes connected through gateways and cloud services, illustrating how robotics-specific requirements fit into broader IoT and cyber-physical system trends.

#### **2.4.1.4 Key Challenges in Distributed Robotics**

The evolution of computing paradigms also shapes distributed robotics models. A retrospective on distributed systems traces the progression from centralized mainframes to cloud, fog, and edge computing is discussed in Lindsay *et al.* (2021). The authors highlight how systems repeatedly alternate between centralization and decentralization and identify open challenges such as energy efficiency and managing complexity at scale. Tank & Gandhi (2023) propose a comparative study of cloud, edge, and fog computing to classify these paradigms along dimensions such as latency, scalability, security, and management cost. According to Dordevic *et al.* (2023), these directly influence where robotic workloads such as localization, perception, and planning should execute. At the same time, empirical studies on computation offloading for mobile robots show that moving parts of the pipeline to nearby servers over WiFi can reduce onboard energy usage but makes performance sensitive to bandwidth, interference, and workload characteristics, underscoring the need for joint design of offloading policies and control architectures.

Collectively, these works indicate that distributed robotics must balance three intertwined concerns. First, robots increasingly depend on external resources to support data-intensive AI models, high-rate sensing, and complex coordination tasks (Hu *et al.* (2012)). Second, real-time and safety constraints limit acceptable end-to-end latency and jitter for control and perception loops, which constrains where functions can be placed and how communication should be structured (Krishna Chand Gudi *et al.* (2018)). Third, heterogeneity in robots, networks, and compute platforms introduces configuration and management challenges, motivating architectures that support dynamic placement, monitoring, and adaptation across cloud, fog, and edge layers (Muratore & Tsagarakis (2023)).

## 2.4.2 Data Distribution and Dynamic Configuration

Effective distributed robotics requires not only architectural models but also concrete mechanisms to move data and reconfigure services across nodes. Communication paradigms, middleware choices, and data distribution patterns all influence latency, reliability, and resource usage. This subsection reviews work on communication infrastructures, multi-layer data flows, and dynamic configuration techniques in edge–fog–cloud systems.

### 2.4.2.1 Communication Paradigms and Middlewares

Data distribution mechanisms determine how sensor streams, state information, and control commands flow between robots and across infrastructure layers. Publish-subscribe middleware, as discussed in Kang *et al.* (2020), is widely used to decouple producers and consumers in IoT and robotic systems, allowing components to exchange messages without hard-wired dependencies. The authors present an experimental evaluations of Message Queuing Telemetry Transport (MQTT), Data Distribution Service (DDS), and other brokers under diverse IoT traffic patterns show that topic structure, QoS levels, and broker deployment strategies significantly affect throughput, latency, and resource usage, which in turn influence how well distributed robotic applications can meet their real-time requirements. A similar experiment is proposed in Simard *et al.* (2023) for the use of various MQTT brokers in a local dissemination scenario. In smart-city scenarios, as discussed in Ksentini *et al.* (2020), fog-computing simulations with iFogSim demonstrate that placing processing elements closer to data sources reduce network usage and improves latency for delay-sensitive services such as healthcare monitoring and intelligent transport, which supports similar designs for robots operating in urban environments.

### 2.4.2.2 Multi-layer Data Distribution

Internet of Robotic Things architectures, introduced in Ray (2016), explicitly structure communication and data flows across perception devices, gateways, fog nodes, and clouds. The authors in Samara *et al.* (2022) describes how sensor data from robots and other devices is

preprocessed and filtered at edge or fog nodes, with only selected information forwarded to cloud services for long-term storage, analytics, or cross-domain coordination. Such hierarchies allow many control and interaction decisions to be taken near the robots, while global services build shared situational awareness and support remote supervision or fleet management. Serverless data pipelines in Poojara *et al.* (2021) extend this idea by composing event-driven functions across fog and cloud to handle streaming applications, showing that different pipeline configurations can be better suited for bandwidth-intensive or compute-intensive workloads, which suggests that robotic dataflows may also benefit from function-level placement and reconfiguration.

#### **2.4.2.3 Dynamic Configuration and Adaptation**

Several works address dynamic configuration and resource orchestration across edge-fog-cloud environments. Authors in Happ *et al.* (2021) formulates the joint placement of analytics operators and publish-subscribe brokers in fog platforms as an optimization problem and proposes heuristics that reduce latency and communication cost compared with cloud-only deployments, especially when producers and consumers are geographically clustered. PuzzleMesh introduced in Sanchez-Gallegos *et al.* (2022), defines an infrastructure-agnostic service mesh where microservices are modeled as puzzle pieces that can be composed and reconfigured to process large data streams across edge, fog, and cloud. This approach enables flexible deployment of data-processing pipelines under changing resource conditions.

Within robotics, dynamic configuration also appears in studies that analyze how message flows and execution times evolve as ROS2 applications are distributed over multiple machines. Hybrid architectures proposed in Muratore & Tsagarakis (2023) demonstrate that robot skills can be migrated between local and cloud nodes based on observed network performance and resource availability, suggesting that orchestration decisions should incorporate both infrastructure metrics and application-level constraints such as control-loop deadlines. Finally, fog-robotics frameworks for monitoring tasks propose multi-agent methods to redistribute computation among robots when failures occur, while minimizing communication load and balancing energy

usage across the group. This highlights the importance of coordination mechanisms that react to both environmental changes and platform-level events.

### **2.4.3 Multi-layer Orchestration**

As robotic applications and infrastructures grow in scale and complexity, manually managing where services run and how resources are allocated becomes impractical. Multi-layer orchestration frameworks aim to automate service placement and reconfiguration under heterogeneous constraints and objectives. This subsection reviews orchestration concepts, algorithmic approaches, and their application to robotics, with a particular emphasis on digital twin-based evaluation.

#### **2.4.3.1 Orchestration Concepts**

The literature converges on viewing edge, fog, and cloud as a computing continuum that must be orchestrated as a whole rather than as isolated tiers. Tank & Gandhi (2023) highlight how cloud, edge, and fog provide complementary capabilities for IoT workloads, motivating orchestration mechanisms that decide where along this continuum to execute functions based on latency, bandwidth, and scalability constraints. Building on this, the work in Tianfield (2019) explicitly frames edge-cloud computing as a three-layer hierarchy (smart edge, fog, cloud) in which the fog layer assumes orchestration and intermediary roles, coordinating data pipeline functions and service provisioning across mission-oriented, real-time IoT systems. In Nakamura *et al.* (2016), the authors introduce IFoT, which distributes stream-processing tasks across proximity devices according to declarative recipes, thereby orchestrating flows and operators near data sources while still integrating with cloud resources.

#### **2.4.3.2 Orchestration Approaches**

A core thread of work focuses on systematic service placement and load balancing over the edge-fog-cloud continuum. Nezami *et al.* (2020) formulate IoT service placement as a decentralized

multi-objective optimization problem over an edge-to-cloud graph and propose a multi-agent collective learning scheme in which nodes locally construct candidate plans and cooperatively select placements that jointly balance workload and QoS costs. Kapsalis *et al.* (2017) design a cooperative fog approach for workload balancing, using context such as node utilization, latency, and battery level to decide which fog or edge host should serve each request. Badidi (2019) presents a fog node architecture for real-time urban IoT streams that orchestrates pre-processing, filtering, and aggregation tasks between city gateways and cloud analytics to reduce uplink load while preserving responsiveness. Wang *et al.* (2018a) add a multi-agent mechanism to fog architectures for cyber-physical power systems, in which agents coordinate the distribution of control and monitoring tasks across fog and cloud resources. Together, these works treat orchestration as a placement and scheduling problem that must jointly consider heterogeneity (devices, links, applications) and multi-dimensional QoS objectives.

Another line of work explores autonomous and policy-driven orchestration for edge-native applications. Petri *et al.* (2021) present the concept of “autonomics” at the edge, where resource orchestration for edge-native services is guided by cost models and resilience objectives, and where management loops adjust placements and configurations in response to changing load and failures. An adaptive scheduling architecture is proposed in Alhameed (2024) that selects among cloud, edge, and fog resources for IoT workloads, exposing a scheduling layer that can adapt to application constraints and infrastructure variability. The work in Manivannan *et al.* (2023) address energy-efficient scheduling in Industrial IoT, using metaheuristics to allocate tasks to fog nodes and cloud so as to respect deadlines while optimizing energy and utilization. The authors in Akbari *et al.* (2022) propose an overlapping routing approach for large-scale IoT, where routing and placement decisions jointly exploit fog and cloud to improve path diversity and reliability. These contributions emphasize orchestration as an ongoing control loop, rather than a one-shot mapping, and highlight the importance of energy, resilience, and routing-awareness in multi-layer decision making.

### 2.4.3.3 Orchestration in Robotics

Several works bring multi-layer orchestration closer to robotics and control. In Hu *et al.* (2012), the authors discuss cloud robotics architectures and how computation and data can be partitioned between robot, edge, and cloud to support perception, planning, and learning at scale, while flagging latency and reliability as central orchestration challenges. Quintas *et al.* (2011) similarly envision cloud robotics as context-aware robotic networks that leverage service-oriented architectures to dynamically bind robots to cloud-hosted and edge-hosted services. Muratore & Tsagarakis (2023) propose a hybrid cloud architecture in which robotic skills can be transparently relocated between local real-time execution and cloud instances based on network quality, latency requirements, and energy constraints, effectively orchestrating a two-dimensional local-cloud space for field robotics. Baxi *et al.* (2025) tackle time-critical control in edge robotics over 5G, emphasizing joint co-design of communication and edge compute to achieve end-to-end deadlines, and thus implicitly specifying orchestration requirements on where control loops and offloaded computations should run. Din Din *et al.* (2024) discuss AIoT integration for autonomous vehicles, where distributed road-vehicle cooperation requires orchestrating perception, prediction, and coordination services across vehicles, roadside edge, and cloud to meet stringent real-time and safety constraints.

Also, multiple works address orchestration-relevant mechanisms that, while not full orchestration frameworks, influence how multi-layer orchestration should be designed. Moore & Stouch (2016) provide a generalized Extended Kalman Filter (EKF) implementation for ROS, illustrating how state-estimation components can be modularized and potentially relocated across nodes in a distributed system. Park & Kim (2016) study logical time synchronization for safety-critical distributed systems, which is essential if control and sensing tasks are split across layers yet must obey temporal safety constraints. Krishnappa *et al.* (2024) survey IoT cloud platforms and integration challenges, underscoring interoperability, real-time processing, and edge integration as key requirements for orchestrated IoT-cloud solutions. Khoshvaght *et al.* (2025) propose a multi-objective deep reinforcement learning algorithm for spatio-temporal latency optimization in mobile IoT edge computing networks, showing how learned policies

can steer task placement and routing decisions across edge infrastructure over time. Cassales *et al.* (2023) explore performance-energy trade-offs for data-stream classification at the edge, suggesting that orchestration must not only meet latency targets but also balance energy usage when deciding where to run continuous analytics pipelines.

Across these works, multi-layer orchestration emerges as a cross-cutting concern that spans architecture (hierarchical continuum designs), middleware (data-flow and microservice platforms), optimization and learning (multi-agent and DRL-based placement and scheduling), and domain-specific constraints (real-time control in robotics, production, and autonomous vehicles). This research provides the conceptual and technical foundations upon which robotics-oriented platforms such as BlazeAIoT can build, by adapting generic principles of multi-layer orchestration to the strict timing, and safety constraints of distributed robotic systems.

#### **2.4.3.4 Digital Twin and Simulation for Orchestration**

Digital-twin and production-control scenarios extend these ideas to industrial settings. Nie *et al.* (2023) propose a multi-agent cloud-edge orchestration framework for digital twins in distributed production control, in which heterogeneous agents coordinate where to execute digital-twin models, prediction components (*e.g.*, LSTMs), and control logic across cloud and edge nodes. Kaneko *et al.* (2020) present a microservice-based industrial control system that spans cloud and MEC, with microservices placed across cloud and edge to satisfy control and communication constraints in 5G-enabled factories. Li *et al.* (2024) introduce a mobile additive manufacturing robot framework for smart manufacturing, which implicitly relies on orchestration of planning, monitoring, and printing services between robots, shop-floor edge, and cloud backends. Marinho *et al.* (2024) describe a multi-arm robotic platform tightly coupled with digital twins, while their focus is on design and validation, the platform presupposes orchestrated execution of simulation, estimation, and control components across local and remote compute.

#### **2.4.4 Synthesis and Implications**

The distributed robotics literature shows a clear shift from monolithic onboard computing or pure cloud robotics toward hybrid, multi-layer architectures that span robots, edge, fog, and cloud. Early cloud robotics work established patterns for offloading perception, planning, and learning to remote services, but also exposed how latency, bandwidth, and connectivity constraints limit applicability in safety and time-critical tasks. Hybrid frameworks address this by dynamically relocating tasks between local real-time execution and cloud instances based on network conditions, latency requirements, and energy budgets. However, fog and edge robotics introduce intermediate layers that host both control and learning components to reduce traffic, balance load across groups of robots, and improve robustness under failures and mobility. In parallel, DRL-based orchestrators for multi-layer fog environments treat offloading as a sequential decision problem, learning context-aware policies over energy, resource, and latency constraints. Also, edge robotics studies show that time-critical control requires co-design of control loops, compute placement, and radio scheduling. Digital-twin and production-control systems further illustrate how cloud-edge orchestration can place predictive models and controllers across the continuum for coordinating multi-robot and multi-cell systems.

### **2.5 Conclusion**

This chapter has reviewed the evolution of autonomous robotics and distributed computing from monolithic, robot-centric designs toward hybrid edge-fog-cloud continuum. Across application domains such as domestic assistance, industrial production, autonomous driving, and healthcare, robots increasingly rely on AI for perception and localization pipelines that must satisfy strict real-time and safety constraints under limited onboard compute and energy. The survey of perception and localization work highlights that DL models can achieve high accuracy, but often at execution costs that are incompatible with embedded platforms. This motivates algorithmic optimization and operationalization across heterogeneous hardware. In parallel, the distributed systems and Internet of Robotic Things literature clarify how edge, fog, and cloud

layers, combined with suitable communication middlewares, can alleviate resource limits while introducing new challenges related to latency, reliability, security, and platform heterogeneity.

To provide a consolidated view of the surveyed literature, Table 2.1 summarizes representative works across the main categories considered in this chapter. For each category, it highlights typical strengths, remaining limitations, and opportunities for improvement that motivate the contributions of this thesis. As shown in the table, there is a recurring gap in the joint treatment of real-time AI execution, multi-layer communication, and constraint-aware orchestration for autonomous mobile robots.

On top of these architectural and communication foundations, recent research on multi-layer orchestration provides mechanisms to place and migrate services across the continuum using optimization, heuristics, and deep reinforcement learning. These approaches increasingly account for heterogeneous resources, network variability, energy constraints, and application-specific QoS, yet most remain agnostic to the tight real-time and embodiment constraints that characterize autonomous robots. Taken together, these gaps motivate the contributions of this thesis:

- optimizing DL localization models for real-time execution on edge, fog, and cloud hardware;
- designing BlazeFlow as a multi-layer communication middleware that unifies heterogeneous protocols and supports high-rate robotic workloads;
- extending this data plane into BlazeAIoT, a modular, Kubernetes-based platform that integrates edge, fog, and cloud robotics under a single configuration and monitoring layer
- developing BlazeMaestro, a digital-twin-driven orchestration framework that evaluates and deploys multiobjective and learning-based placement strategies for distributed robotic services.

The next chapters build on this literature by detailing the design, implementation, and experimental validation of these contributions in realistic robotics scenarios.

Table 2.1 Summary of representative works reviewed in Chapter 2

| Category                                        | Representative works                                                                                             | Context and objective                                                                                                                           | Strengths                                                                                                                                                                           | Limitations                                                                                                                                                                 |
|-------------------------------------------------|------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Real-time AI for localization and perception    | Clark <i>et al.</i> (2017); Wang <i>et al.</i> (2017); Krishnan & Sahin (2019); Han <i>et al.</i> (2022)         | On-board or near-robot execution of deep models for localization and perception under strict latency constraints.                               | Show that optimized visual odometry and sensor-fusion models can reach real-time performance on specific embedded platforms, with detailed profiling and implementation strategies. | Generally focus on a single task and hardware setup, with limited analysis of energy, temperature, and integration into full multi-layer control pipelines.                 |
| Multi-layer communication and data distribution | Kang <i>et al.</i> (2020); Simard <i>et al.</i> (2023); Ray (2016); Samara <i>et al.</i> (2022)                  | Middleware and data-distribution frameworks for IoT and robotics across edge, fog, and cloud infrastructures.                                   | Provide abstractions for heterogeneous protocols and pub/sub communication, and demonstrate scalability for representative IoT and smart-environment scenarios.                     | Rarely evaluated with high-rate robotic sensor streams; support for multiple protocols and dynamic reconfiguration across layers remains partial.                           |
| Distributed robotics platforms                  | Hu <i>et al.</i> (2012); Quintas <i>et al.</i> (2011); Muratore & Tsagarakis (2023); Bédard <i>et al.</i> (2023) | Architectures that integrate robots with cloud/edge services and shared resources, sometimes including configuration and monitoring facilities. | Show that robots can offload computation and share maps or skills via cloud or hybrid edge-cloud platforms, and provide reusable software components, APIs, or tracing tools.       | Often limited to specific use cases or small-scale deployments, with incomplete support for integrated monitoring, configuration, and heterogeneous multi-layer topologies. |
| Orchestration and service placement             | Nezami <i>et al.</i> (2020); Kapsalis <i>et al.</i> (2017); Badidi (2019); Petri <i>et al.</i> (2021)            | Algorithms for placing services and allocating resources in cloud-fog-edge infrastructures under multiple objectives.                           | Propose formal formulations and solution methods (heuristics, metaheuristics, multi-agent schemes) with clear objectives and constraints on synthetic or IoT-oriented workloads.    | Typically ignore robotics-specific timing, wireless, and energy constraints, and seldom evaluate on realistic robotic service pipelines.                                    |
| Digital twins and evaluation frameworks         | Nie <i>et al.</i> (2023); Kaneko <i>et al.</i> (2020); Marinho <i>et al.</i> (2024); Li <i>et al.</i> (2024)     | Simulation and digital-twin frameworks for evaluating distributed systems, control strategies, and orchestration policies.                      | Enable reproducible experiments and what-if analyses under controlled network and workload conditions, reducing cost and risk compared with purely physical testing.                | Rarely coupled with live multi-cluster testbeds and detailed models of wireless mobility and robot energy use, limiting direct applicability to autonomous mobile robots.   |



## **CHAPTER 3**

### **CONTEXT AND PROJECT DETAILS**

Based on the identified gaps and on the literature review of the previous chapters, some preliminary work was required to complete the thesis. This section will detail the context for the project's execution and explain the implementation flow that led to the various publications, each addressing a gap to meet the research project's objectives.

This chapter will begin by presenting the robot used in the project, along with the modernization process that enables the legacy robot to scale to a modern, distributed-robotics-compatible system. Then, the network topology, including the edge, fog, and cloud clusters, will be introduced. This setup was essential to completing the project. Finally, each research paper will be introduced and justified to establish the project's execution flow, which will conclude with an assessment of this thesis's contribution.

#### **3.1 Robot Modernization**

The initial work for the project was introduced in Melancon *et al.* (2023). This section summarizes and updates the modernization process for the legacy robot, the Pioneer 3AT (P3AT) and details the ROS packages used for the project.

##### **3.1.1 Robot Platform**

The P3AT robot platform, from Adept, a subsidiary of Omron (Omron (2023)), was widely used in research since its introduction in 1995. The robot can be upgraded with multiple sensors that are natively supported by the programming interface provided with the robot (MobileRobots (2010)). The Advanced Robotics Interface for Applications (ARIA) provided access to various sensor data and the robot's control.

To use the P3AT in a distributed robotics solution, a modernization process was necessary. Most of the components were replaced, except for the chassis, the wheels (motors and encoders

included), and the sonar (including the circuit to access the 16 sensors). The Figure 3.1 presents a high-level view of the new hardware architecture.

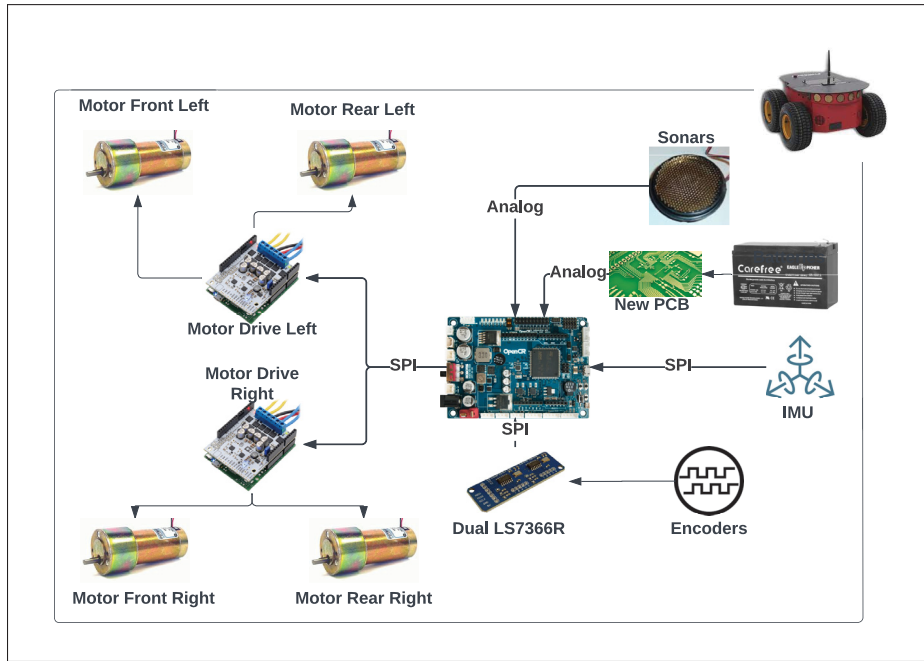


Figure 3.1 New P3AT high-level hardware architecture

To reduce the number of wires, two Printed Circuit Boards (PCBs) were designed. The first one is responsible for the power management on the robot. The original 12V batteries were kept in the design. The three batteries, connected in parallel, provide a capacity of 21.6 Ah. Two 24V batteries were added to allow the integration of the Ouster OS1 LiDAR (Ouster (2017)) and the Kinova Gen3-Lite robotics arm (Kinova (2020)). The two batteries are connected in parallel, providing a capacity of 60 Ah. Each battery power supply is sensed using a voltage and current sensor to track power usage. The 12V portion supplies the 5V required by some internal components.

The second PCB, which enables robot control, is stacked on top of the power management PCB. The goal was to fully decouple the control from the power management. The Figure 3.2 presents the robot PCBs installed in the robot. The control PCB (4) integrates the main controller (1), the motor drives (2) and the encoder interface (2). The PCB also facilitates

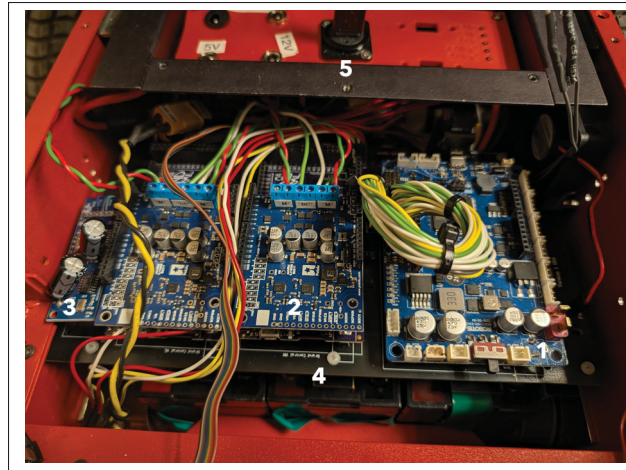


Figure 3.2 New P3AT control board

connection to the top plate (5) to provide power (5V, 12V, and 24V), a USB connection to the internal components, and LEDs to visualize the robot's state. The main controller is an OpenCR, an open-source robot controller by Robotis (Robotis (2023b)). It interfaces with the external computer through a Universal Serial Bus (USB) port using the Dynamixel Protocol 2.0 from Robotis (Robotis (2023a)), providing a communication rate of 1 Mbps. The main controller has access to the robot's various internal sensors. It retrieves the encoder values from the Dual LS7366R Quadrature Encoder Buffer from SuperDroid Robots (Superdroid Robots (2023)) via the Serial Peripheral Interface (SPI). The sensor data from the voltage and current sensors on the power management PCB are accessible as analog inputs, while the sonar array is controlled via the General-Purpose Input/Output (GPIO) to compute obstacle distances. The motor drivers are two Pololu Dual G2 High-Power Motor Driver 24v14 shields for Arduino (Pololu (2023)), mounted on each an Adafruit Grand Central M4 Express (Adafruit (2023)) board. The two Arduino-based boards are configured as SPI peripherals, connected to the main controller.

In addition to internal modernization, extra computing power and sensing capabilities were added to a stacked tower at the front of the robot, as shown in Figure 3.3. A Jetson Orin AGX (NVIDIA (2023)) compute node (1) is connected to the OpenCR through USB connectivity. A ROS node that interfaces the internal components with various ROS packages is configured on the Jetson. Section 3.1.2 details the various ROS nodes used to control the robot. A network

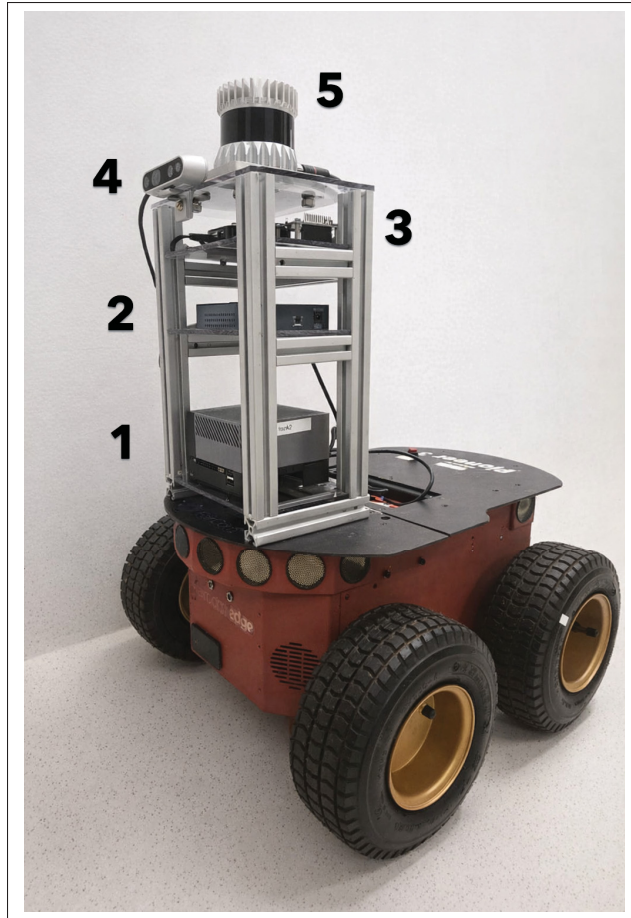


Figure 3.3 Modernized P3AT robot with new external sensors and compute node

switch (2) allows the Jetson to communicate with the LiDAR (5) and eventual components such as a robotics arm. An IMU WT901BLECL 9-axis from WitMotion (2026) is installed on the third (3) layer of the tower, along with a UWB tag MDEK1001 from Qorvo (Qorvo (2025)) for the eventual study of indoor localization with such a technology. Finally, an Intel RealSense D435i camera Intel (2023) is mounted on the top layer (4). The IMU, UWB tag, and camera are connected via USB to the Jetson.

The robot holds two roles in the multi-layer topology defined in Section 3.2. The internal sensors and controllers are considered part of the device layer since the computing resources are limited

and cannot be shared. The external sensors and the compute node are considered part of the edge layer.

### 3.1.2 Robot ROS Packages

To complete the robot's modernization, multiple software components were integrated to enable control. Figure 3.4 shows the components that were used for the project. In the figure, the green components are the ones implemented as part of the project.

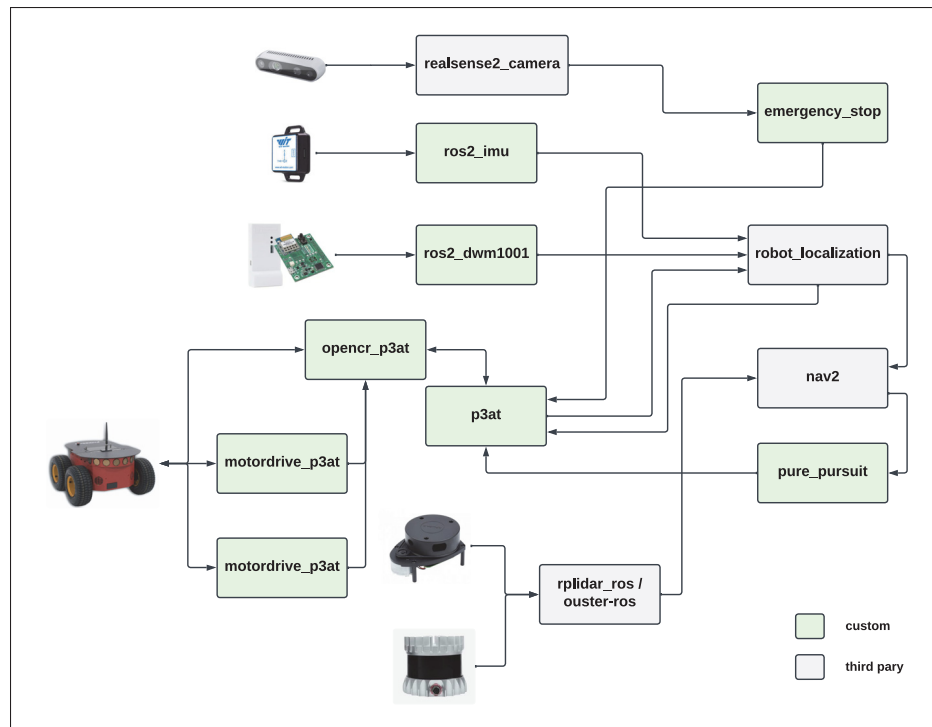


Figure 3.4 Modernized P3AT robot software components

The Intel RealSense camera is accessed with the ROS package developed and maintained by the RealSenseAI organization (RealSenseAI (2004)). For LiDAR packages, Ouster provides a ROS package for their product (Ouster (2023)), while Slamtec provides the same for their RPLIDAR product (Slamtec (2018)).

A ROS package was implemented to give access to the IMU and UWB data. For the robot itself, a ROS node was implemented with the same structure as the open-source TurtleBot3

robot developed by Robotis (TurtleBot3 (2017)). The firmware running on the OpenCR is also inspired by this project. The motor drive embedded application was developed from scratch and integrated with the OpenCR controller.

The localization is performed with an extended Kalman filter configured from the Robot Localization ROS package (Moore & Stouch (2016)), while the SLAM is integrated from the Nav2 ROS package (Macenski *et al.* (2020)), maintained by the Open Navigation Organization. Two custom ROS packages were implemented for this project. The first is an implementation of the Pure Pursuit algorithm, and the other is a simple vision-based safety stop. They are both detailed in Chapter 6.

### 3.2 Multi-Layer Topology

This section describes the network topology and cluster configuration used in the project's experimental scenarios. Figure 3.5 shows the robot within the overall distributed robotics solution.

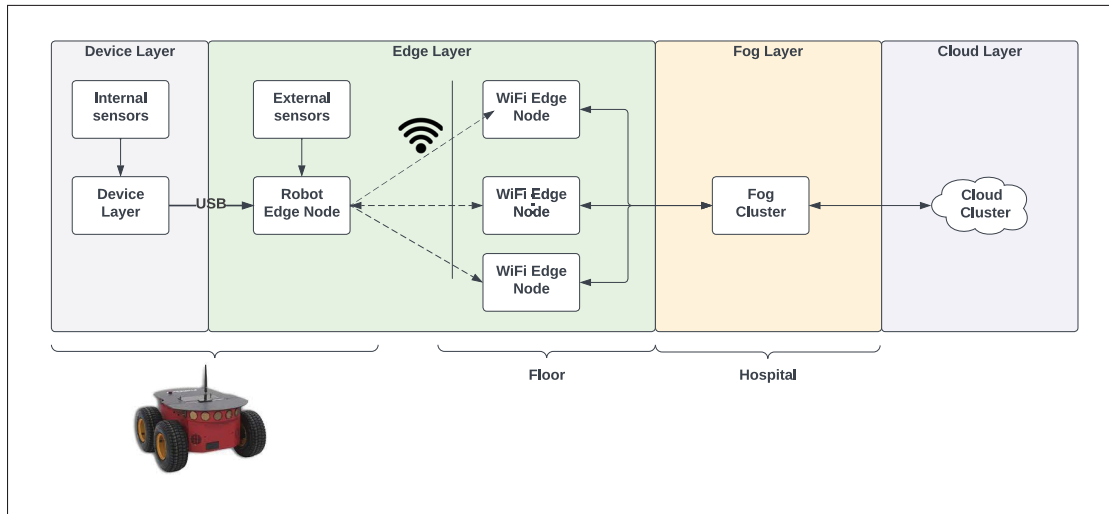


Figure 3.5 High-level architecture of the Distributed Robotics Solution

The high-level network topology is presented in Figure 3.6. A router is configured to act as a private network and is configured to access the Internet from the university network. The only network traffic outside the robot network is to fetch Docker images or data from external sources.

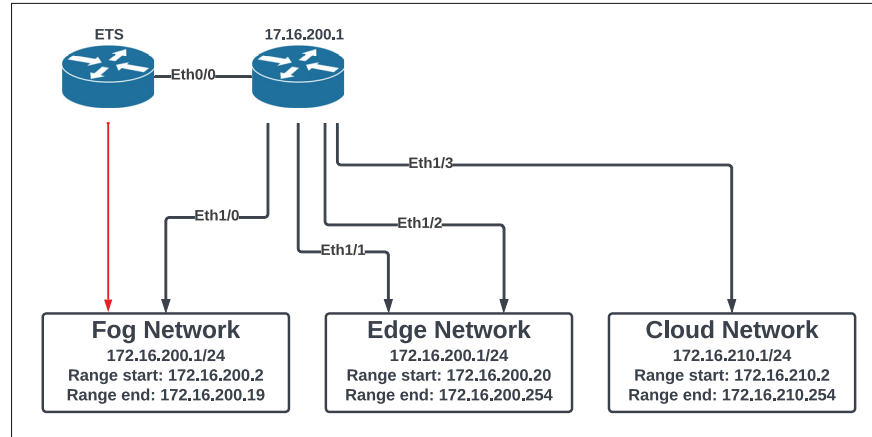


Figure 3.6 High-level network topology

### 3.2.1 Edge Layer

The edge layer is divided into two different paths as shown in Figure 3.7. The first one is connected to a network switch, and the other one to a wireless router. The wired edge nodes are distributed throughout the lab, while the wireless nodes are the robots connected via the WiFi 6 protocol.

As an example, in a hospital setting, the wired node could be installed on the ceiling of each floor and configured as a wireless access point to provide computational capabilities as close to the robot as possible.

The configuration of the edge nodes is detailed in Table 3.1. A K3S cluster (Cloud Native Computing Foundation (2026)) is configured on the edge layer with the *edge-cpu1*, *edge-cpu2*, and *edge-cpu3* nodes acting as *control plane* and *etcd* according to the Kubernetes terminology, meaning that they are used only as management nodes. The other nodes act as worker nodes, meaning they are available to execute applications.

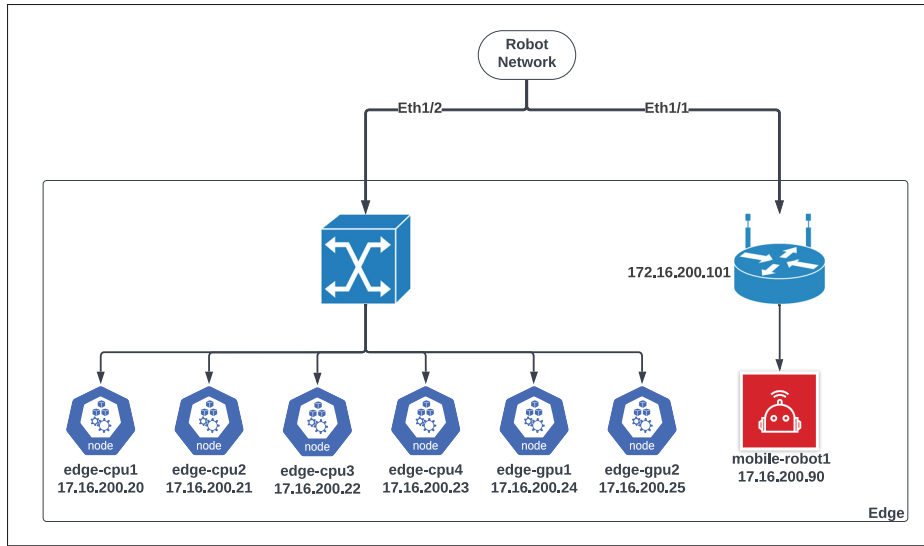


Figure 3.7 Edge layer topology

Table 3.1 Edge cluster nodes

| Suffix    | Description                   | CPU   |             | RAM       | GPU       |       |      | Storage   |
|-----------|-------------------------------|-------|-------------|-----------|-----------|-------|------|-----------|
|           |                               | Cores | Speed (GHz) | Size (GB) | Size (GB) | Cores | TOPS | Size (GB) |
| edge-cpu1 | Raspberry Pi 5                | 4     | 2.4         | 8         | -         | -     | -    | 512       |
| cpu2      | Lenovo ThinkStation P320 Tiny | 4     | 2.9         | 16        | -         | -     | -    | 512       |
| cpu3      | Lenovo ThinkStation P320 Tiny | 4     | 2.9         | 16        | -         | -     | -    | 512       |
| cpu4      | MiniPC AMD Ryzen 7            | 8     | 4.9         | 32        | -         | -     | -    | 1024      |
| gpu1      | Jetson Orin Nano              | 6     | 1.7         | 8         | Shared    | 1024  | 67   | 512       |
| gpu2      | Jetson Orin NX 16GB           | 8     | 2.0         | 16        | Shared    | 1024  | 157  | 512       |
| Total     |                               | 34    | -           | 96        | Shared    | -     | -    | 3584      |

### 3.2.2 Fog Cluster

The fog layer is composed of multiple nodes installed in a server rack in the lab. In a healthcare setting, the fog cluster could be localized to the hospital's server room. The topology used for the tests is represented in Figure 3.8. All the fog nodes are connected to a network switch, which is connected to the robot network. A proxy node is also connected to the university network, allowing remote access via Secure Shell (SSH).

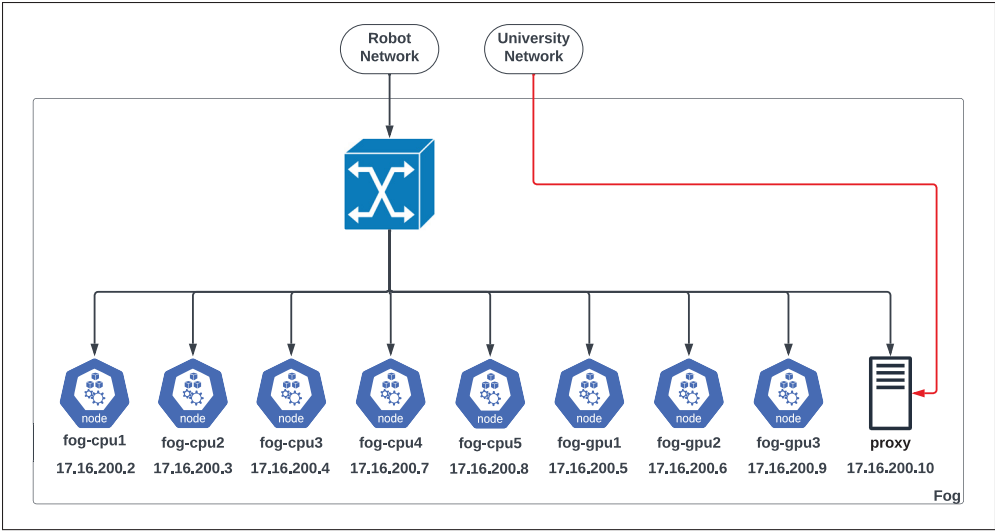


Figure 3.8 Fog layer topology

The hardware configuration of the fog nodes is detailed in Table 3.2. A Kubernetes cluster is set up on the fog layer with Rancher RKE2 (Rancher (2026)). The *fog-cpu1* and *fog-cpu4* are configured as *control plane* and *etcd*, the *fog-cpu3* has all the responsibilities, including *worker*, and the remaining nodes are configured as *worker* nodes.

Table 3.2 Fog cluster nodes

| Suffix   | Description                   | CPU   |             | RAM       | GPU       |       |      | Storage   |
|----------|-------------------------------|-------|-------------|-----------|-----------|-------|------|-----------|
|          |                               | Cores | Speed (GHz) | Size (GB) | Size (GB) | Cores | TOPS | Size (GB) |
| fog-cpu1 | UpXtreme i5                   | 4     | 1.6         | 8         | -         | -     | -    | 256       |
| cpu2     | Lenovo ThinkStation P320 Tiny | 4     | 2.9         | 16        | -         | -     | -    | 512       |
| cpu3     | Lenovo ThinkStation P320 Tiny | 4     | 2.9         | 16        | -         | -     | -    | 512       |
| cpu4     | UpXtreme Celeron 4305UE       | 2     | 2.0         | 4         | -         | -     | -    | 256       |
| cpu5     | Lenovo ThinkStation P320 Tiny | 4     | 2.9         | 16        | -         | -     | -    | 512       |
| gpu1     | Desktop i7-6700               | 4     | 3.4         | 16        | 8         | 2560  | 8.8  | 1024      |
| gpu2     | Desktop i7-6700               | 4     | 3.4         | 16        | 8         | 2560  | 8.8  | 1024      |
| gpu3     | AI Desktop i9-10980XE         | 18    | 3.0         | 128       | 48        | 10752 | 309  | 2048      |
| Total    |                               | 44    | -           | 220       | 64        | -     | -    | 6144      |

### 3.2.3 Cloud Layer

The cloud layer for the lab tests is another cluster of nodes connected to a router, configured to use a different network, as shown in Figure 3.9. In a healthcare scenario, the cloud layer could be hosted in a cloud environment from any cloud provider on multiple virtual machines or a Kubernetes Platform as a Service (PaaS).

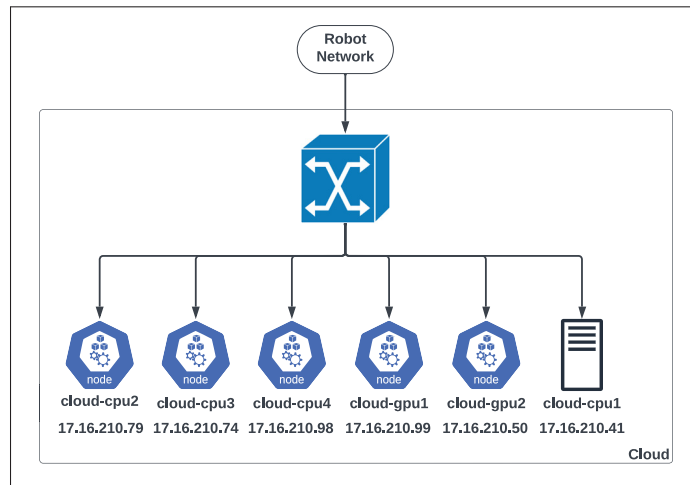


Figure 3.9 Cloud layer topology

The hardware configuration of the cloud nodes is presented in Table 3.3. The cloud-cpu1 node is configured as the Rancher server and is not part of the Kubernetes cluster's node pool. A RKE2 cluster is composed of the *cloud-cpu1*, *cloud-cpu2*, configured as *control plane*, *etcd*, and *cloud-cpu4* configured with the three roles. The remaining nodes are configured as *worker* nodes only.

## 3.3 Related Capstone Projects Supervision

This section presents the capstones projects that were performed by students from the Electrical Engineering, and the Software and Information Technology Engineering departments. These projects were part of the vision of the research and are not necessarily part of the thesis project directly.

Table 3.3 Cloud cluster nodes

| Suffix     | Description                 | CPU   |             | RAM       | GPU       |       |      | Storage   |
|------------|-----------------------------|-------|-------------|-----------|-----------|-------|------|-----------|
|            |                             | Cores | Speed (GHz) | Size (GB) | Size (GB) | Cores | TOPS | Size (GB) |
| cloud-cpu1 | Lenovo ThinkCentre M73 Tiny | 2     | 2.8         | 16        | -         | -     | -    | 1024      |
| cpu2       | Lenovo ThinkCentre M73 Tiny | 2     | 2.8         | 16        | -         | -     | -    | 1024      |
| cpu3       | MiniPC AMD Ryzen 7          | 8     | 4.9         | 32        | -         | -     | -    | 1024      |
| cpu4       | MiniPC AMD Ryzen 7          | 8     | 4.9         | 32        | -         | -     | -    | 1024      |
| gpu1       | Dell G5 5590                | 8     | 4.8         | 32        | 6         | 1536  | 5.5  | 1024      |
| gpu2       | Acer Predator Orion         | 12    | 3.6         | 32        | 24        | 10496 | 285  | 1024      |
| Total      |                             | 70    | -           | 160       | 30        | -     | -    | 6144      |

### 3.3.1 Preliminary Sensor Board Extension

The first project, completed by Kevin Yergeau, Rajat Singh Rajput and Anes Kadi, is a preliminary PCB designed to integrate the robot sensors with the OpenCR Arduino connector. The resulting PCB layout is presented in Figure 3.10. It was used on the first version of the robot, before the full modernization process.

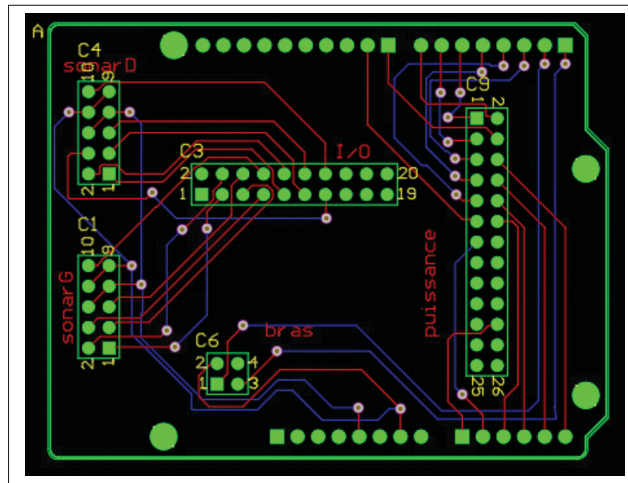


Figure 3.10 Altium PCB layout

The resulting integration in the robot is presented in Figure 3.11.

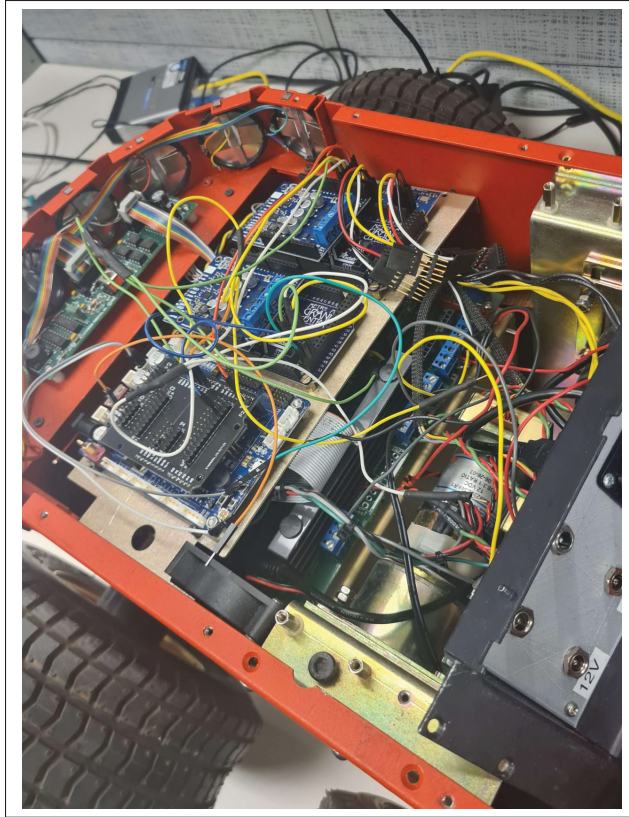


Figure 3.11 Preliminary robot modernization

The main reason for the transition to a two-board solution is the amount of cables of this solution along with the possibility to do errors while connecting the cables. The design done by the students was used as a basis for the new boards creation.

### 3.3.2 Digital Twin Simulator

To enable future navigation training, multiple capstone projects were performed by students to create a digital twin representation of the robot in a realistic simulator environment.

The first student team work on the integration of the Vortex simulator provided by our industrial partner CM Labs<sup>1</sup>. The team consisted of Jonathan Dionne-Morin, Sébastien Fortin-Ruel, Luis-Émile Lachance-Castellanos, and May Madany. In this iteration, the physical simulation

---

<sup>1</sup> <https://cm-labs.com>

was done in Vortex, while the visualization was processed in Unreal Engine. Due to the complexity of integration, CM Labs decided to stop the support of Unreal Engine, which lead to the second student team (Antoine Desbiens, Soulaha Djenabou Balde, and Sébastien Charbonneau) that did the visual integration in Unity. The students also integrated a simple crowd logic in the simulator to help with the navigation in crowded environment. The result is presented in Figure 3.12.



Figure 3.12 Robot Simulation with Vortex and Unity

In this version, the map is constructed manually based on an existing map, which can be long and difficult to have a realistic rendering. The goal is to avoid having to do a long retrain of AI model in a real environment which lead to a need to build dynamically a scene from existing robot sensor data. A third project was perform by Samuel Bolduc, Truong-Giang Phung, and Charles Simon, to solve this problem. The first part of their work was to create the scene geometry from the depth camera and the LiDAR as shown in Figure 3.13.

The next step was to apply textures on the created geometries to get a realistic scene for eventual navigation training. The result is presented in Figure 3.14. This preliminary work as a Visual SLAM is promising, but the rendering of the scene is far from being considered usable for AI model training. The project opened some interesting questions for an eventual improvement of the visual aspect of the project.

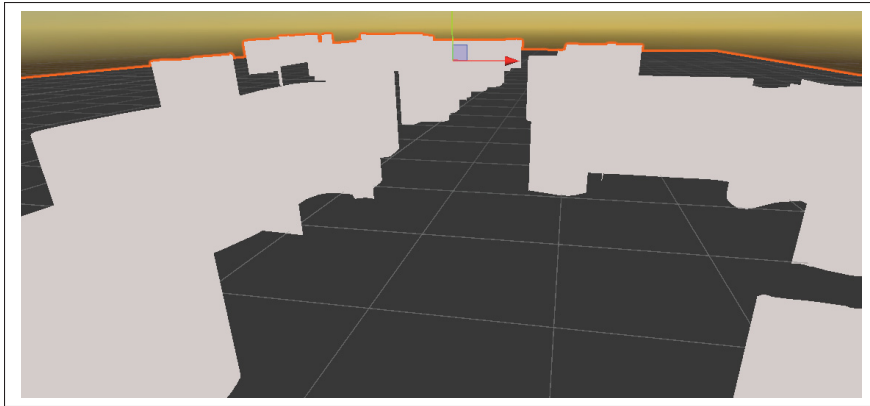


Figure 3.13 Resulting scene created from robot sensor data

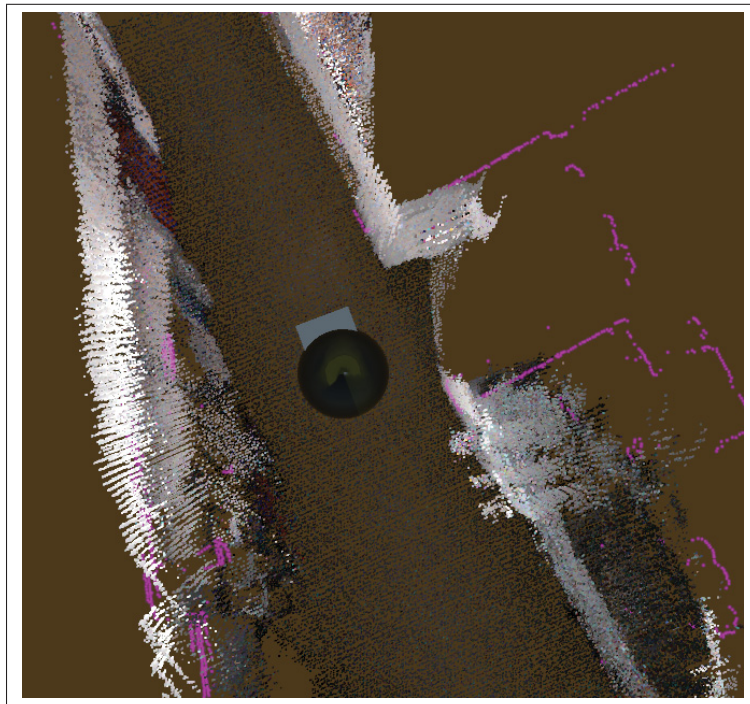


Figure 3.14 Textures applied on the created scene

Further work would be required to complete the simulator, but the actual state is promising to get a robust simulator scenario to train AI navigation models.

### 3.3.3 Management Application for a Dynamic Distributed Platform

A third project was proposed to create a management application for BlazeAIoT. A team from the new Distributed System program, composed of Mihai Marius Mandrescu, Matei Martin, Cydrick Gauthier, and Maxime Gagnière, performed this capstone project. The goal was to have a user interface to be able to see the whole solution topology and to be able to modify the configuration. Another objective was to be able to manually place a service on a given node as a preliminary work for BlazeMaestro.

Figure 3.15 presents a view of the dashboard to manage the various nodes of the solution.

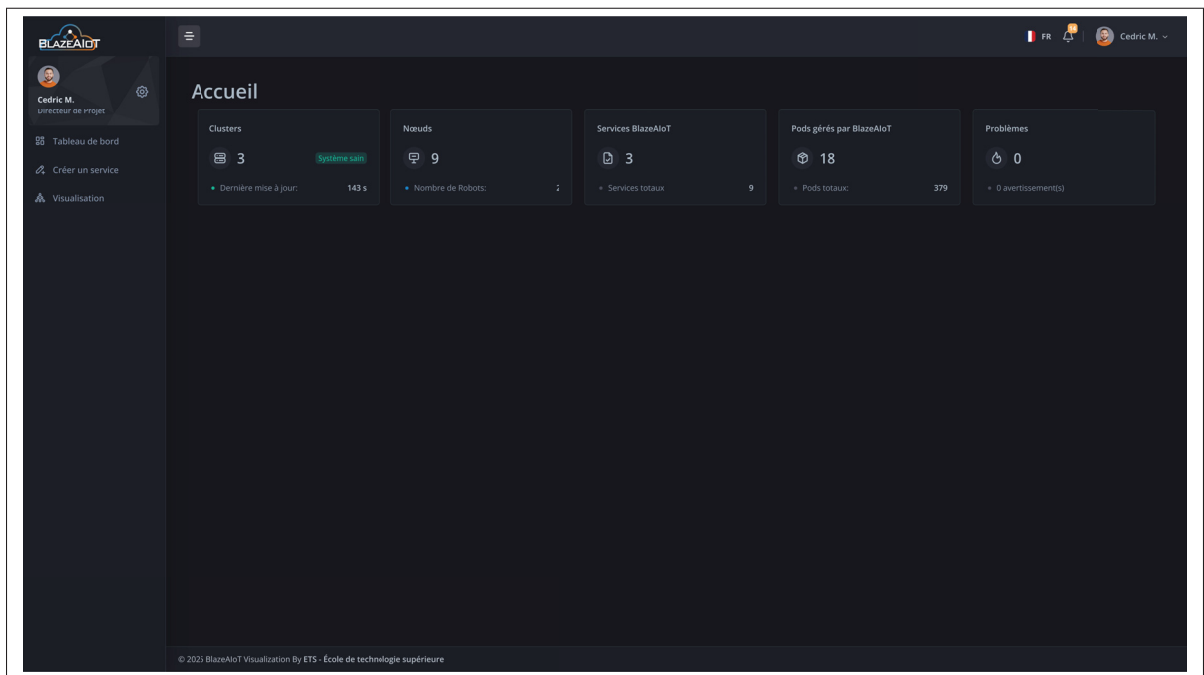


Figure 3.15 BlazeAIoT management user interface

Figure 3.16 shows a view to create or manage services in the platform. The services can include the various ROS packages needed to perform a certain robotics task.

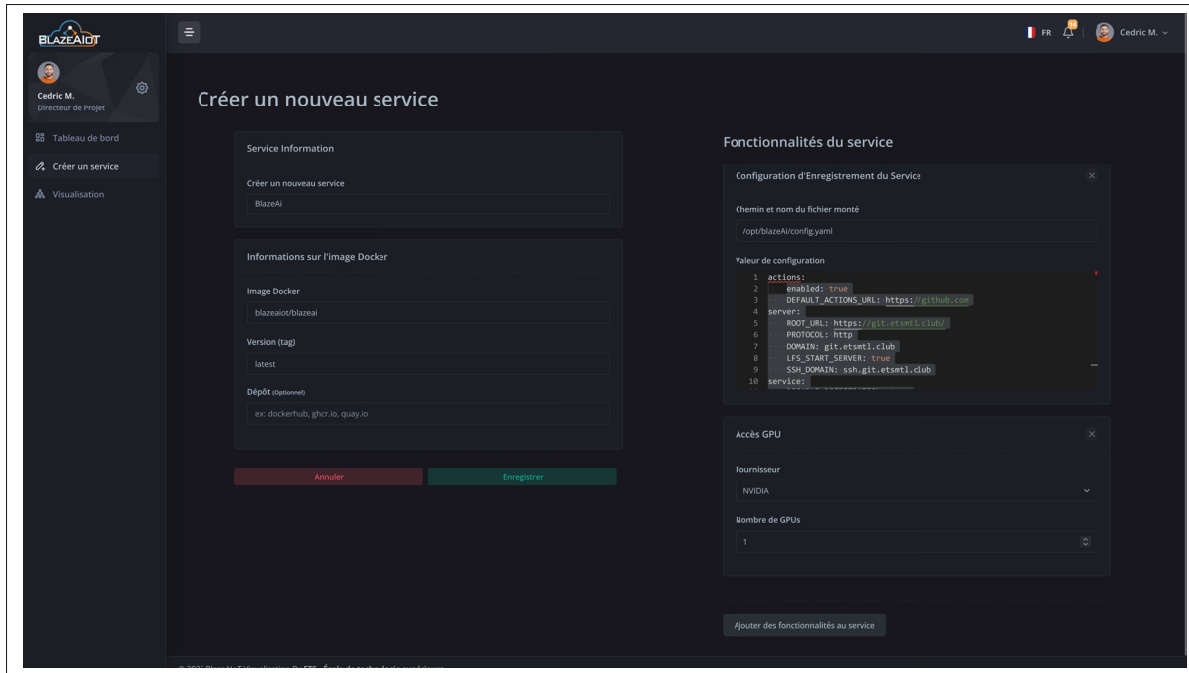


Figure 3.16 Resulting scene created from robot sensor data

### 3.4 Research Project

This section presents the context of each phases of the project that lead to the publications along with the gap covered, as presented in Section 1.1.

#### 3.4.1 Real-time AI

The first phase of the research project, following the initial modernization phase, is to study the feasibility of running complex AI models directly on the robot to address **Gap I**. A visual odometry AI model using CNN and LSTM based on camera images and IMU. The goal of this phase is to study whether it is possible to operationalize an AI model without modifying its structure. The execution time, resource usage, component temperatures, and energy consumption are analyzed under different algorithm modifications, including parallelizing the various AI model components and unrolling the LSTM loop. To increase the research value, the same

process was repeated across multiple heterogeneous hardware configurations to visualize the effect of hardware on the results.

The resulting research of this phase is detailed in Chapter 4. The main conclusion of this phase is that running this type of AI model requires significant computing power to operate in real time, resulting in high energy consumption. A good way to mitigate this issue is to distribute the model, if possible, on multiple nodes outside the robot to lower the need for high-end computing resources.

### 3.4.2 Multi-Layer Data Distribution

Optimizing the visual odometry model for real-time execution addresses only one part of the challenge. The resulting localization, perception, and control data must still be transported reliably across the distributed infrastructure. In most of the scenarios, these data streams are produced on the robot and must often be consumed or processed by services running at the edge, in the fog, or in the cloud, under strict latency and throughput constraints. Relying solely on a single middleware or on ad hoc, point-to-point links would quickly become unstable as services are added, reconfigured, or moved across layers. This motivates the need for a multi-layer data distribution framework that can systematically bridge heterogeneous publish–subscribe technologies and sustain high-rate robotic workloads across robot, edge, fog, and cloud nodes.

ROS2 is widely used in robotics and is built on the DDS protocol. Unfortunately, not all data come from robotics applications, and the limited programming languages of the DDS libraries make it difficult to integrate with some business applications. To overcome this limitation, the BlazeFlow middleware is proposed, addressing the **Gap II**.

The resulting research of this phase is detailed in Chapter 5. The resulting framework allows intra-node communication for edge nodes, intra-layer communication to share messages within a cluster layer, and inter-layer communication to inter-connect the multiple layers of the distributed robotics solution. This work was presented during the 1st International Workshop on Middleware for the Computing Continuum.

### 3.4.3 Dynamic Configuration

Having a multi-layer data distribution framework does not provide everything for executing a distributed robotics solution. To prepare for the **Gap III**, which is a need for an orchestration solution, another aspect has to be improved to complete the **Gap II**, and it is the addition of dynamic configuration. In static service distribution, the bridges from one node to another are also static. When a service can move from node to node, starting a service may require specific parameter configuration per node or layer. Also, the bridging solution of BlazeFlow needs to manage the existing bridges to dynamically enable or disable data transfer of a data between two nodes or layers. Dynamic configuration therefore allows each service to retrieve specific node layer parameters at startup and to adapt its connectivity accordingly, for example by enabling, updating, or disabling the required BlazeFlow bridges when it is started on a different node or layer, while preserving the necessary data paths across the robot, edge, fog, and cloud layers and across heterogeneous middlewares.

During the execution of this phase, it was noticed that sometimes the available bandwidth was not enough to sustain the data size and/or rate. To mitigate this issue, the BlazeFlow framework was improved to support rate limiting and filtering to reduce the burden of the network. The resulting research of this phase is detailed in Chapter 6.

### 3.4.4 Service Placement

With all the previous phases available, the last phase remaining is the service placement in a multi-layer configuration. This phase implementation addresses the **Gap III**, which is the last gap of this research. The particularity of robots in a distributed system scenario is its mobility. A dynamic orchestration is preferable, to be able to adapt to the changing parameters such as the latency. Since robots are limited in energy, the service placement should also consider the robot's available energy to optimize its autonomy.

The resulting research of this phase concludes this project and is detailed in Chapter 7.

### **3.5 Conclusion**

This chapter has established the experimental foundation of the thesis by modernizing the legacy P3AT platform and deploying a realistic multi-layer edge–fog–cloud infrastructure inspired by smart healthcare environments. It has also outlined how the resulting testbed supports the four research phases associated with the identified gaps and thesis objectives, namely real-time AI execution on the robot, multi-layer data distribution, dynamic configuration, and constraint-aware service placement. The following chapters build directly on this context by detailing each phase, presenting their experimental validation, and quantifying the benefits and limitations of the proposed approaches. Finally, it presented the supplementary work performed on the project through capstone projects that were not directly used for the research, but contributed to potential future work.



## CHAPTER 4

### IMPACT ANALYSIS OF ALGORITHM OPTIMIZATION ON ROBOT DEEP LEARNING LOCALIZATION MODEL FOR REAL-TIME EXECUTION

Cédric Melançon<sup>1</sup> , Maarouf Saad<sup>1</sup> , Kuljeet Kaur<sup>1,3,4</sup> , Julien Gascon-Samson<sup>2</sup>

<sup>1</sup> Department of Electrical Engineering, École de Technologie Supérieure, Montréal, Québec, Canada

<sup>2</sup> Department of Software and IT Engineering, École de Technologie Supérieure, Montréal, Québec, Canada

<sup>3</sup> Department of Electrical Engineering, Canadian University Dubai, Dubai, UAE

<sup>4</sup> Centre for Research Impact & Outcome, Chitkara University Institute of Engineering and Technology, Chitkara University, Punjab, India

Paper published in the journal « IEEE Access » on November 26th, 2025

#### 4.1 Abstract

The increased demand for autonomous robots in industries such as healthcare, manufacturing, and logistics is driven by the need to address labor shortages and enhance operational efficiency. A critical aspect of these robots is their ability to navigate complex environments autonomously, which relies heavily on multiple artificial intelligence models, including visual odometry. Visual odometry enables robots to estimate their motion by analyzing visual data, making it essential for navigation and obstacle avoidance. However, ensuring such models operate within real-time constraints is paramount for the robot's responsiveness, safety, and overall functionality. The complexity and depth of such a model, which utilizes camera images and other sensor data, make it challenging to achieve real-time performance. Thus, this chapter thoroughly evaluates the performance of a DL visual odometry model, with a focus on its execution time and computational overhead. Furthermore, it proposes an optimized implementation to meet the stringent real-time requirements for safe and efficient robot operation. The optimization is achieved through algorithm improvement to preserve the model structure and accuracy. Multiple hardware platforms were utilized to demonstrate the resource differences between edge, fog, and cloud deployments, thereby validating the effect of the proposed model optimization. The resource usage is also compared to see the impact of the modification on multiple aspects, including

computation, memory, temperature, and energy. The resulting model operationalization improves the execution time while highlighting limitations for some hardware configurations.

## 4.2 Introduction

Over the last decades, robotics has shown promising evolution, and autonomous robotics solutions are increasingly present in our lives. Many industrial fields benefit from the use of such robots. In construction, robots are used to install ceiling boards, as described in Hachijo *et al.* (2024), which is a very demanding task for humans due to the weight of the boards and the necessity of working at heights. The industrial sector also uses autonomous robots to transport components from one workstation to another. Authors in Li *et al.* (2024) propose a robot framework that enables tasks such as 3D printing parts while navigating to optimize production. Robots can also be used in agriculture to identify, localize, and pick fruits as detailed in Yin *et al.* (2023). Identifying potential diseases in the orchard and sorting out problematic fruits is possible with artificial intelligence (AI) Singh *et al.* (2024). Finally, healthcare is another field that has started to leverage robotics. Authors in Bakshi *et al.* (2022) explained the adoption of robotics in the healthcare sector to perform various roles, such as frontline workers, logistics, escort, manufacturing, and surveillance.

Different AI models can be leveraged to perform such industrial applications, but they require significant computing power. However, running all computational tasks locally on the robot increases its cost manifold, primarily due to the miniaturization of the electronic components that must fit within the robot's body. Moreover, the miniaturization of electronics increases fabrication complexity and the likelihood of thermal issues. Additionally, such computing power is highly energy-intensive, which poses a problem for battery-operated robots Anshul & Mohan (2022); Valtronic (2020). To circumvent these challenges with hosting computationally intensive tasks on robots, the *Internet of Robotic Things (IoRT)* has emerged as a potential solution. As introduced in Ray (2016), IoRT lays the foundations for distributed robotics, enabling it to benefit from external data sources and enhance the robotics solution. It uses cloud computing technologies to provide distributed computing capabilities for data from Internet of Things (IoT)

sensors. For instance, authors in Saravanan *et al.* (2018) proposed applying this concept to monitor the health of indoor plants by integrating environmental sensors with robot observations.

The use of cloud resources implies a high communication latency, as stated in Seisa *et al.* (2022). Many algorithms or models require real-time constraints to ensure the safety of autonomous agents, which is not compatible with high-latency solutions. Traffic sign detection (Chen *et al.* (2022a)), navigation (Kruse *et al.* (2013)), and control (Di & Shi (2021)) are examples of features that AI models solve, necessitating fast responses. Authors in Terven *et al.* (2023) compared the accuracy of various implementations of the You Only Look Once (YOLO) model. It can be stated that larger architectures generally achieve higher accuracy but at the cost of longer execution times. Deploying such a model in a real-time, time-critical system requires carefully balancing execution time and accuracy. Furthermore, larger models, characterized by greater depth and a higher number of neurons per layer, require more computational resources, resulting in increased energy consumption.

Image processing models are good examples of such large models. Processing camera feeds is computationally intensive and plays a crucial role in various functionalities of autonomous agents, including robots and vehicles. It is used for positioning and movement control (Chuprov *et al.* (2020)) or obstacle detection (Vu *et al.* (2024)). For instance, the Simultaneous Localization and Mapping (SLAM) algorithm has been used to explore an environment to build a map that can be used for robot navigation (Chen *et al.* (2023b); Pareigis *et al.* (2023)). Visual odometry is a feature used in robotics to estimate the pose and localization of the robot using image processing. Authors in Wang *et al.* (2017) proposed a DL approach for visual odometry through sensor fusion of Inertial Measurement Unit (IMU) with color camera images. Other sensors can also be used for indoor localization, such as Ultra-Wideband (UWB) sensors (Bach *et al.* (2024)) or Light Detection and Ranging (LiDAR) sensors (Nimura *et al.* (2023)). The resulting localization is also used for robot motion through motor control. These time-critical tasks ensure the robot avoids obstacles or stops when they are detected. All these robotics features utilize multiple sensors that generate a large volume of data, presenting a challenge for real-time processing.

Most of the algorithms mentioned above are implemented for the Robot Operating System (ROS) middleware (Macenski *et al.* (2022)), a widely used framework in robotics solutions. It offers a distributed approach to performing tasks, with sensor nodes providing telemetry data and controller nodes utilizing sensor data to perform tasks such as motor control, localization, navigation, and more. The proposed work utilizes ROS to execute an AI model on sensor data input, leveraging all the features available in the framework.

#### 4.2.1 Motivation

This chapter aims to execute a resource-demanding AI model, such as DL-based Visual Odometry, into a robotics solution with real-time constraints without impacting the model's accuracy. The execution time improvement is achieved through algorithm modification, rather than modifying the model size, which can affect its accuracy. The goal is to keep the model structure and parameter values of a proven working DL model to improve execution time and meet real-time constraints.

Knowing precisely a robot's three-dimensional position and orientation is the foundation of any robotics solution. These values can be used through derivatives to obtain the robot's linear and angular velocities or accelerations with the second-order derivative. The obtained results are used for robot control and navigation. The execution rate of such algorithms depends on multiple factors, including the robot's speed, responsiveness, and accuracy. The faster the rate, the faster the robot can react and the more stable its motion will be. When dealing with kinematic control (based on position, velocity, and acceleration), the required execution rate is slower than that of dynamic controllers (using wheel torque, friction, and all dynamic aspects of the robot), as specified in Sciavicco & Siciliano (2000).

Dynamic controllers are more sensitive to variations and need faster reactivity to ensure smooth motion. Such controllers are more flexible but more complex to model and implement, and require more computing power than kinematic controllers. For kinematic control, increasing the execution rate increases responsiveness and accuracy, but also increases computational needs.

If the robot's speed of operation is slow, the need for responsiveness is lower. The rate for slow-moving robots (0.1 to 0.5 m/s) can be 10-20 Hz (MohandSaidi & Mellah (2022)), while it can go up to 100 Hz (Chen *et al.* (2018)) for fast-moving robots (higher than 1.0 m/s). Given that an autonomous robot navigating a crowded environment cannot move swiftly, we consider a maximum velocity of 0.75 m/s, which justifies a rate of 30 Hz, representing the real-time constraint for the visual odometry model execution.

This chapter leverages an existing DL model for robot localization proposed in Zhang *et al.* (2022) and available on GitHub, which serves as the baseline for this work. It uses data from the IMU and color camera images. Since image processing is the model's most time-consuming component, the image rate determines the model's execution speed. Therefore, the selected camera frame rate is 30 Hz, and the model is applied to every received camera image.

#### 4.2.2 Related Work

AI processes are divided into two operations, namely training and execution (or inference). Both operations are not necessarily intended to be performed on the same computer configuration, as training typically requires more memory and computing resources to determine the best parameters for achieving the desired accuracy using a large amount of data from a given dataset. The resulting network parameters are used to execute the model inference on a single set of inputs.

Table 4.1 presents multiple references to real-time scenarios solved through AI models. It is essential to note that while authors discuss real-time constraints for AI models, they refrain from precisely quantifying the timing objectives. It becomes difficult to assess if the resulting timings claimed (when provided) are valid for such a real-time scenario. The use cases necessitating real-time execution include autonomous vehicle scenarios (such as collision detection, traffic sign detection, intrusion detection, and traffic monitoring), crowd-related scenarios (facial recognition, human pose estimation, and crowd enumeration), and healthcare applications (such as prostate cancer detection). The implemented models utilize well-known

Table 4.1 Real-time model references

| Use case                                           | Reference                      | Model type     | Methodology                                       | Timing                 |
|----------------------------------------------------|--------------------------------|----------------|---------------------------------------------------|------------------------|
| Traffic sign detection                             | Barodi <i>et al.</i> (2022)    | CNN            | Hyperparameter tuning                             | Not measured           |
| Traffic sign detection                             | Avramovic <i>et al.</i> (2020) | CNN            | Region of interest segmentation                   | Not measured           |
| Traffic monitoring                                 | Barbosa <i>et al.</i> (2023)   | Sparse CNN     | Model optimization with sparse network            | 50% reduction          |
| Facial recognition                                 | Munim & Rahman Kohinoor (2023) | CNN            | Various model comparison                          | 18.1 ms/image          |
| Intrusion detection                                | Alladi <i>et al.</i> (2021)    | LSTM, CNN, MLP | Model architecture comparison                     | 14.29 ms/image         |
| Human Pose Estimation                              | Shah <i>et al.</i> (2023)      | CNN            | Model comparison                                  | Not measured           |
| Crowd enumeration                                  | Kounte <i>et al.</i> (2023)    | CNN            | Model modification                                | Not measured           |
| Collision detection                                | Vu <i>et al.</i> (2024)        | CNN            | Execution pipeline                                | 1.52 s/test            |
| Prostate cancer detection                          | Natali <i>et al.</i> (2025)    | Transformers   | Model modification                                | 993 queries per second |
| Robot Simultaneous Localization and Mapping (SLAM) | Pudasaini <i>et al.</i> (2024) | CNN            | Pruning and quantization                          | Not measured           |
| Robot Control                                      | Paniego <i>et al.</i> (2024)   | CNN            | Pruning, quantization, fine-tuning and clustering | 15-79 Hz               |

neural networks, including Convolutional Neural Networks (CNN), Multilayer Perceptrons (MLP), Recurrent Neural Networks (RNN), such as Long Short-Term Memory (LSTM), and transformers. The methodology for achieving the desired execution time involves hyperparameter tuning, utilizing sparse networks, comparing multiple model architectures to select the optimal

one, and integrating model execution into a pipeline to process model predictions. Another approach is to divide the full-scaled image into smaller images to reduce the model size and increase the execution time. It is also possible to parallelize multiple regions when multiple GPUs are available. The timing analysis depends on the model's usage and the type of input. All of these works utilize color images captured by a camera, except for the prostate cancer detection scenario, which employs ultrasound images. The latter advertises fast model processing, which can be attributed to the grayscale image and the small model size.

Pruning and quantization techniques are discussed in Pudasaini *et al.* (2024). The authors successfully optimized the SLAM model by up to 79.8% while keeping its accuracy by retraining it with fine-tuning techniques. Authors of Paniego *et al.* (2024) compared various model optimization techniques, including pruning, quantization, fine-tuning, and clustering, across different AI frameworks, such as PyTorch and TensorFlow, as well as their optimized variants. Their analysis reveals that, depending on the framework, specific optimization techniques are more effective than others, yet can significantly improve inference time with minimal impact on model accuracy, and sometimes even yield better results.

Another essential factor for successful autonomous robotics solutions is their energy efficiency (Chinmay & Nisha (2024)). Since the robots are battery-operated, particular care must be taken to minimize the computing and memory requirements of the designed algorithm. An analysis of the impact of AI on energy consumption is detailed in Bültemann *et al.* (2023), and the obtained results highlight that model choice significantly impacts energy consumption (from basic machine learning algorithms to complex neural networks). Further, the programming language used for implementation can also influence energy consumption. Thus, particular attention should be devoted to selected models that are reliable in accuracy and energy-efficient.

Another essential factor impacting the robot's energy consumption is the resource type chosen for the model execution, *i.e.*, Graphics Processing Unit (GPU) and Central Processing Unit (CPU). Authors in Caspart *et al.* (2022) compared the energy consumption of training and inference execution on GPU and CPU. Even if the GPU consumes significantly more energy

than the CPU, the gain in execution time makes it more energy efficient than the CPU for a particular task. They also highlighted that a GPU consumes significant power even when idle. This means that when using GPU-based systems for non-AI workloads, the energy consumed by idling GPUs considerably impacts energy efficiency. A model partitioning solution is proposed in Kim *et al.* (2020), which distributes the model on GPU and CPU for edge devices to optimize the balance of energy consumption and execution time.

Different approaches are discussed in the literature to optimize AI models that reduce energy consumption, including bagging algorithms, pipelining, and dynamic programming. Authors in Cassales *et al.* (2023) presented a bagging algorithm to parallelize computation using mini-batches, thereby improving energy consumption while minimizing the impact on model performance. Josyula *et al.* (2019) proposed a pipeline solution to perform point cloud segmentation to optimize run-time execution. A downside of pipelining is the addition of communication overhead, which affects the overall execution time. Additionally, the segmentation results in data reduction, which improves execution time. Li *et al.* (2022a) proposed a dynamic programming approach to distribute processes on both GPU and CPU on mobile devices. Their results depicted better efficiency than using only a GPU. Along similar lines, authors in Niu *et al.* (2020) explored compiler optimization to improve model execution time, demonstrating faster execution times on both CPU and GPU while having a minimal impact on model accuracy.

Unlike the optimization techniques discussed, which focus primarily on model architecture changes or hyperparameter tuning, this chapter's proposed approach preserves the model structure and, instead, optimizes the execution pipeline and deployment strategy through algorithmic optimizations.

### 4.2.3 Contributions

Since the work relies on a real-world scenario based on visual odometry, this chapter defines the execution time limits and proposes an optimized model to meet these constraints. The main contributions of this chapter are:

- *Definition of the real-time constraint* with justifications of an autonomous robot localization module.
- *Multiple phase algorithm optimization* of a Visual Odometry DL model, including the adaptation of the model in ROS.
- *Impact analysis* of the various phase optimizations on the execution time, resource usage, and energy consumption.
- *Heterogeneous hardware configuration comparison* to model the execution at different layers (edge, fog, or cloud computing).

It is important to emphasize that the structure of the Visual Odometry DL model was intentionally left unchanged in this study. While some basic hyperparameter tuning was performed to achieve reasonable accuracy, our primary objective was to evaluate and optimize the model's execution pipeline for real-time performance, rather than to maximize accuracy through extensive hyperparameter tuning. The goal is to establish a model baseline for evaluating the impact of our proposed work on inference execution time, aiming to achieve a 30 Hz frame rate. Further work will be conducted in the future to enhance the model's accuracy and, by applying the proposed algorithm, to meet both objectives of accuracy and real-time processing.

#### **4.2.4 Organization**

This chapter is organized as follows. Section 4.2.2 provides an overview of the related work. Section 4.3 introduces the baseline visual odometry model used for this work and explains the process used for the AI model training to obtain the results for this chapter. Section 4.4 explains the various optimization phases applied to the model to operationalize it and meet real-time criteria. Section 4.5 evaluates the performance of the optimized model and provides an analysis of the resources used for execution, along with a discussion of the various results. Finally, a conclusion presents potential areas for improvement in future work.

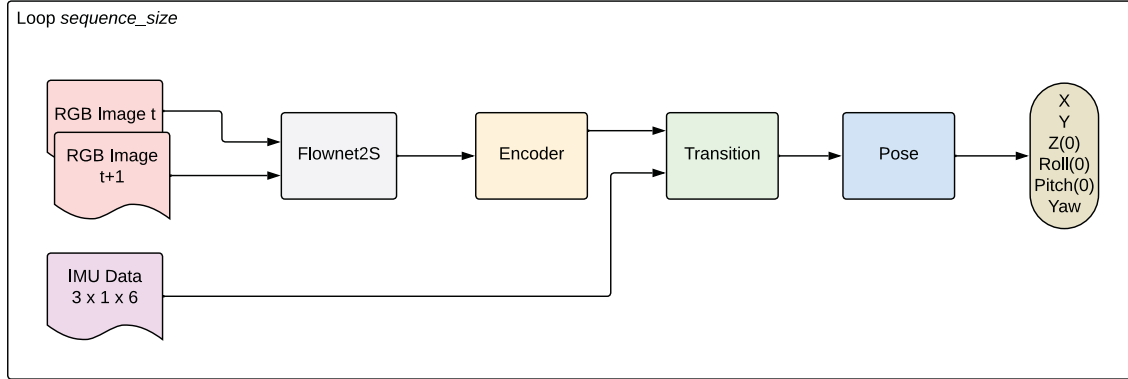


Figure 4.1 Deep Learning Visual Odometry model

### 4.3 Visual Odometry Model

The concept of odometry uses data from various sensors to estimate the position and orientation variation of the robot. By itself, it does not directly provide the robot's localization. Still, when the original robot's position is known, the global position can be computed by applying the relative motion to the previous localization.

Visual odometry provides the same relative motion but utilizes visual data to enhance the accuracy. This data can originate from sensors such as camera feed (color or depth images) and LIDAR point cloud. The high-level structure of the visual odometry model, proposed in Zhang *et al.* (2022) and used as the baseline for this work, is detailed in Fig. 4.1. It performs sensor data fusion using data from an IMU and an RGB Camera to predict the robot's relative position and orientation. This predicted localization can feed other services, such as motor control or navigation models.

This model is divided into multiple submodels. The first submodel is responsible for the feature extraction using the Flownet2S model introduced by Ilg *et al.* (2017). Flownet is a DL-based optical flow algorithm that takes two frame images of a movement and detects the motion variation from one frame to another. Since the visual odometry model uses the Flownet only for feature extraction, the last dense layer from the model is removed. If the camera frame rate is 30

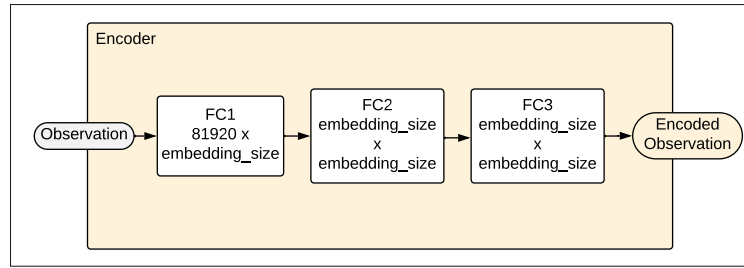


Figure 4.2 Encoder model

Hz, the first image captured does not produce any output data; however, the subsequent frames are used in pairs with the previous frame as input to the feature extraction submodel.

The resulting data, labeled *Observation*, is pushed into the next submodel, acting as an encoder. The encoder is detailed in Fig. 4.2. It is designed to help reduce the data size used for the remaining model through three dense layers. The resulting data reduction is labeled *Encoded Observation*.

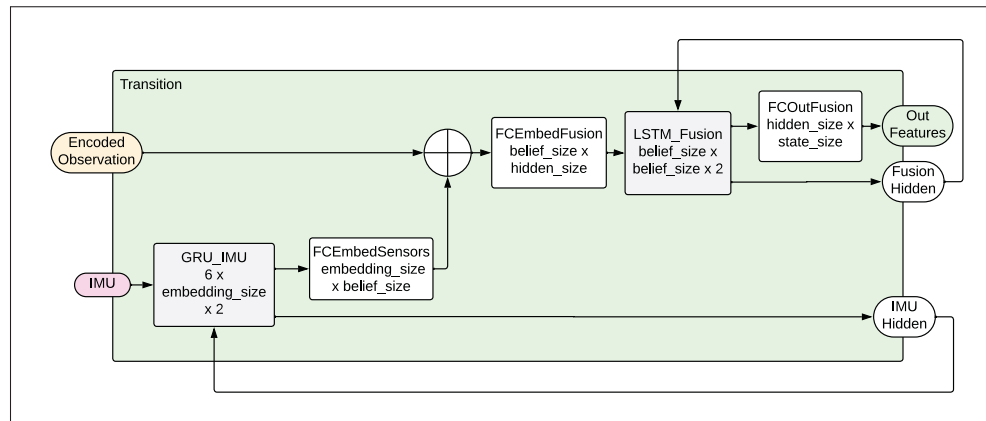


Figure 4.3 Transition model

The transition submodel following the encoder is represented in Fig. 4.3. It is responsible for the sensor fusion using the *Encoded Observation* and the data originating from the IMU sensor. A sensor rate of 100 Hz is considered for the model design, meaning that three IMU frames are received between two camera frames. This data is processed using Gated Recurrent Units (GRU) RNN Cho *et al.* (2014). The output of the RNN is provided to a dense layer to scale

the result data size and is then concatenated with the encoded image before being used in an LSTMHochreiter & Schmidhuber (1997). The idea behind the two RNNs is to leverage the previous data from the sequence over time. This is performed through a closed loop, and the resulting RNN hidden layer data is reinjected in the subsequent execution as a memory of the past. Two fully connected layers follow to provide features resulting from this sensor fusion, labeled *Out Features*.

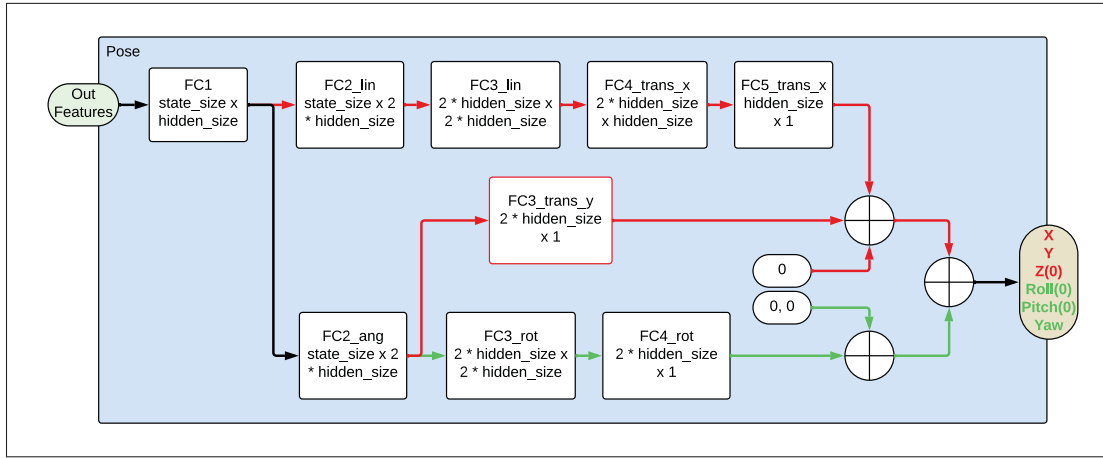


Figure 4.4 Pose model

Finally, a pose model, as described in Fig. 4.4, performs pose prediction through a succession of dense layers. They are divided into two distinct paths. The first path (in red) predicts the position in two dimensions (since the robot is moving on flat floors, the Z coordinate is considered to be zero) with a minor influence from the angular portion on the Y coordinate. The second path (in green) predicts the yaw (rotation around the Z-axis) while the roll (X) and pitch (Y) are considered zero due to the 2D motion and to reduce the complexity. The result is the relative variation of the position/orientation between two frames.

The obtained relative position and orientation are converted into a transform matrix and used to determine the new predicted robot's global position. It is expected that the initial position has to be known for this algorithm to work.

### 4.3.1 Deployment

Over the past few years, the approach to deploying robotics solutions has undergone significant evolution. Initially, the computation was only located on the robot platform. With the increase of AI models to execute robot operations, such a solution has limitations regarding available resources and energy. The concept of Cloud Robotics, as introduced in Mello *et al.* (2023), addresses resource availability but introduces significant communication latency, rendering it ineffective for real-time scenarios. Resources used in the cloud can be virtualized to match computing requirements and scaled as demand grows.

Fog Robotics (Casamayor Pujol & Dustdar (2021)) brings the computation closer to the user, reducing the latency and making it more accessible for critical scenarios. The amount of resources is more limited than the cloud, but it can include computers with high-end resources. A compromise can be made using Edge Robotics, which brings computing power to the edge of the robotics solution with viable communication latency for real-time constraints. While Edge computing holds more resources than robots, it is limited compared with Fog capabilities.

This chapter compares the various hardware configurations representing the deployment types (Edge, Fog, and Cloud).

### 4.3.2 Model training

The project's use case is based on indoor mobile robotics. Accordingly, the MIT Stata Center Dataset (Fallon *et al.* (2013)) was selected for training the model. The dataset is derived from an indoor scenario involving a robot equipped with components similar to those available in our lab. It also has the advantage of providing the data in a ROS bag file representing a real-time recording of a sequence. This sequence can be replayed in ROS to facilitate the inference evaluation in a near-reality environment. Specific ground-truth labeling is available for multiple sequences, facilitating supervised training. The data rate for the IMU is 100 Hz, and the camera feed runs at 30 Hz, as designed in the model.

The accuracy obtained through the model's training is not optimal, with an accumulated error of around 20 cm in X, 8 m in Y, and 20 degrees in theta. The primary factor contributing to the accumulated error in Y is the orientation error. While more extensive hyperparameter tuning could reduce these errors, such optimization was beyond the scope of this study, which focused on improving the execution pipeline and, therefore, the operationalization of the model, rather than refining the model. Another factor that can impact the model's accuracy is the image size. While the MIT Stata Center dataset provides 640x480 images, the original model was trained on the KITTI dataset (Geiger *et al.* (2012)) with 1382x512 images, resulting in more extensive training data.

While our experiments focus on the MIT Stata Center dataset due to its compatibility with our hardware and ROS-based pipeline, the operationalization strategies proposed are applicable to other datasets, such as KITTI (outdoor vehicle scenario) and EuRoC (three-dimensional aerial vehicle), which are used for training in the original chapter. Each dataset presents its own particularities in terms of vehicle velocity, indoor or outdoor environment, or two or three-dimensional navigation. They all provide sequences with camera images and IMU data. Since the aim of the project is indoor two-dimensional navigation, the characteristics of the MIT Stata Center dataset are more suitable, and no generalization is necessary for other scenario types.

## **4.4 Model Operationalization**

The model's operationalization was performed in three phases. This section details them and discusses their potential impact on the model's execution time.

### **4.4.1 Phase 1: ROS conversion**

The code baseline, implemented in Python with the PyTorch framework, is incompatible with ROS. The first phase involves converting the visual odometry model to make it compatible with the ROS framework. A ROS node can combine the concepts of publishers sending data,

subscribers receiving data, and services performing client-server actions. This conversion is mainly necessary to leverage the replay capability provided by the ROS bag file. Another motivation for this phase is that ROS is widely used in the robotics industry. This phase involves no changes to the original model execution, but rather the method for providing the dataset to the model (from a regular PyTorch dataset to a ROS subscriber). The ROS2 Humble distribution was used for this work.

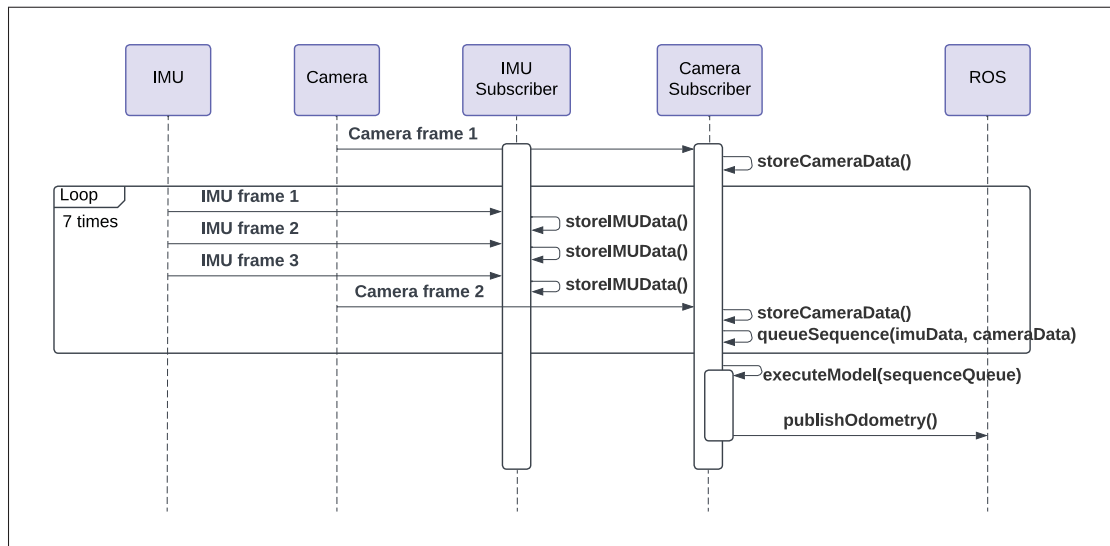


Figure 4.5 Phase 1 Sequence Diagram

The resulting sequence diagram is represented in Fig. 4.5. Each sensor stream (IMU and RGB Camera) is handled by a dedicated ROS subscriber running in separate threads. Every IMU and Camera data is stored in a queue in a loop over seven frames (*sequence\_size*). Mutexes are used to synchronize access to the shared 7-frame queue, ensuring that only one thread can modify the queue at a time. When an image is received, the process copies the last three IMU values and the received camera image into a sequence queue. The whole model can be performed when the queue has seven sequence frames. A mutex also protects the model execution to prevent concurrent predictions. The model is implemented precisely as described in the base code, which computes the prediction by looping over seven frames, performing the Flownet, Encoder, Transition, and Pose submodels for each sequence execution, as represented in Fig. 4.1.

ROS can be executed in a single-threaded or multi-threaded fashion. The Phase 1 optimization leverages multi-threading to perform every message reception in a separate thread. ROS also offers two options for thread execution, namely, mutually exclusive and reentrant execution (Open Robotics (2025)). The concept of mutually exclusive execution is similar to protecting the thread with a mutex, ensuring that a thread completes its execution before starting the execution of the following thread instance. Reentrant execution enables the parallel execution of processes, leaving the developer responsible for ensuring thread-safe execution. The latter option is implemented for this optimization phase to simplify the implementation of the subsequent improvements. This implementation is very similar to the mutually exclusive thread execution, except that storing camera image data can be executed simultaneously with the model execution.

Since the model execution is protected, an overrun (execution time over the 30 Hz framerate) has a direct impact on the validity of the data. The synchronization between the RGB camera and the IMU would be broken, resulting in invalid predictions, which is considered a failure.

#### **4.4.2 Phase 2: Parallelization**

The idea behind Phase 2 optimization is to remove the Flownet submodel from the overall model execution, allowing for parallel processing. While in the previous phase, we introduced multithreading, its only benefit is that it enables sensor data acquisition (since a mutex protects the execution of the whole model). The mutex acquisition difference between phases 1 and 2 is represented in Fig. 4.6. The fact that the mutex acquisition in phase 2 is pushed after the Flownet submodel allows it to be executed in parallel to the remaining model execution.

Instead of waiting for the entire visual odometry model to be completed, the Flownet submodel execution is initiated by the ROS subscriber upon receiving an image, using the two previously received frames. If the odometry prediction comes before the end of the Flownet execution, the new prediction will start. Otherwise, it will wait for the process to be completed. The main reason this modification is possible is that Flownet is completely decoupled from the remainder of the model, focusing solely on camera images.

This simple modification can significantly improve execution time due to parallelization, but it is expected to consume more resources for the same reason.

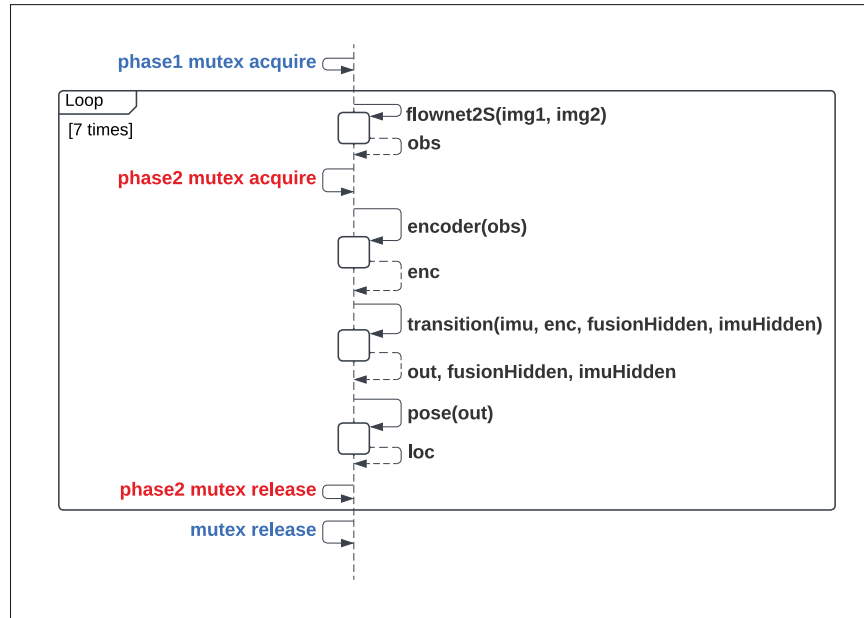


Figure 4.6 Model execution sequence diagram

For this phase, an overrun of the Flownet execution would break the synchronization with the IMU, meaning that the three latest received IMU frames would not be aligned with the pair of camera images, which would be considered a failure.

#### 4.4.3 Phase 3: Loop unrolling

The Phase 3 implementation continues to execute the visual odometry model over seven sequence frames at each camera frame reception. The loop unrolling optimization aims to leverage the RNN concept appropriately. When the GRU or LSTM network is executed, it provides the resulting tensor and the values of its hidden layers, representing its internal memory. The values of these hidden layers can be injected as input for the subsequent execution, meaning that there is no added value to computing the result over the entire sequence, since only the last values are used. Sequence is essential for RNN, mainly for the training process. The initial hidden layer

values can be randomized or set to zero, and this initialization is considered a hyperparameter for the training process.

To improve the execution time, this loop is unrolled, meaning the hidden layers (*Fusion Hidden* and *IMU Hidden* in the transition model) are stored for the subsequent model execution. The sequence diagram in Fig. 4.7 illustrates the execution of the visual odometry model, including the loop unrolling optimization. The thread synchronization mutexes are kept based on the Phase 2 implementation to ensure parallelization of the FlowNet submodel.

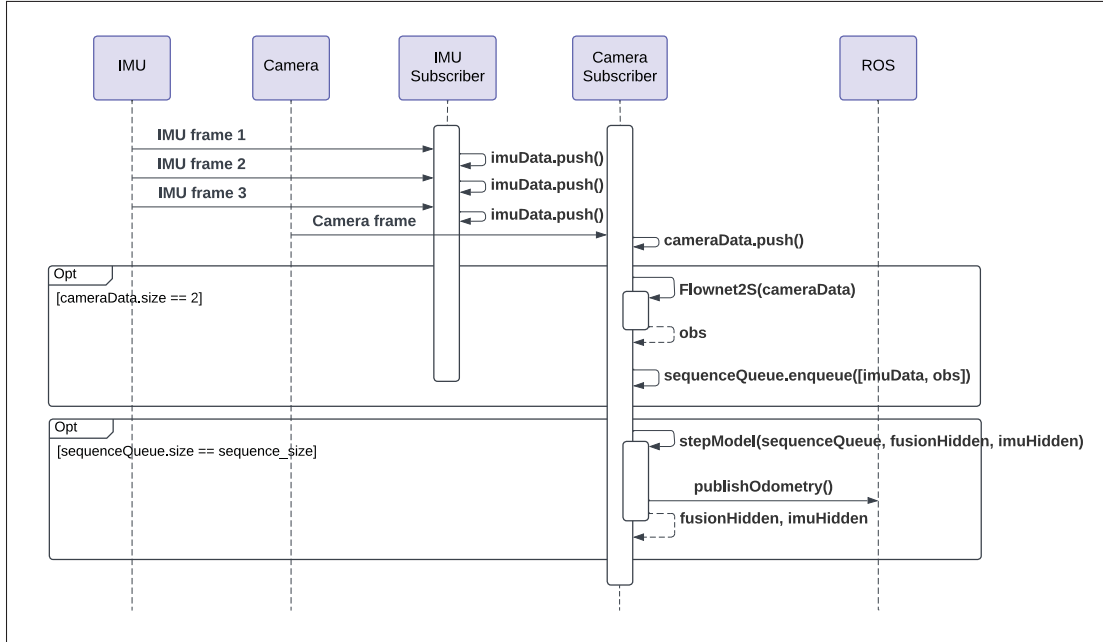


Figure 4.7 Phase 3 sequence diagram

Before obtaining the first result from the model, Flownet2S requires two frames. Also, since the hidden frames of the RNN are either initialized to random values or zeros, the first predictions are not very accurate. To mitigate this problem, a warm-up period of 7 frames is considered. Therefore, the first localization data will be available only after the 8<sup>th</sup> frame. To emphasize the different execution philosophy compared with Phase 2, the model execution is labeled *stepModel*, meaning that the process performs one frame step instead of the whole seven-frame execution.

The benefit of this phase is the reduction of computations during model execution, as the sequence loop of 7 frames is accumulated over time instead of being executed at every camera frame reception. This reduction removes some burden on processing units (CPU/GPU) but does not affect memory usage.

The Flownet overrun impact on the synchronization is the same as for Phase 2, which is considered a failure.

## 4.5 Evaluation

To validate the model in various deployment scenarios (edge, fog, and cloud), different hardware configurations are used, which are detailed in Section 4.5.1 along with a description of the executed tests. Section 4.5.2 discusses the frame rate validity considered for the evaluation, while Sections 4.5.3, 4.5.4, and 4.5.5 advertise the phase-by-phase results to show the improvement brought by each implementation in terms of execution time. To further analyze the results, resource usage is evaluated in Section 4.5.6, along with power consumption and hardware temperature in Section 4.5.7. Finally, Section 4.5.8 concludes the evaluation with a discussion of the impact of the optimization changes on model execution.

### 4.5.1 Methodology

Three hardware configurations were used to compare the performance of the model's execution (described above) for robot localization. These three configurations vary in terms of resources and are used to represent different execution scenarios in a distributed robotics solution. Typically, servers at the fog and cloud levels utilize compute virtualization; however, we consider the suggested hardware as a representative configuration for such virtual machines. These are detailed as follows:

- **Jetson:** A Jetson Orin AGX 32 Gb RAM running over Jetpack 6.0 (based on Ubuntu 22.04 Desktop), representing the robot's computing power at the edge.

- **Laptop:** A laptop with an Intel Core i7-10750H, 32 Gb of RAM, and an NVIDIA GTX 1660 Ti (6 Gb of GPU memory) running over Ubuntu 22.04 Desktop, which is considered as an edge node (not localized on the robot but could be installed as a gateway on a floor) or a small fog node.
- **Desktop:** A desktop computer with an Intel Core i7-12700K, 32 Gb of RAM, and an NVIDIA RTX 3090 (24 Gb of GPU memory) running over Ubuntu 22.04 Desktop, considered a fog or cloud node.

For all configurations, the ROS package running the visual odometry model is executed in a Docker container using CUDA, and the sequence is also executed in a separate Docker container on the same computer. Communication between the two containers is done through the Docker virtual network and is not advertised outside the computer.

The sequence used from the MIT Stata Center dataset is 2012-04-03-07-56-24, which contains 45720 camera frames representing a run of 25 minutes and 27 seconds.

The evaluation examines the execution time for each optimization phase and the resource usage, including CPU, GPU, Random-Access Memory (RAM), power consumption, and temperature. The Basic Hardware Monitor script by Isenko (2023) was integrated into the ROS node to obtain the metrics detailed above. To obtain GPU information on the Jetson, some changes were required using the `jetson_stats` library from Bonghi (2007). The energy consumed during the sequence was monitored using a wattmeter to capture the overall energy consumption throughout the sequence execution. The computer is directly connected to the wattmeter to obtain the energy data, and the acquisition is reset manually when the sequence is initiated. In the following box plots (Figs. 4.8 to 4.9), the red line represents the maximum execution time of 33.3 ms (a frame rate of 30 Hz).

For each optimization phase, the entire dataset sequence was processed independently on each hardware configuration. All tests were conducted under identical conditions, with system load minimized and hardware monitoring scripts running concurrently to ensure consistency in

resource usage measurement. The random seed value used for all the runs is similar to ensure consistency. Therefore, the model accuracy is equivalent in all cases as depicted in Section 4.3.2.

#### 4.5.2 Frame Rate

It is essential to note that failing to meet the 30 Hz threshold results in prediction failure. The model relies on two camera frames and three IMU frames to predict its position before the next camera frame is captured. If a deadline is missed, two options are available. First, the faulty frame could be ignored; however, in that case, it would impact the model, as it expects consistent hidden layer values over time for the RNN. The other option is to always wait for the prediction.

Table 4.2 represents the number of frames executed over the 25-minute sequence based on the second approach. Under ideal circumstances, we would expect 45,719 prediction values (one frame is not used for prediction due to the Flownet model). From these values, it is evident that the second option has an impact. An accumulation of time can be observed, indicating that services requiring localization will also be delayed. The laptop and desktop configurations work optimally for Phase 3, with all frames processed. Still, for the Jetson, only 67% of the frames are executed over the nominal sequence time. The result of this overflow makes it impractical for real-time applications since the prediction of the last frame would be computed approximately 12 minutes later. The implemented improvements have a significant impact. For the Jetson, during the nominal sequence time, only 11.3% of the frames were processed with Phase 1, 15.5% for Phase 2, and 67.3% for Phase 3. The laptop shows similar improvement with only 9.9% of the frames processed for Phase 1, 22.8% for Phase 2, and a successful 100% for Phase 3. The Desktop was already running almost fine with 99.99% of the frames processed for Phase 1, no noticeable improvement for Phase 2, and 100% for Phase 3.

Table 4.2 Number of frames executed over the sequence by configuration and development phase

| Configuration | Phase 1 | Phase 2 | Phase 3 |
|---------------|---------|---------|---------|
| Jetson        | 5173    | 7065    | 30780   |
| Laptop        | 4513    | 10411   | 45719   |
| Desktop       | 45713   | 45713   | 45719   |

### 4.5.3 Phase 1

The results in Fig. 4.8 illustrate the execution time of the three configurations for the first optimization phase. This phase is considered the baseline since the original code was incompatible with ROS.

The box plot diagram shows that the Jetson's average execution time (248 ms) is better than that of the laptop (278.1 ms). The laptop configuration exhibits greater variability, ranging from 6 ms to 629 ms, compared to the Jetson's variability, which ranges from 14 ms to 540 ms. However, the real-time constraint of 30 Hz (or 33.3 ms per frame) is not respected with the laptop and the Jetson. The desktop configuration is usually within tolerances, with an average execution time of 10.7 ms, but some outlier values are taking longer to process (up to 117 ms).

To understand the impact of hardware configurations, comparing their parameters is mandatory. The Jetson Orin AGX features 1792 CUDA cores, compared to 1408 for the laptop's GTX 1660 Ti. Based on Technical City (2024), its clock speed (930 MHz) is less than the laptop's (1530 MHz). The Jetson has a floating-point processing power of 3.333 TFLOPS compared with 5.027 TFLOPS for the computer GPU. When comparing these specifications, the GTX 1660 Ti presents better parameters but performs worse than the Jetson. One possible justification is that the GTX 1660 Ti is a general-purpose GPU, while the Jetson is explicitly designed for AI tasks. The Jetpack operating system, while based on Ubuntu 22.04, is customized for the particular embedded characteristics of the Jetson. This specialization can explain the better results. In comparison, the RTX 3090 desktop is powered by 10,496 CUDA cores running at 1.7 GHz, which significantly surpasses the two other configurations.

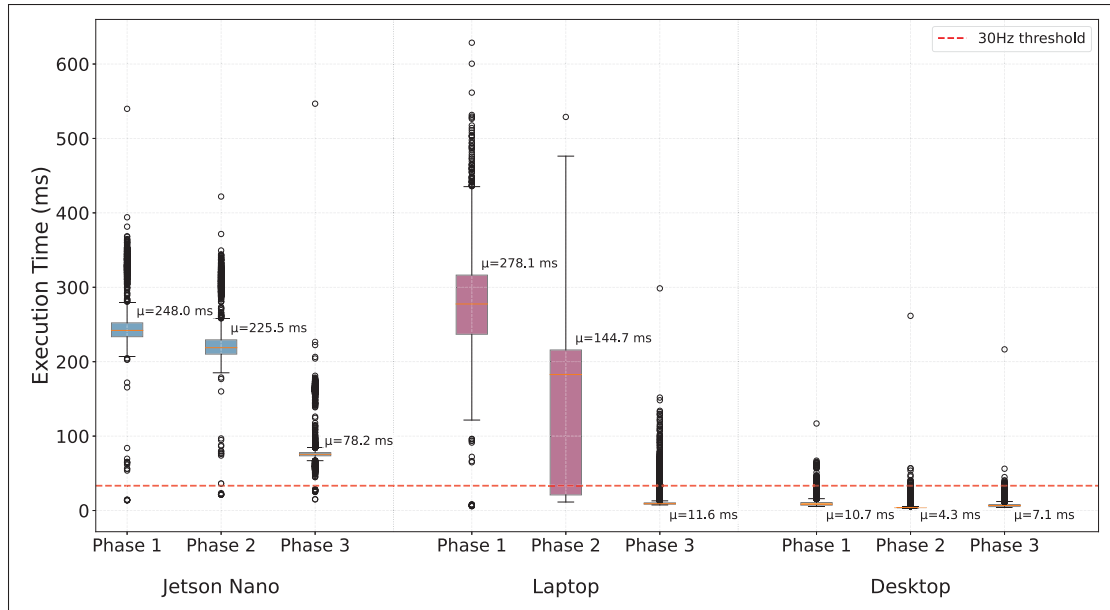


Figure 4.8 Model execution time for each configuration and phase

Another possible reason for the high variability in execution time is the kernel's task scheduler. Since other processes run on the computer, the available CPU and memory can be affected. Moreover, the three configurations run a desktop version of Ubuntu, so the user interface update and other services are workloads the operating system must handle. For the Jetson, the CPU and GPU share the same memory, which can add complexity to the operating system's resource management. This is not the case for traditional Nvidia GPUs, which have dedicated memory.

#### 4.5.4 Phase 2

The execution time for the second optimization phase is also shown in Fig. 4.8. All configurations exhibit a better average execution time, although the improvement is less significant for the Jetson configuration. Effectively, the Jetson achieves an average execution time of 225.5 ms, with peaks of up to 422 ms, while the laptop achieves an average execution time of 144.7 ms. The desktop has improved to 4.3 ms and exhibits less variability. The desktop outliers are less frequent than for Phase 1 but are still problematic (up to 262 ms). The variation in laptop configuration is more significant, with a maximum of 529 ms. This variation can be explained

by the accumulation of parallel processing induced by the frame overflow that may overwhelm the system.

#### 4.5.5 Phase 3

The final optimization phase execution time results are shown in Fig. 4.8. Again, the global performances were improved significantly for the Jetson (78.2 ms with a peak of 547 ms) and laptop (11.6 ms with a peak of 299 ms) configurations. The Jetson's average execution time is enhanced by 147.3 ms, and for the laptop, by 133.1 ms. In both cases, the variability is also narrowed down. Unfortunately, Jetson's performance is still higher than the real-time constraint tolerance, but the laptop is now within the tolerance. However, the outliers can be problematic over time. The desktop average execution is slightly higher than Phase 2, with an average execution time of 7.1 ms, but with less variability and a peak value marginally better at 217 ms.

Table 4.3 Submodels execution times (in ms) per configuration on development phase 3

| Configuration | Flownet | Encoder | Transition | Pose | Total |
|---------------|---------|---------|------------|------|-------|
| Jetson        | 48.2    | 19.4    | 10.6       | 17.4 | 95.6  |
| Laptop        | 10.0    | 0.3     | 1.3        | 16.2 | 27.8  |
| Desktop       | 6.1     | 0.2     | 1.4        | 1.4  | 9.2   |

To understand the distribution of execution times across the entire visual odometry model, a breakdown of execution times for each submodel is presented in Table 4.3, which represents the execution time of each submodel for each configuration. The Flownet and Pose submodels are the most demanding due to the size of their neural networks. From these results, it is evident that running Flownet on the Jetson is not viable for meeting real-time constraints, as it exceeds the available time alone. The laptop and desktop timings are all below the 30 Hz (33 ms) threshold, as well as the sum of the submodel's execution time, which makes them viable configurations.

Suppose it is necessary to perform at least the FlowNet submodel due to the high latency induced by communicating images to other layers (edge, fog, or cloud). In that case, the model must be

modified to reduce the size of the FlowNet network. This reduction will impact the accuracy, necessitating some hyperparameter tuning to achieve a comparable model accuracy.

The Jetson Orin AGX offers multiple power configurations (15 W, 30 W, 50 W, and MAXN). These pre-settings try to limit the hardware's power consumption to the provided value. Although it is not guaranteed that the limits will not be exceeded, the average consumption is below the threshold. The MAXN configuration enables the hardware to utilize as much power as necessary, provided it remains within the safety limits of the hardware. The model was executed using two different Jetson system parameters to determine their impact on execution time. The Fig. 4.9 shows the differences in execution time for the MAXN and 15 W, which are the extremes in terms of configuration.

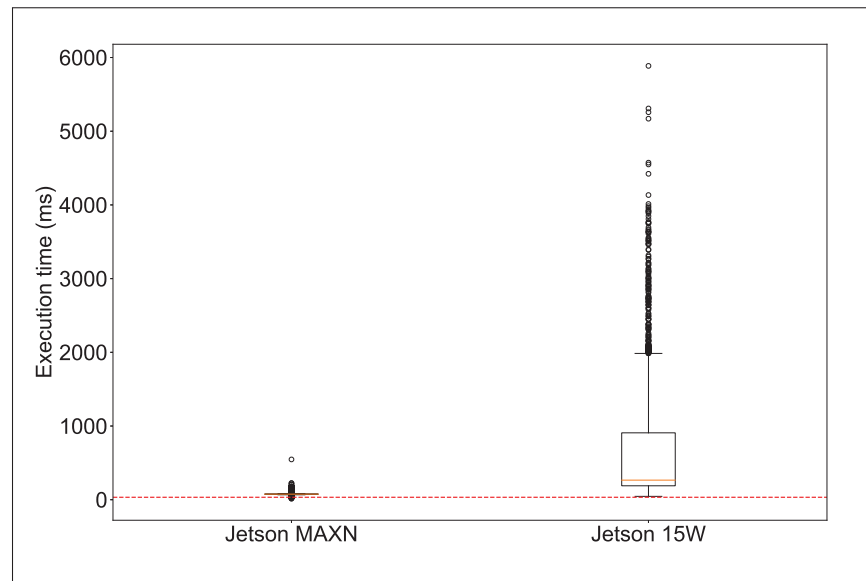


Figure 4.9 Phase 3 execution time for two Jetson power profiles

Power usage has a significant impact on performance. For the MAXN configuration, the average execution time of 78 ms exhibits smaller variability (ranging from 15 ms to 547 ms) compared to the 15 W configuration, which has an average execution time of 674 ms with a variation ranging from 15 ms to 5.5 seconds. This can be explained by the various parameter changes required to reduce power, as detailed in Table 4.4, which implies a slower processing frequency that

directly impacts performance. To achieve real-time execution time for AI models on a Jetson, a compromise must be made regarding energy consumption.

Table 4.4 Jetson power profile configuration

| Profile | CPU Freq.<br>(MHz) | GPU Freq.<br>(MHz) | CPU Cores | GPU Cores | RAM Freq.<br>(MHz) |
|---------|--------------------|--------------------|-----------|-----------|--------------------|
| 15W     | 1113.6             | 420.75             | 4         | 3         | 2133               |
| MAXN    | 2201.6             | 1301               | 12        | 8         | 3200               |

#### 4.5.6 Resource Usage

Another critical factor that influences the execution time is resource availability. Table 4.5 shows the CPU, RAM, and GPU usage metrics over the sequence execution. The comparison of resource usage between configurations is not relevant since the amount of resources varies from one to another, as seen in Section 4.5.1. However, the resource usage for a given configuration from one development phase to another is interesting.

Table 4.5 Resource usage of each configuration by development phase

| Configuration | Phase 1    |            |            | Phase 2    |            |            | Phase 3    |            |            |
|---------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|
|               | CPU<br>(%) | RAM<br>(%) | GPU<br>(%) | CPU<br>(%) | RAM<br>(%) | GPU<br>(%) | CPU<br>(%) | RAM<br>(%) | GPU<br>(%) |
| Jetson        | 14.8       | 69.3       | 68.1       | 21.7       | 69.4       | 69.0       | 20.8       | 71.4       | 71.7       |
| Laptop        | 17.4       | 30.6       | 42.8       | 17.2       | 30.6       | 25.3       | 24.2       | 30.7       | 25.1       |
| Desktop       | 7.7        | 36.6       | 8.0        | 8.7        | 37.6       | 12.0       | 9.4        | 39.2       | 8.9        |

According to Table 4.5, the Jetson's CPU usage increases by 6.9% with the parallelization and slightly decreases by 0.9% for the loop unrolling. The laptop's CPU usage is stable for the first two phases and increases by 6.8% for Phase 3. The desktop displays a regular rise in CPU usage across the various phases. For the Jetson and the laptop, as the number of frames processed increases, the amount of disk access required to write the experimental data to files increases accordingly, which can justify the high variation. The CPU variation for the desktop configuration is almost linear and represents a 1.7% increase, which can be attributed to multiple factors external to the model execution. For the laptop and the Jetson, the increase can be

associated with the rise in frames processed, which requires more processing for data preparation and publication.

RAM usage does not significantly vary from one phase to another. External factors could cause the variation. While all the hardware configurations share the same amount of memory (32 GB), the Jetson uses more memory than the other configurations. The justification stems from the fact that the Jetson memory is shared among the CPU and the GPU, whereas the GPU has its own dedicated memory for laptops and desktops.

The GPU usage on the Jetson presents a low increase of 5%. The behavior on the laptop and the desktop is entirely different. On the laptop, Phase 1 presents GPU usage almost 17.7% higher than the other phases. Despite the absence of parallel processing, the high GPU anomaly may be attributed to the sequential inference, which results in prolonged GPU engagement per frame. In contrast, later phases implement more efficient frame handling, allowing the GPU to return to lower usage states between computations. Additionally, the GTX 1080 GPU of the laptop differs significantly from the RTX 3090 and Jetson Orin AGX GPUs in terms of architecture, available resources, and driver-level resource management. All these factors can contribute to the observed anomaly.

On the desktop, Phase 2 presents a 1.2% increase. The increase can be justified by the parallelization of the Flownet with the remaining part of the model, which adds to the computational burden. Since only one step is performed during Phase 3 instead of the entire seven steps of the sequence, the GPU usage decreases from 12% to 8.9%. This effect is not visible on the laptop or the Jetson, mainly because of the accumulated execution delay, which puts constant pressure on the resources compared to the laptop and desktop, which benefit from the idle time between frames.

#### **4.5.7 Power and Temperature**

Since the visual odometry model is designed for a battery-powered robot, other factors, such as temperature and energy consumption, must also be analyzed. Table 4.6 presents the GPU

temperature and the average power consumed for each configuration and development phase. The Jetson temperature is stable at 47.3°C from Phase 1 to Phase 2 and decreases to 45.9°C in Phase 3. The same temperature stability is observed from Phase 1 to Phase 2 for the laptop at 77.5°C with a slight variation of 0.9°C. The temperature rises to 83.7°C during Phase 3. For the desktop, the Phase 1 temperature is 54.6°C, which rises to 66.1°C for Phase 2 and decreases to 50.9°C for Phase 3.

Except for the laptop configuration in Phase 3 and the desktop configuration in Phase 2, there is no significant variation in temperature. The possible cause of the laptop's issue is a problem with the cooling system, as the main difference in resource usage is between the CPU and the GPU. For the desktop, we see a more significant usage of the GPU, which can explain the variation. Sustained GPU temperatures above 80°C may trigger thermal throttling, which reduces clock speeds and negatively impacts real-time performance. This behavior can explain the outlier execution time values of Phase 3 on the laptop configuration.

Table 4.6 GPU Temperature in °C for each configuration and development phase

| Configuration | Phase 1 | Phase 2 | Phase 3 |
|---------------|---------|---------|---------|
| Jetson        | 47.3    | 47.3    | 45.9    |
| Laptop        | 77.5    | 76.6    | 83.7    |
| Desktop       | 54.6    | 66.1    | 50.9    |

The GPU power usage is shown in Table 4.7. The **Jetson** consumes an average of 16.5 W for Phase 1 and is relatively stable at 20.9 W for Phase 2 and 20.7 W for Phase 3. The **laptop's** power usage gradually increases from 28.2 W for Phase 1 to 38.5 W for Phase 2 and 47 W for Phase 3. The power consumption pattern for the **desktop** is different, with a power usage of 134.2 W in Phase 1, a significant increase to 229.2 W in Phase 2, and a lower value of 126.5 W in Phase 3.

The Jetson is optimized for low-power computation, as an event in the MAXN power mode requires a maximum of 21 W. The temperature and power usage of the laptop and desktop can be correlated. A potential justification for this is that when the GPU is heating, the power

necessary for the cooling system seems to consume a significant amount of power to reduce the temperature.

Table 4.7 GPU Power in Watts for each configuration and development phase

| Configuration | Phase 1 | Phase 2 | Phase 3 |
|---------------|---------|---------|---------|
| Jetson        | 16.5    | 20.9    | 20.7    |
| Laptop        | 28.2    | 38.5    | 47.0    |
| Desktop       | 134.4   | 229.2   | 126.5   |

The GPU power is not necessarily the best way to identify the model's real impact. Table 4.8 details the GPU power consumption while adding information about the energy consumed over the sequence execution. When considering the energy, it is noticeable that the optimization has a negative impact. The desktop consumes 39 Wh less with all the optimizations, while the laptop's consumed energy increases by 7 Wh, and the Jetson remains at the same energy level.

The decrease in energy consumption from the desktop can be justified by the reduction in computation due to the Phase 3 loop unrolling. The energy values are not representative of the laptop and Jetson configurations, as noted earlier, since the monitoring was conducted over a 25-minute sequence execution. Unfortunately, most configurations and phases did not execute the process over all the frames, which increasingly impacts actual energy consumption. Effectively, when examining the desktop configuration that executed the model over all frames, regardless of the development phase, we see that, contrary to expectations, the optimization has a positive impact on energy consumption. It is worth noting that the total power consumed by the Jetson and the laptop for the first two phases is not representative, as it only covers a portion of the total frame to process.

#### 4.5.8 Discussions

The Table 4.9 summarizes the impact on the model execution time through the various optimization phases.

Table 4.8 Total power for each configuration and development phase

| Configuration | Phase 1 |         |             | Phase 2 |         |             | Phase 3 |         |             |
|---------------|---------|---------|-------------|---------|---------|-------------|---------|---------|-------------|
|               | min (W) | max (W) | total (kWh) | min (W) | max (W) | total (kWh) | min (W) | max (W) | total (kWh) |
| Laptop        | 54.3    | 107.2   | 0.029       | 57.9    | 103.7   | 0.027       | 28.7    | 115.3   | 0.036       |
| Desktop       | 266.8   | 313.5   | 0.124       | 200.7   | 239.2   | 0.092       | 199.9   | 230.1   | 0.088       |
| Jetson MAXN   | 19.8    | 33.6    | 0.011       | 20.3    | 31.5    | 0.010       | 18.4    | 34.1    | 0.011       |
| Jetson 15W    | -       | -       | -           | -       | -       | -           | 14.2    | 19.5    | 0.007       |

Table 4.9 Execution Time Improvement Across Optimization Phases

| Configuration | Phase 1 $\rightarrow$ 2 | Phase 2 $\rightarrow$ 3 | Phase 1 $\rightarrow$ 3 |
|---------------|-------------------------|-------------------------|-------------------------|
| Jetson        | 9.07%                   | 65.32%                  | 68.47%                  |
| Laptop        | 47.97%                  | 91.98%                  | 95.83%                  |
| Desktop       | 59.81%                  | -65.12%                 | 33.64%                  |

While the improvement is not sufficient for the Jetson, it is evident that with an improvement of 68.47%, the changes are still insufficient to meet the real-time constraints. Still, it significantly improves it, so that when combined with other model optimization methods, such as pruning and quantization, it would be possible to meet the goal. The 95.83% improvement for the laptop makes it compatible with the real-time constraints. While the desktop was already running within the constraints, the 33.64% improvement leaves more room for other processes to run on the same computer.

Due to the size of the neural network, DL-based visual odometry models for predicting robot localization are highly resource-demanding. The proposed algorithm modifications significantly improve execution time, but based on the obtained results, real-time execution on a Jetson cannot be achieved. While slightly increasing the energy consumption on the desktop configuration, the main impact of the optimization is the reduction of outliers that result in overruns.

An interesting aspect arises from studying the energy consumption of Jetson. Even if the resource usage increases, the impact on power usage is negligible. This emphasizes that the Jetson architecture is optimized for systems running on batteries, even when running extensive

AI models. We would need to reduce the camera frame rate to run the model effectively on this configuration, thereby providing more time for model execution. Lowering the robot's speed would allow this rate to decrease.

Compared to other optimization methods, the proposed solution does not modify the model structure, ensuring the same accuracy as long as the execution time meets the real-time frame rate. Adding more optimization would potentially decrease model accuracy while improving execution time.

Since the desktop is the only configuration capable of running all the frames, it is the best example for comparing the impact of the optimization phases. The desktop's average execution time is approximately 2.9 ms longer than phase 2. However, the less frequent and shorter outliers show improvement. It can be seen that the optimization has a low impact except for the reduction of outliers. One justification for this is the availability of better resources than the Jetson and the laptop. Even at Phase 1, the average model execution time was within the limits of real-time execution. To have a better impact, the optimization needs to move the model execution closer to or even below the limit, as it frees resources induced by heavy parallelization (more than two frames executed at once).

Although our evaluation focuses on single-machine deployments, network latency and communication overhead are critical factors in distributed robotics. The network communication latency is greatly impacted by the data size and the data rate, as concluded in a previous work (Melançon *et al.* (2023)). The image processing should preferably be executed on the robot itself to reduce the size of the data exposed to the other layers. While our approach preserves model accuracy by avoiding structural changes, this inherently limits the potential speed-up in inference. Model compression techniques, such as pruning and quantization, could further reduce execution time but may require more fine-tuning training to maintain similar accuracy. The combination of the proposed execution pipeline and deployment strategy with model optimization is part of future work, along with parallelization in a distributed manner across the edge-fog-cloud continuum.

## Conclusion

This study defined real-time execution requirements for a DL visual odometry model and proposed algorithmic optimizations to meet these constraints without altering the model structure. The evaluation across Jetson, laptop, and desktop configurations showed that while optimizations significantly reduced execution time (up to 95% improvement on laptop configuration). Jetson hardware still cannot achieve the 30 Hz real-time threshold, making full on-board processing impractical for mobile robots. These findings highlight trade-offs among performance, energy consumption, and hardware feasibility and suggest that distributed processing across edge, fog, and cloud layers may be necessary. Future work will address communication latency, connectivity resilience, and fallback strategies to ensure robust real-time localization in autonomous robotic systems.

## CHAPTER 5

### **BLAZEFLOW: A MULTI-LAYER COMMUNICATION MIDDLEWARE FOR REAL-TIME DISTRIBUTED IOT APPLICATIONS**

Cédric Melançon<sup>1</sup> , Guillaume Simard<sup>2</sup> , Maarouf Saad<sup>1</sup> , Kuljeet Kaur<sup>1</sup> , Julien Gascon-Samson<sup>2</sup>

<sup>1</sup> Department of Electrical Engineering, École de Technologie Supérieure, Montréal, Québec, Canada

<sup>2</sup> Department of Software and IT Engineering, École de Technologie Supérieure, Montréal, Québec, Canada

Paper presented to the « 1st International Workshop on Middleware for the Computing Continuum », December 11–15, 2023, Bologna, Italy

#### **5.1 Abstract**

The Internet of Things landscape has grown steadily over the last decade, fueling digital transformation. Data is now regarded as a precious asset. As a result, many companies have shifted their operations to the cloud to avoid maintaining massive infrastructure to hold and process all the data. However, in many settings, sending all the raw data to the cloud for processing and storage is not possible due to the unreliability of wide-area connectivity, coupled with the high costs of data transfer, storage, and processing. Also, some applications exhibit stringent response-time requirements. Edge computing can mitigate some of these issues, but it also has some drawbacks. This chapter presents BlazeFlow, our vision of a multi-layer data flow solution that can transport data to and from any layer of the cloud-to-device continuum based on publish-subscribe abstractions. BlazeFlow handles data flows between different services deployed onto the same node, between different devices of the same layer, and between different layers (edge, fog, and cloud). This is realized by a cross-layer common bridging solution that can automatically aggregate various protocols that are appropriate for each layer/device (e.g., ROS2, MQTT, Redis, and Kafka). BlazeFlow is designed to support bandwidth-intensive and time-critical services deployed at the various layers of the system. We evaluate a preliminary

design of BlazeFlow over an autonomous robotics use case, and we show that it can sustain multi-layer data flows at a high frequency and with a low latency.

## 5.2 Introduction

Systems with real-time constraints, high-volume data, and high-frequency sensors are not good candidates for cloud computing because of the communication latency induced by the distance between the system and the cloud data centers, as well as the high cost of data storage and transfer. Rather, to enable rich and time-sensitive services, the data needs to be processed at different levels of the device-to-cloud continuum. For instance, the edge and fog layers allow the data to be processed closer to where it is produced and consumed, and can also enable time-critical services, in combination with the cloud, which can be used for high-level analytics, monitoring, and persistence. This drives the need for a flexible multi-layer communication engine that can sustain the needs of various Internet of Things (IoT) applications.

Autonomous robotics solutions are an excellent example of a system that requires fast data processing and that produces a massive volume of data with relatively high frequency. This scenario (more details in Section 5.3) raises the need for a low-latency mechanism to transport the data from the source to various services deployed at every layer. Also, having a distributed solution to execute the control of the robot requires the data solution to be flexible and to provide easy access to the data within each node dynamically.

Several technologies exist to share the data between nodes at the same or different layers. Further, multiple services can also be deployed on the same node. Flexibility is essential, as some services can be dynamically migrated between layers to adapt to the current conditions and constraints. For instance, a critical service that requires low latency should ideally be deployed within the device but may need to be migrated to the edge to reduce the strain, given low battery conditions (Tank & Gandhi (2023)). Also, a service that suddenly needs to process a higher message throughput might need to be migrated from the edge to the fog layer to benefit from additional resources (Nezami *et al.* (2020)). As an example, one can consider an obstacle

avoidance service that might need to increase the camera rate when detecting a potential obstacle to avoid.

Interoperability is one of the significant challenges of the IoT landscape. There exists a plethora of protocols, but there is no unified way to translate from one protocol to another. Also, many protocols are inefficient for systems requiring high-throughput and real-time constraints. This work aims to provide a generic way to bridge any protocol to a common one that answers to the constraints of a distributed real-time solution. Most existing middleware systems focus on sharing data from one layer to another and do not consider sharing locally within a node to reach various services deployed onto the same device. This work presents a global approach that can handle internal and layer-to-layer data sharing.

This chapter presents a preliminary design of BlazeFlow, a middleware platform designed to handle the inter and intra-layer communication needs of distributed IoT applications based on the device-to-cloud continuum. This chapter provides the following contributions:

- Handling all end-to-end communications from the edge to the cloud by transporting the data between layers and between nodes within the same layer using a publish-subscribe paradigm.
- The ability to dispatch messages to and from different services deployed onto the same local node.
- A universal bridging solution that can translate messages into a layer-specific format to a common format for inter-layer messaging.
- A performance evaluation on an autonomous robotics solution use case, demonstrating that BlazeFlow can sustain the transfer of different types of messages (e.g., sensor readings, images, etc.) at a high frequency and with low latency.

### 5.3 Motivating Use Case

Autonomous robots typically have substantial real-time requirements; *i.e.*, they have to sustain a minimal rate of around 30 Hz for the control process (Chen *et al.* (2018)), which makes it impossible to deploy the whole processing on the cloud because of the high latency induced by

the communication. Most existing robots host their computing capability locally, which implies more power consumption, higher weight, and increases the cost of the robot (mainly because of the miniaturization of the components). To avoid executing all of the computations on the robot, an option is to offload some of the decision-making tasks from the robot to a combination of edge, fog, and cloud resources. This offloading should consider goals and constraints such as time requirements and criticality. The robot's autonomy comes from its ability to sense and process its environment; therefore, it is equipped with multiple sensors with various rates and payload volumes.

Table 5.1 Data definition of the robot

| Sensor       | Rate    | Size (bytes) |
|--------------|---------|--------------|
| LIDAR        | 20 Hz   | 4265         |
| IMU          | 100 Hz  | 372          |
| RGB Camera   | 29.4 Hz | 307299       |
| Depth Camera | 29.4 Hz | 614494       |

Table 5.1 describes the data definition of the robot that will be used to validate the solution. The payload size is computed using Protocol Buffer (Protobuf) (Google LLC (2023)) encoding. From that table, we can deduce that around 26 Mb/s are sent only for these sensors. Depending on the use of the robotics solution, additional cameras and sensors can be required to enable new functionalities. For example, we can add a manipulator with a camera to the robot, which will generate new data from the camera and potentially other sensors, such as current sensing from each joint of the manipulator. All these sensors increase the data volume. The communication middleware's design must sustain the rate of its components for proper operation.

In an IoT scenario, the robot is the device. While some services can be deployed in the cloud, the latter can only be used by services that are not critical and do not require real-time constraints. For instance, considering the current scenario, the cloud could manage the solution (e.g., dispatching updates, adding or removing robots, and generating reports). Another potential use would be to offload the training of Artificial Intelligence (AI) models to avoid requiring edge nodes with significant GPU capabilities and memory – the cloud also provides the ability

to scale up and down quickly. However, the inference must still be done at the edge to ensure real-time performance.

Globally, the autonomous robot use case raises the following challenges that have to be considered in the design of BlazeFlow: (1) *Real-time execution* – Minimizing the latency of the data processed for quick decision-making; (2) *Handling large volumes of data* – sent and processed at different layers; (3) *Resilience* – Making sure that the data is accessible locally if connectivity to the next layer is lost and the capability to offload data without impacting the performance; and (4) *Data privacy* – Making sure that the data cannot be accessed outside the solution except for the information shared between layers.

## 5.4 System Design

### 5.4.1 Design Goals

The current preliminary design of BlazeFlow focuses on the real-time execution and handling of large volumes of data. The specific technology choices are made to address the following design goals, which are derived from the challenges outlined in Section 5.3.

**(1) Real-time and High-Throughput:** Several use cases require strict response times to meet real-time requirements, e.g., the autonomous robot use case. The framework has to be designed to support real-time operations, even if these applications exhibit high throughput. The solution provides communication capability between services within a network layer or a node.

**(2) Interoperability:** A wide variety of IoT sensors exist, each using its message format and communication protocol. The proposed solution is designed to be interoperable by supporting the most popular protocols with minimal implementation. The platform must have a way to centralize and standardize all of this data to make it available to all applications, regardless of the data source.

**(3) Compatibility:** The solution should be easy to integrate from any programming language since the services can be implemented with any technology. Furthermore, the solution should also be cloud-agnostic, given that different companies use different cloud providers and that some data processing and storage will be deployed at the edge. As such, we avoid relying on cloud-based PaaS services; instead, we deploy our middleware and services on virtual machines or containers, which are provider-agnostic (i.e., *Infrastructure-as-a-Service* – IaaS). However, it would be easy to integrate already existing cloud services into the solution to take advantage of the data from existing applications.

**(4) Privacy and Resiliency:** The data shared by services within a node should not be accessible from outside the node. All data shared between nodes or layers should only be readable within the solution with proper access.

## 5.4.2 Architecture

The architecture of the BlazeFlow multi-layer data flow is described in Figure 5.1. The main idea is that a protocol bridge is implemented to translate from any protocol to the internal protocol selected for the layer.

### 5.4.2.1 Inter-Layer Communications

We chose DDS for all inter-layer communications as it provides high performance, security, reliability, and robustness. Further, the ROS2 robotics framework uses it as its primary protocol and has proven good performance for multiple robotics projects. DDS provides various security mechanisms to ensure that data is not accessed or altered from outside without impacting the protocol's performance (Object Management Group (2023)). The privacy objective is met with DDS security components since the intra-layer communication technologies are not exposed outside its layer. DDS uses encoded binary content, which reduces the size of the messages and the bandwidth. The data coming from the inter-layer communication is pushed either to the intra-layer or intra-node protocol directly with a bridge provided by the framework.

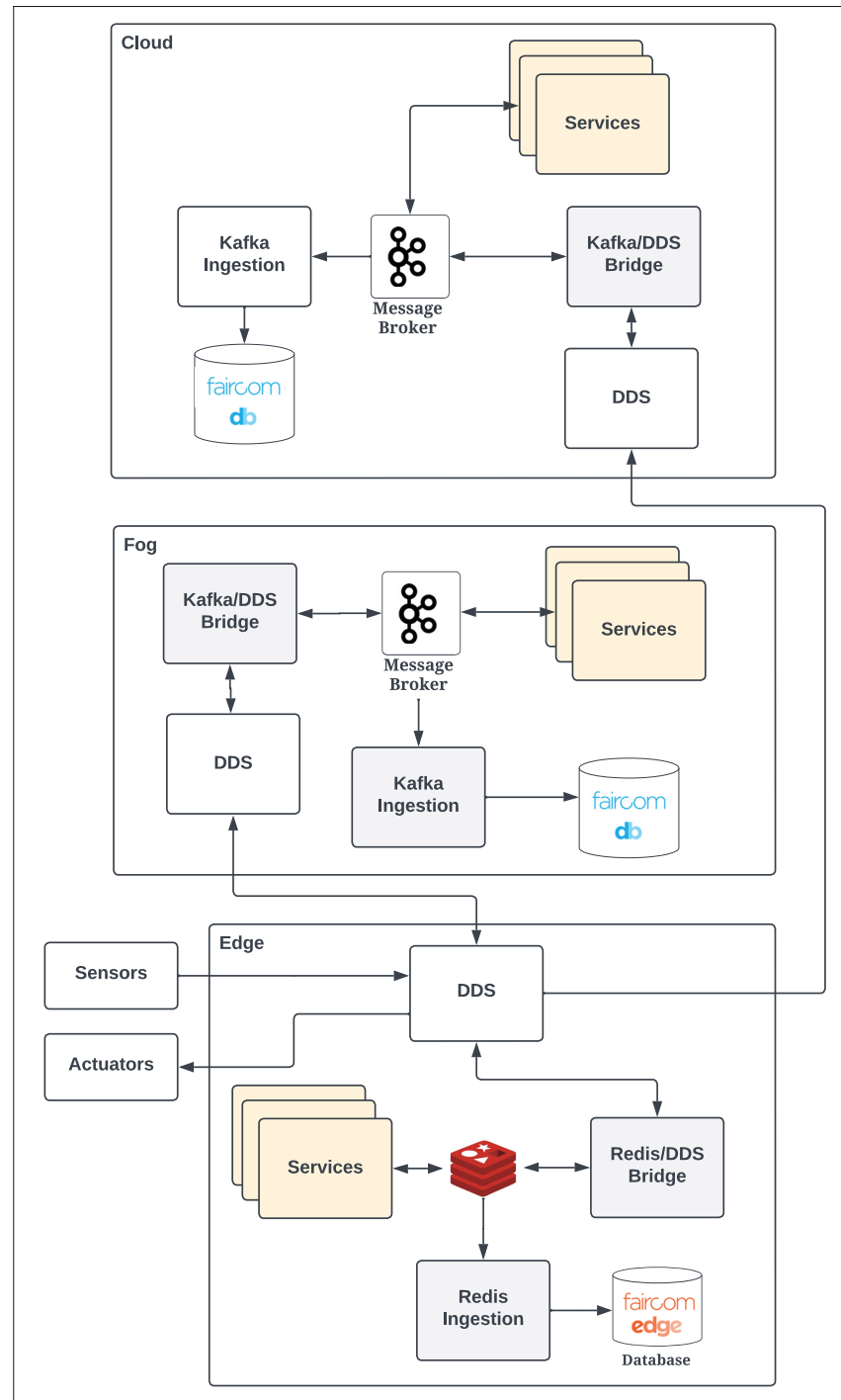


Figure 5.1 Architecture of the proposed solution

#### **5.4.2.2 Intra-Layer Communications**

BlazeFlow also handles communications between different nodes at the same layer. The choice of intra-layer protocol depends, as the resources available in the different layers are not the same (for instance, a node at the edge layer could be a Raspberry Pi, whereas a node at the fog layer could be a more powerful dedicated server). For the edge layer, we also choose Data Distribution Service (DDS) due to its flexibility regarding node topology compared to Redis and MQTT, and it provides a better security mechanism. We decided to use Kafka as the intra-layer protocol for the fog and cloud layers due to its performance, robustness, and ease of use with multiple existing frameworks (such as Apache Spark for AI).

#### **5.4.2.3 Intra-Node Communications**

Intra-Node communications provide a way to interconnect various services deployed onto the same node at the edge layer. MQTT and Redis were considered for their performance, lightweight communication, and the availability of the client libraries in multiple languages. However, we realized that the MQTT performance was insufficient for large messages such as images. Given that the robot produces streams of images at a high frequency, we chose Redis as the intra-node protocol for the edge layer, which provides better performance for sending images (more details in Section 5.5).

#### **5.4.2.4 General Considerations**

Ideally, a single protocol would be used to unify all communications; however, no protocol can satisfy all of the constraints described in Section 5.4.1. For instance, the DDS protocol covers most of them but lacks flexibility regarding client library languages (it only supports C++ and Python) and is not as scalable as other solutions. Kafka covers these aspects but has a significant overhead, which can impact the deployment on edge nodes due to more limited resources. Redis and MQTT are more suitable for edge intra-node communications, but unlike Kafka, the solution's scalability is not as robust, along with the load balancing.

We initially considered JSON for message serialization, but we discovered that it generated too much overhead, which degraded the performance. To reduce bandwidth usage, all data is now encoded using Protobuf. As a result, the payload size is expected to be reduced by around 70% (compared to JSON) according to (Popic *et al.* (2016)) while adding very low computation time. The Protobuf encoding is applied for the fog intra-layer and edge intra-node communications.

#### **5.4.2.5 Local Storage**

By default, one service at each layer is responsible for ingesting the data into the internal database. The database helps with connectivity loss and improves the resilience of BlazeFlow, which is one of the objectives of the solution. A synchronization service can ensure that all the data gathered during the outage can be forwarded to the layers interested in the data. For these reasons, *FairCom Edge* (for edge nodes) and *FairComDB* (for fog and cloud layers) were selected. They both provide a high-performance NoSQL database with a synchronization capability, which will reduce the pressure on the DDS protocol to recover missing data during connectivity loss. The implementation of local storage will be subject of future work.

### **5.5 Evaluation**

#### **5.5.1 Implementation and Deployment**

We implemented a subset of the features of the proposed BlazeFlow design (described in Section 5.4). To accelerate the implementation, the DDS protocol between all layers was replaced by ROS2, which is based on DDS. The resulting solution is shown in Figure 5.2, for the edge and fog layers. Given that the project is still under development, we did not implement the cloud layer (the architecture follows a similar pattern as the fog layer). The *Edge* node is a Jetson Orin AGX (12-core Arm with 64GB of RAM). The *Publisher* and *Kafka Server* nodes are 2-core Pentium G3220T with 16GB of RAM, and the *Subscriber* is a 12-core Intel i7-10750H with 32GB of RAM.

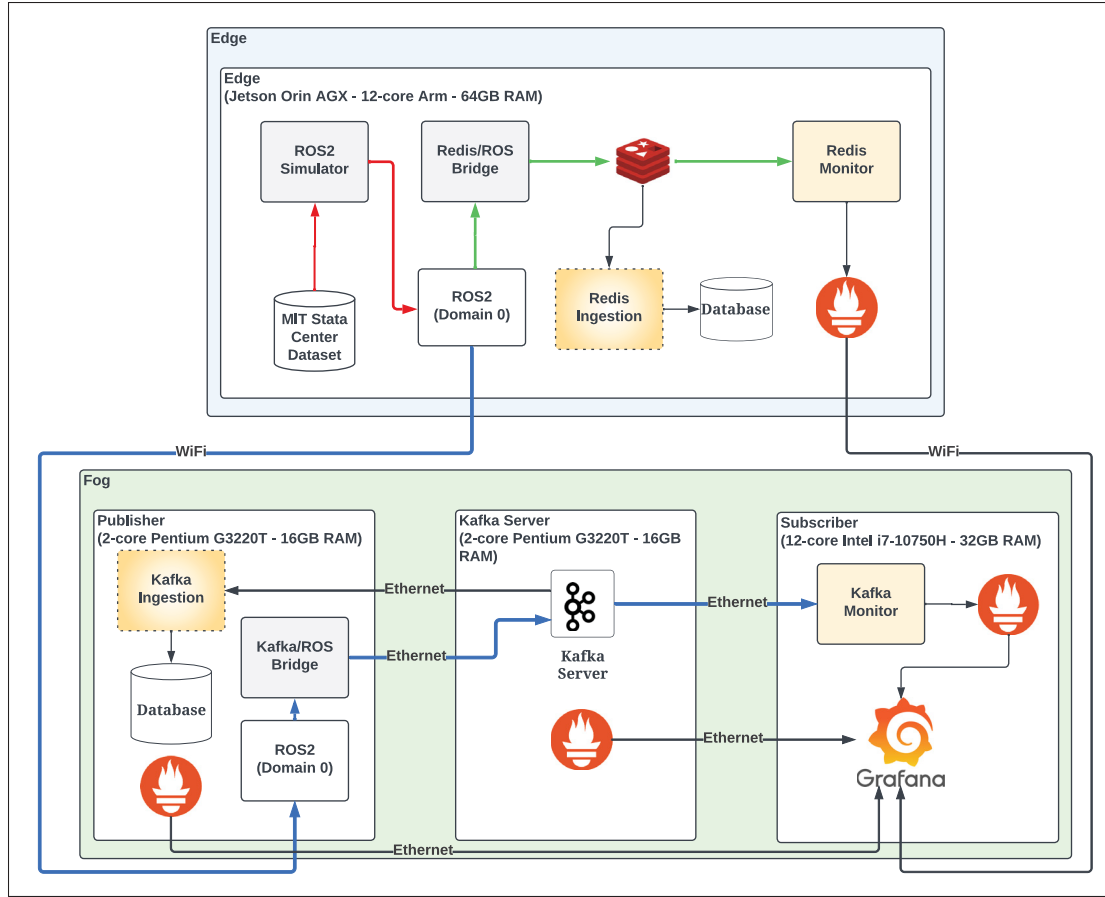


Figure 5.2 Used solution for the evaluation

We use the MIT Stata Center dataset (Fallon *et al.* (2013)), which provides ROS bag files of multiple scenarios in which a robot navigates inside a building. This is very similar to the use case defined in Section 5.3<sup>1</sup>. The ROS2 bag player feature is used to replay the MIT Stata Center dataset scenario as if a real robot were running.

Each layer comprises a monitor service that gathers the metrics shown in this section. Metrics are pushed in a Prometheus server, and a dashboard implemented in Grafana displays the various graphics. The monitor service is also an example of how a new service can be implemented to interact with the protocol bridge. Our implementation and evaluation use the following protocols/middleware platforms, as described in Section 5.4.2: the open-source version of Redis,

<sup>1</sup> We are currently working on designing the physical robot in itself (future work). The robot in the dataset has sensors and data rates similar to the robot we are building.

the Eclipse Mosquitto broker for MQTT, and the open-source version of Kafka. BlazeFlow interconnects these systems.

5.5.2 Experimental Results

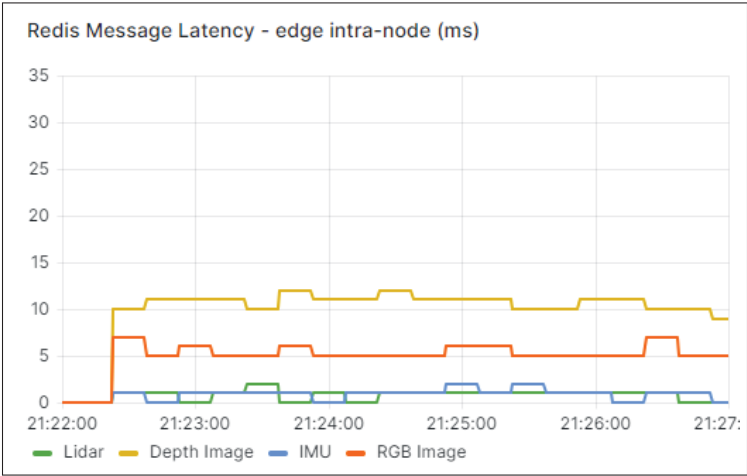


Figure 5.3 Redis message latency

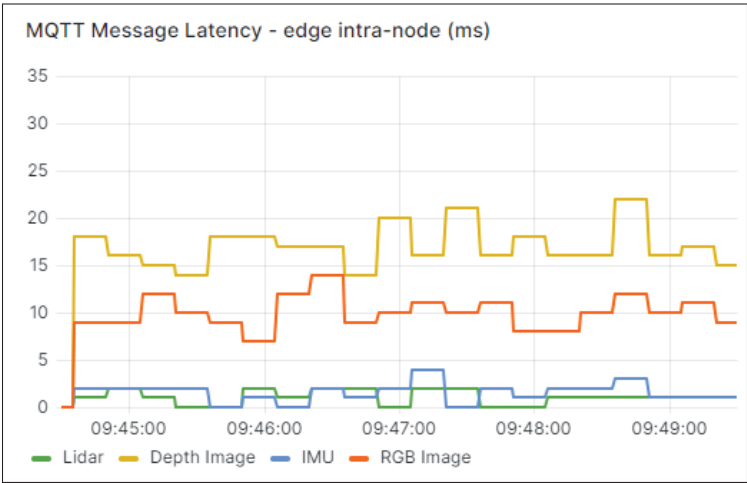


Figure 5.4 MQTT message latency

5.5.2.1 Methodology

The process for the evaluation is as follows. When the ROS2 Simulator is started, it plays the MIT Strata Center dataset bagfile at the same speed as recorded (Table 5.1). The messages are

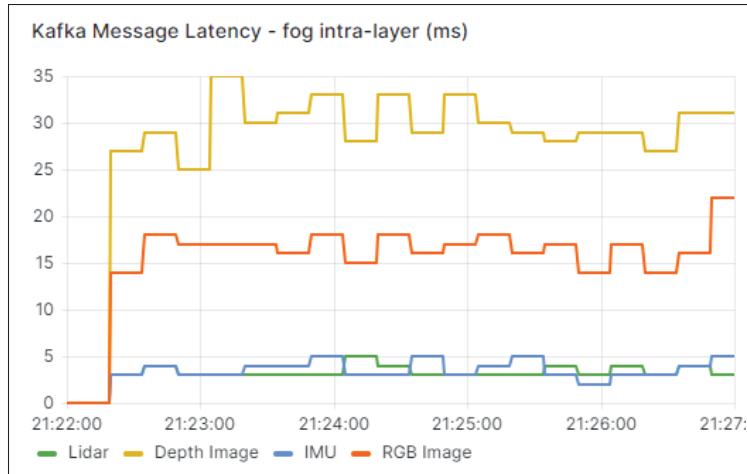


Figure 5.5 Kafka message latency

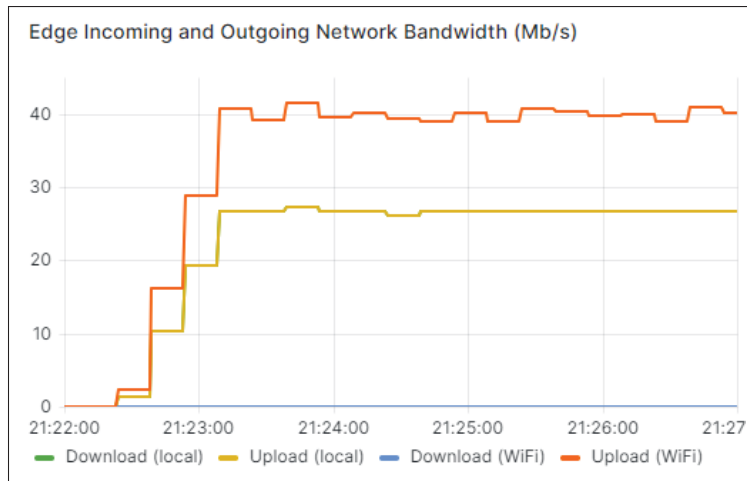


Figure 5.6 Edge incoming and outgoing network bandwidth

sent to ROS, which are consumed by the ROS2 Bridge service on the *Edge* node. The bridge then translates the message to Protobuf, encodes it, and publishes it using the edge intra-node protocol (Redis or MQTT). The Monitoring service consumes the message and produces the metrics in Prometheus. For the fog layer, the *Publisher* node receives the message from DDS (ROS2) in the Bridge service. The message is then translated to Protobuf, encoded, and published to the *Kafka Server* node. On the *Subscriber* node (also at the fog layer), the Monitor service

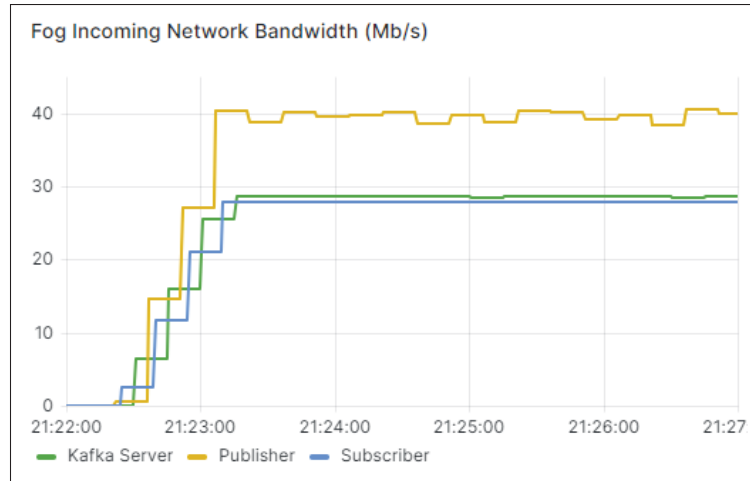


Figure 5.7 Fog incoming network bandwidth

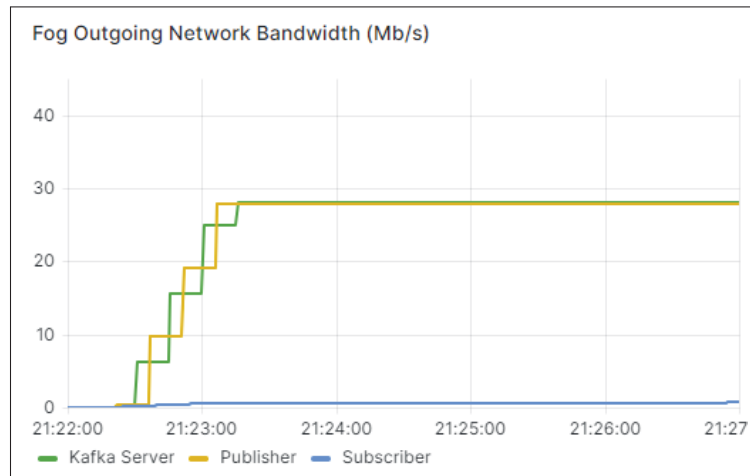


Figure 5.8 Fog outgoing network bandwidth

subscribes to the message topics and receives all of them. When a message is received, the metrics are computed and produced in the Prometheus server. Another computer is connected to the Grafana server to display the metrics in a dashboard in real-time. The results presented in this section are extracted from Grafana. The X-axis represents the time interval corresponding to when the end-to-end experiment was run, and the first minutes of the experiment depicts the ramp-up phase.

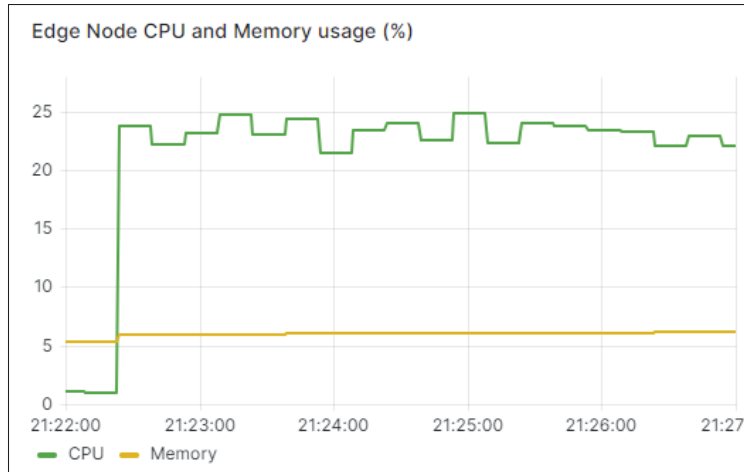


Figure 5.9 Edge CPU/Memory metrics

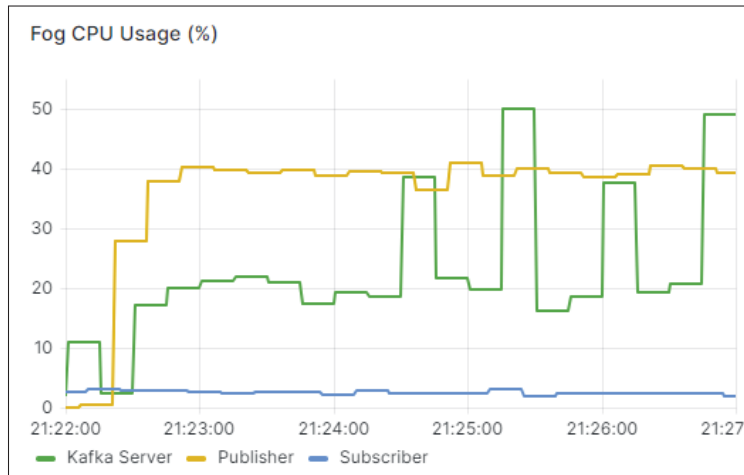


Figure 5.10 Fog CPU usage

### 5.5.2.2 Latency

Figures 5.3 to 5.5 show the latency results for the Redis, MQTT, and Kafka communication protocols/platforms. As mentioned in Section 5.4.2, Redis and MQTT were used for intra-node communications only (no network usage), while Kafka was used for intra-layer fog communications. We can observe that MQTT message latency is 50% higher compared to Redis (hence, the reason why we chose Redis in our system design). For the fog layer (Kafka), the latency is significantly higher than the edge layer (up to three times higher). The ROS message

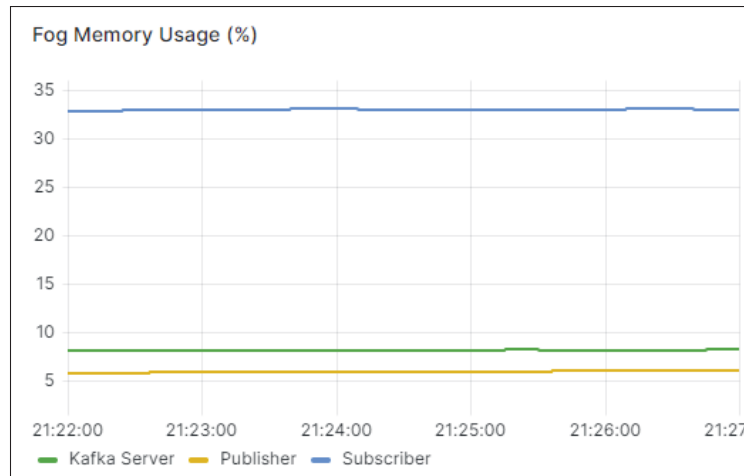


Figure 5.11 Fog memory usage

is triggered from the *Edge* node and is received by the *Publisher* node. The Kafka bridge then publishes the message to the *Kafka Server* node. The Kafka monitoring service receives the message in the *Subscriber* node. All these jumps from one node to another have a cost, which explains the higher latency.

As we can observe, the size of the messages influences the latency. For instance, the *Depth Image sensor* message size is about two times the size of the *RGB Image sensor* and shows a difference in latency time in the same range of difference. However, studying the *IMU* and *LiDAR* messages makes it possible to realize that the size difference (a factor of 10 times) has no impact. We can deduce that the latency can be considered constant up to a particular message size, which probably comes directly from the implementation of the bridge solution. As a limitation of using the MIT dataset, we do not have the inter-layer latency results; we can obtain them once the physical robot is built.

### 5.5.2.3 Bandwidth Utilization

Figure 5.6 shows the network bandwidth utilization at the edge, and Figures 5.7 and 5.8 for the fog layers. At the edge layer (first graph), we notice that the incoming and outgoing rates at the local network interface (intra-node) are balanced, which means that everything pushed through

the local interface is consumed (the curves are superimposed). The data going through the local interface is generated by the bridge into Redis. The wireless interface incoming rate is always 0, as the edge layer does not consume any data from outside. The outgoing bandwidth is higher for the following reasons: (1) the metrics data published by the Prometheus server go through the wireless interface as they are uploaded to the fog, and (2) the encoding technique DDS uses differs from the Protobuf solution used by the bridge but is proportional.

At the fog layer (second and third graphs), the outgoing bandwidth is equivalent for the *Publisher* and the *Kafka Server* nodes since they both relay the same information. The data uploaded by the *Subscriber* node comes mainly from the Grafana server since a browser from an external computer from the evaluation infrastructure displays the metrics in real-time. The incoming bandwidth for the *Kafka Server* and the *Subscriber* nodes are equivalent also because they share the same data. The *Publisher* node incoming bandwidth is equivalent to the edge node outgoing bandwidth of the wireless interface, which is expected since they share the same ROS messages.

#### 5.5.2.4 CPU and Memory Utilization

Figure 5.9 plots the CPU and Memory utilization for the nodes at the edge, and Figures 5.10 and 5.11 for the fog layers. We can observe that CPU usage at the edge layer is acceptable, with about 25% of CPU usage shared between the bridge and the ROS (DDS) usage. In terms of memory usage, the utilization is low (with a variation between 2-3% induced by the framework). At the fog layer, the three nodes display very different usage CPU and memory patterns. The different roles of each node can explain this. The *Publisher* and *Kafka Server* nodes have a higher CPU usage, given that the *Publisher* processes all the data from ROS and forwards it to Kafka, and that the *Kafka Server* node runs the Kafka middleware server and Zookeeper, which process all messages from the solution. We can observe that a node running a client service has little impact on the CPU usage, as can be seen on the *Subscriber* node. The three nodes (*Publisher*, *Kafka Server*, and *Subscriber*) share the same constant memory usage with or without running the solution.

Overall, our preliminary results show that BlazeFlow has the potential to handle IoT applications that span several layers, with a high data volume and strict constraints (e.g., autonomous robotics), with good performance.

## 5.6 Related Work

Migration of processes from the cloud to the edge and fog becomes relevant when dealing with high-volume data and real-time applications. Tianfield (2019) explains the concept of edge-cloud computing by highlighting the different constraints of each layer and proposes a framework to automate the service provisioning. The authors in Poojara *et al.* (2021) introduce a serverless approach for data pipelining to enable flexibility for the distribution of the services inside the whole infrastructure. In Sanchez-Gallegos *et al.* (2022), a meshing approach is proposed to improve the robustness of the solution and reduce the latency.

Distributed robotics solutions are a popular subject nowadays, as there is a colossal volume of data to be processed by computing-demanding services. Cloud robotics is discussed in Mello *et al.* (2023), where the authors show how cloud computing can enable some functionalities and scale the robotics solution. The edge robotics concept is discussed in Seisa *et al.* (2022), and some existing edge platforms are described to help implement such a solution. Fog robotics is described in Casamayor Pujol & Dustdar (2021) with all its challenges, namely, privacy, security, and robustness. Authors in Liu *et al.* (2022a) propose a framework to deal with very demanding services flexibly and efficiently. The ROS framework usage is a common denominator for all the robotics solutions.

Multiple protocols exist for transporting data from one service to another. The authors of Kang *et al.* (2020) compare the performance of multiple middlewares, showing that the Data Distribution Service (DDS) protocol has a better overall latency and throughput for high-frequency dataflow and high-volume traffic. Finally, Hejazi *et al.* (2018) presents a survey of platforms for massive IoT by describing the requirements for such use cases and the protocol involved in each solution.

In contrast to the aforementioned works, this chapter focuses on how the data is transported throughout the solution to meet real-time performance and enable dynamic workload distribution amongst the various layers of the cloud-to-device continuum.

## **5.7 Conclusion**

Prompt and efficient processing of high volumes of data is essential across many IoT applications, such as autonomous robotics. This chapter crystallizes our vision of BlazeFlow, a framework designed to handle the communication flows at each of layer of the cloud-to-device continuum. BlazeFlow is designed to support the execution of time-critical services, supports various heterogeneous publish/subscribe-based protocols, and bridges them to form a unified communication path that can handle intra-node, intra-layer, and inter-layer communications across the edge, fog, and cloud layers.

A preliminary implementation and evaluation of BlazeFlow on a robotics use case scenario show that the solution can handle workload-intensive data (e.g., high-frequency sensor readings and images) with low latency and resource usage.

This work is the foundation of our vision of an end-to-end distributed autonomous robot solution. We envision that the framework will enable data for AI services to control the robot and for service distribution based on strict constraints that must be achieved. The development of an autonomous robot prototype in itself for medical applications will also be the subject of future work.

## CHAPTER 6

### **BLAZEAIOT: A MODULAR MULTI-LAYER PLATFORM FOR REAL-TIME DISTRIBUTED ROBOTICS ACROSS EDGE, FOG, AND CLOUD INFRASTRUCTURES**

Cédric Melançon<sup>1</sup> , Maarouf Saad<sup>1</sup> , Kuljeet Kaur<sup>1</sup> , Julien Gascon-Samson<sup>2</sup> , Simon Savard<sup>1</sup>

<sup>1</sup> Department of Electrical Engineering, École de Technologie Supérieure, Montréal, Québec, Canada

<sup>2</sup> Department of Software and IT Engineering, École de Technologie Supérieure, Montréal, Québec, Canada

Paper submitted to the journal « Elsevier Robotics and Autonomous Systems » on December 21st, 2025

#### **6.1 Abstract**

The increasing complexity of distributed robotics has driven the need for platforms that seamlessly integrate edge, fog, and cloud computing layers while meeting strict real-time constraints. This chapter introduces BlazeAIoT, a modular multi-layer platform designed to unify distributed robotics across heterogeneous infrastructures. BlazeAIoT provides dynamic data transfer, configurable services, and integrated monitoring, while ensuring resilience, security, and programming language flexibility. The architecture leverages Kubernetes-based clusters, broker interoperability (DDS, Kafka, Redis, and ROS2), and adaptive data distribution mechanisms to optimize communication and computation across diverse environments. The proposed solution includes a multi-layer configuration service, dynamic and adaptive data bridging, and hierarchical rate limiting to handle large messages. The platform is validated through robotics scenarios involving navigation and artificial intelligence-driven large-scale message processing, demonstrating robust performance under real-time constraints. Results highlight BlazeAIoT's ability to dynamically allocate services across incomplete topologies, maintain system health, and minimize latency, making it a cost-aware, scalable solution for robotics and broader IoT applications, such as smart cities and smart factories.

## 6.2 Introduction

The distributed robotics concept first emerged with the advent of cloud robotics (Quintas *et al.* (2011)) and has since evolved into edge robotics (Antevski *et al.* (2018)) and fog robotics (Krishna Chand Gudi *et al.* (2018)). At its core, distributed computing in robotics addresses the limitations of onboard resources and the restricted energy autonomy imposed by battery-powered systems. Offloading intensive computations to external layers alleviates these constraints, but simultaneously introduces challenges related to network connectivity (Sahni *et al.* (2017)), latency (Bédard *et al.* (2023)), and the complexity of distributed topologies (Lindsay *et al.* (2021)).

This work builds upon our previous work, a distributed data flow framework called BlazeFlow (described in Chapter 5), which helps bridging data in a multi-layer distributed configuration. The proposed BlazeAIoT platform is introduced to encompass the three different concepts of Distributed Robotics, namely, edge, fog, and cloud robotics, in a single solution. To demonstrate its feasibility, a concrete robotics scenario is presented that evaluates the distribution of computational tasks across these layers and examines the resulting impact on the robot execution. Given that most robotics services operate under strict real-time constraints, particular attention is paid to assessing whether services can be reliably executed at each layer, or whether network latency and other factors render specific deployments impractical.

### 6.2.1 Motivation

Although multiple distributed robotics platforms have been proposed in the literature, as described in Section 6.3, none support all three layers (edge, fog, and cloud) simultaneously. BlazeAIoT aims to introduce additional capabilities, as a platform, to address the challenges posed by such a distributed solution. The motivation for developing a novel Distributed Robotics platform is driven by multiple objectives:

- **Cost-aware solution:** Minimize platform cost by leveraging open-source components and maintaining vendor-agnostic compatibility with existing enterprise infrastructure.

- **Programming language flexibility:** Support multiple languages (C++, Python, C#, NodeJS, etc.) to facilitate integration with diverse services, particularly those built on the Robot Operating System (ROS) Macenski *et al.* (2022).
- **Multi-layer support:** Ensure robustness across edge, fog, and cloud layers and allow incomplete topologies (*e.g.*, missing a fog layer).
- **Flexible service distribution:** Allow services to be executed on any layer, except for specialized services requiring hardware interactions.
- **Real-time performance:** Select technologies that minimize overhead and meet stringent real-time constraints.
- **Robustness and security:** Provide resilience against connectivity loss and service failures, while ensuring data security both in transit and at rest.

While optimized for robotics, the platform is equally applicable to broader Internet of Things (IoT) and distributed system scenarios, including smart cities, smart factories, and more.

### 6.2.2 Contributions

The contributions of this work are as follows:

- **Service-oriented architecture for distributed robotics:** All computation is performed through services, enabling multi-layer deployment and configuration.
- **Dynamic data transfer:** Services dynamically share and consume data, with the platform automatically connecting related services across heterogeneous brokers.
- **Configurable services:** Standardized configurations with customizable options for deployment-specific needs, ensuring that services can adapt to the execution environment.
- **Monitoring solution:** Integrated mechanisms for performance metrics collection, enabling optimization and system health monitoring at the layer, node, and service level.
- **ROS integration:** Seamless integration of ROS packages into the platform to cover most of the robotics use cases (Portugal *et al.* (2024)), while maintaining service modularity.

- **Multi-layer deployment:** Demonstration of edge, fog, and cloud integration in a multi-cluster/multi-tenant environment, showing how services can be strategically allocated to balance latency, resource constraints, and computation speed.

### 6.2.3 Organization

This chapter is organized as follows. The state of the art is discussed in Section 6.3. Sections 6.4 and 6.5 detail the architecture of the proposed BlazeAIoT platform and its alignment with the objectives outlined in Section 6.2.1. Section 6.6 describes experimental infrastructure, including the robot, the various ROS packages, and the network topology. Section 6.7 presents the evaluation scenarios and analyzes the platform performance across multiple layers. Section 6.8 presents the discussion of the results and compares the platform with the objectives. Section 6.9 concludes with a discussion of results and outlines directions for future work.

## 6.3 Related Work

Many concepts in distributed robotics are not unique to robotics. Similar principles are applied in the Internet of Things (IoT), which also relies on edge, fog, and cloud topologies (Ahmed *et al.* (2023)). The authors describe two common IoT architectures: the three-layered architecture (perception, network, and application layers) and the service-oriented architecture (sensing, network, service, and interface layers). In Gangrade *et al.* (2025), the authors discuss the use of edge and fog computing in cyber-physical systems (CPS) to address limitations of cloud computing, such as high latency, limited bandwidth, and security/privacy concerns. Robots can be considered CPS because they are controllable systems that require real-time interactions.

Cloud robotics has been extensively studied over the past decade. In Hu *et al.* (2012), the authors present the cloud robotics concept and detail the implementation challenges. They identify computational challenges that can generate a high volume of data for network transfer, along with communication challenges such as network latency and potential connectivity loss. The XBot2D project (Muratore & Tsagarakis (2023)) proposes a cloud robotics architecture

for a Robot-as-a-Service (RaaS) scenario. To mitigate network faults, the XBot2D platform switches robot processes locally when network degradation is detected. The task is executed in both environments (cloud and local) at different rates (200 Hz and 1 KHz) to match resource availability. However, this approach continuously consumes resources across both layers, which is problematic for battery-powered robots.

Fog and cloud robotics solutions are compared in Krishna Chand Gudi *et al.* (2018) for human-robot interaction scenarios. The authors propose an offloading methodology to optimize the task execution. If the robot is unable to execute the task, it offloads it to the fog computing, which then offloads it to the cloud as a last resort. While cloud computing offers virtually unlimited resources, the multi-layer offloading introduces delays that can hinder real-time execution. Authors in Khalid *et al.* (2019) also introduce fog computing to an E-poll solution to reduce latency and improve the reliability of cloud computing. They also propose a configuration solution for the nodes that enables them to fetch their own configuration upon connecting to the network. The received configuration will influence packet routing to optimize communication speed and reliability. Fog computing has also been explored in other domains, such as healthcare Aiswarya *et al.* (2021). Authors in Ksentini *et al.* (2020) use a resource management simulator to prove the benefit of fog computing compared with cloud computing for a real-time and delay-sensitive scenario.

Edge robotics, in conjunction with cloud robotics, can benefit from 5G networks, as introduced in Roy *et al.* (2025). The authors proposed a quality-of-control-based resource allocation solution to optimize computing placement with respect to network delay, reliability, and task periodicity. Although promising, the placement strategy is static and does not account for dynamic changes in the robot environment. This aspect is not part of the actual work, but it is considered in the architecture for future improvements. Similarly, Baxi *et al.* (2025) explores 5G-enabled edge robotics in a pick-and-place process. A hybrid fallback mechanism is introduced that combines Quality of Service (QoS)-aware robot adaptation to address network instability and mitigate robot constraints, with Control-aware Dynamic QoS orchestration to ensure task execution

without duplicating processes across layers. They employ a neural network in their solution and emphasize the need for a robust simulator to enable integration in real-world settings.

Distributed robotics is frequently motivated by the need to offload Artificial Intelligence (AI)-related tasks to reduce the robot's computational burden. In that regard, the addition of a digital twin, as proposed in Riedlinger *et al.* (2022), is a valuable approach for collecting training data and leveraging simulation to improve AI models. The authors propose a distributed framework using Kafka as a data broker and ROS for robot communication. They divide the application into three functional layers: AI inference, the shop floor (with physical devices), and the edge cloud (responsible for the digital twin and non-AI computation). Orchestration of an AI model within a distributed robotics solution can also be challenging. The authors in Suneel *et al.* (2025) present a load-balancing solution for AI model inference that ensures real-time execution while prioritizing energy efficiency, particularly on battery-powered edge nodes.

Another essential aspect of distributed robotics is the ability to share data efficiently and reliably across layers. Authors in Serdaroglu *et al.* (2022) propose a comparison of the message-queue-based communication from Kafka with a conventional HTTP-based communication. They demonstrate that using a broker reduces latency as the number of nodes increases. Message volume significantly affects network latency. Authors in Montella *et al.* (2018) present an HTTP-based communication that segments a message optimally to be sent on multiple concurrent threads. Each segment is encrypted, and the server reassembles the message, thereby providing a secure means of sending such messages and reducing overall latency. The platform solution proposed in Sakai *et al.* (2022) also introduces a distributed architecture with a device registration process that changes the solution's topology, using a flow-sharing method to enable data flow. The authors implemented data communication using a Remote Procedure Calls (RPC) based approach in the cloud and used Message Queuing Telemetry Transport (MQTT) for inter-layer communication. MQTT is limited to high-volume, high-throughput scenarios, as explained in Melançon *et al.* (2023), which can be problematic for robotics applications. Another approach to minimize the latency is to use data compression to reduce the message size. Authors of Qingqing *et al.* (2019) propose an analysis of the compression effect for a visual odometry scenario. The

compression shows a slight degradation in odometry precision while significantly improving bandwidth savings. A balance between model precision and performance improvement can be achieved, depending on the task’s criticality.

The aforementioned works highlight the diversity of approaches for distributed robotics and IoT communication, ranging from cloud-centric architectures to fog and edge solutions, often relying on specific brokers or static placement strategies. Our previous work, BlazeFlow (Melançon *et al.* (2023)), introduced an initial vision of multi-layer data bridging and demonstrated the potential of dynamic communication across heterogeneous brokers. While BlazeFlow laid the foundation, it did not yet provide a complete platform for service deployment and orchestration. In contrast with the aforementioned approaches, BlazeAIoT advances this vision by adopting a service-oriented architecture that integrates configuration, monitoring, and adaptive data distribution across edge, fog, and cloud layers. This positions BlazeAIoT as a unified and resilient platform for real-time distributed robotics.

## 6.4 Proposed Solution

This section outlines the architecture of the proposed platform, named BlazeAIoT, which supports an autonomous robotics solution. We introduce and motivate each subsystem to meet the objectives.

### 6.4.1 System Overview

The high-level architecture of BlazeAIoT is represented in Fig. 6.1. The system is divided into three distinct subsystems: edge, fog, and cloud layers. All layers are interconnected via the Data Distribution Service (DDS) as the inter-layer communication protocol, thereby avoiding the need to always rely on the fog layer to forward messages, which would introduce more latency. Additionally, it enables a more flexible topology in which each layer may be optional or replicated for redundancy, ensuring high availability. Each layer is a Kubernetes cluster managed

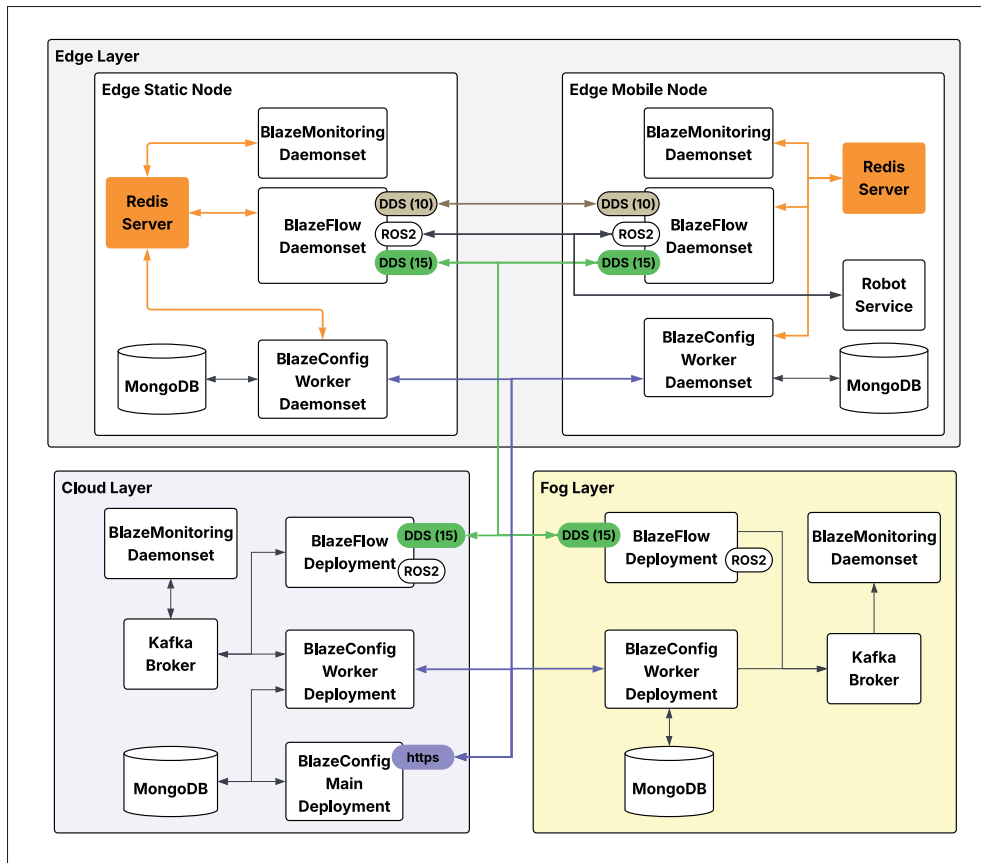


Figure 6.1 BlazeAIoT High-level architecture

with the open-source solution Rancher<sup>1</sup>. The edge layer cluster uses the lightweight open-source Kubernetes distribution K3S<sup>2</sup>, primarily because it is compatible with the ARM64 architecture and the limited resources of edge nodes. The fog and cloud clusters are running the open-source RKE2 Kubernetes<sup>3</sup>.

<sup>1</sup> <https://ranchermanager.docs.rancher.com/>

<sup>2</sup> <https://docs.k3s.io/>

<sup>3</sup> <https://docs.rke2.io/>

### 6.4.2 BlazeAIoT SDK: A standard interface for every service

The BlazeAIoT SDK is a library available in multiple languages. As of now, the supported languages are C++, C#, and Python, which cover most of the possible scenarios for the autonomous robot solution, but support for other languages could be easily integrated.

The SDK is a utility for integrating with the BlazeAIoT platform. A `ServiceManager` singleton is created for the service as a central access point for the various managing tools:

- **BaseConfiguration:** Interprets the configuration file and provides access to layer, node, and service configurations.
- **Broker:** Provides generic access to the local broker, enabling the publication of messages on a topic and the subscription to topics with a callback function for event-based scenarios.
- **ConfigManager:** Enables a service to respond to configuration changes by providing a callback function for altering node, layer, and service configurations.
- **FlowManager:** Enables the service to inform other services about the topics it generates (advertises) and the topics it consumes (requests).
- **MonitorManager:** Serves as a wrapper over Prometheus<sup>4</sup>, which is an open source metrics and monitoring middleware. It creates a metrics server that can be exposed and provides an asynchronous method for storing metrics, such as counters and gauges.
- **CacheManager:** Enables some functionalities leveraging Redis<sup>5</sup> as a cache. It is used to share service availability via a watchdog. It creates a timer that refreshes a hash value in the database. This value has an expiration, and is eventually removed from the database if it is not updated within the specified time.

The primary reason for using Redis to validate service availability is to synchronize interdependent platform services (for example, BlazeConfig requires BlazeFlow, and BlazeFlow requires BlazeConfig). Otherwise, undefined behavior can occur if the sequence is not respected.

---

<sup>4</sup> <https://prometheus.io/>

<sup>5</sup> <https://redis.io/community/>

### 6.4.3 BlazeConfig: Multi-layer Configuration

The BlazeConfig service is a new addition to the BlazeAIoT solution. It enables the management and synchronization of layer, node, and service configurations across all layers of the solution. BlazeConfig is composed of three components:

- **MongoDB Database:** this component is responsible for storing the configuration. At the edge, each node has its own MongoDB<sup>6</sup> instance, and in the fog and cloud, it can be deployed as a Kubernetes application or on a specific node.
- **BlazeConfig Main Service:** this service is hosted on the most centralized layer (the cloud layer when available, otherwise the fog layer). It exposes a REST API to manage globally the configuration of every aspect of the system. All the data is stored in the MongoDB database.
- **BlazeConfig Worker Service:** this service manages the local configuration for the node/layer. It is deployed as a Kubernetes DaemonSet on the edge layer (each node runs a single instance connected to the local MongoDB database) and as a Kubernetes Deployment on the fog and cloud layers. Each service gets its configuration from this service through the local broker (intra-node for the edge and intra-layer for the fog and cloud).

The worker service gets the latest configuration for the whole layer directly from the main service. The received configuration is used to synchronize the worker MongoDB database. After that, all interactions between the main service and the brokers are handled by the local broker. Each service affected by the change receives a notification via the local broker and takes appropriate action. The action can be to update the service's local parameters or the node/layer parameters, such as the database IP address. Each service is responsible for implementing its actions in response to a configuration change.

### 6.4.4 BlazeFlow: Multi-layer communication

A preliminary vision of BlazeFlow was first introduced in a workshop publication (see Chapter 5). The goal was to enable efficient, scalable multi-layer communication in distributed systems. The

---

<sup>6</sup> <https://www.mongodb.com/community/>

middleware introduced three different communication brokers. The intra-node communication uses Redis for local communication within a node. Intra-layer communication varied across layers. For node-to-node communication within the edge layer, DDS middleware was used; the fog and cloud layers used Kafka<sup>7</sup>. Finally, inter-layer communication, used to transfer messages between layers, was implemented using the DDS middleware.

BlazeFlow can also be leveraged for interoperability by enabling broker plugins that support additional protocols, such as ROS2. BlazeFlow bridges supported protocols in any direction (*e.g.* ROS2  $\leftrightarrow$  DDS  $\leftrightarrow$  Kafka).

While retaining the functionality name from previous work, BlazeFlow was significantly extended to integrate with BlazeAIoT by adding the capability to perform dynamic data distribution, defining the network configuration to be used in Kubernetes, enabling communication between multiple clusters, and introducing the concept of adaptive data distribution. Therefore, it represents a significant novel contribution to this work.

#### **6.4.5 BlazeMonitoring: System Monitoring**

BlazeMonitoring is a service that extracts essential platform metrics and makes them available for future optimization. The service is divided into two separate applications.

The BlazeMonitoring DaemonSet is spawned on every node for every layer. It is responsible for collecting node metrics, including CPU, GPU, memory, temperature, and power (when available). It is also used to monitor the latency between nodes in the solution. A ping timer task is executed at a configurable period of time (60 seconds) and sends a ping to all the nodes inside the solution (even the nodes outside the layer). When a node receives a ping message, it replies with a pong message, and the elapsed time divided by two represents the average latency between the two nodes. The primary reason for creating this task, rather than a typical Linux ping, is that it also accounts for the latency introduced by BlazeFlow bridges.

---

<sup>7</sup> <https://kafka.apache.org/>

The BlazeMonitoring Deployment can be used to collect specific metrics from a particular node. For this work, the message latency is computed to assess the impact of message size. Each message is timestamped before transmission by the first service. The monitor service listens to preconfigured topics and computes the elapsed time, which represents message latency (compared with the network latency from the DaemonSet).

All metrics are stored in the Prometheus server via the BlazeAIoT SDK and can be used to create Grafana dashboards or extracted for manipulation in a notebook, as used for the experimental portion of the project.

## **6.5 BlazeFlow as a Dynamic Data Bridge Solution**

This section discusses the key components integrated into BlazeFlow to meet the objectives of BlazeAIoT.

### **6.5.1 Dynamic Data Distribution**

BlazeFlow provides a dynamic data distribution paradigm that supports autonomous robotics solutions. This concept enables dynamic service spawning, ensuring that services receive the expected messages and that other services can process them.

BlazeFlow serves as a dynamic bridge that supports multi-layer communication. When a service is spawned on a node, it provides the advertised and requested topics to the local BlazeFlow service (via intra-node communication on edge nodes and intra-layer communication on fog/cloud nodes). BlazeFlow takes three actions from this information:

- Store the advertise/request topic to be able to take action when another service is involved with the same topic.
- Create a bridge between two protocols if a pair of advertise/request exists on the layer.
- Forward the advertise/request to the other layers. Source validation (node and layer) is performed to prevent message loops.

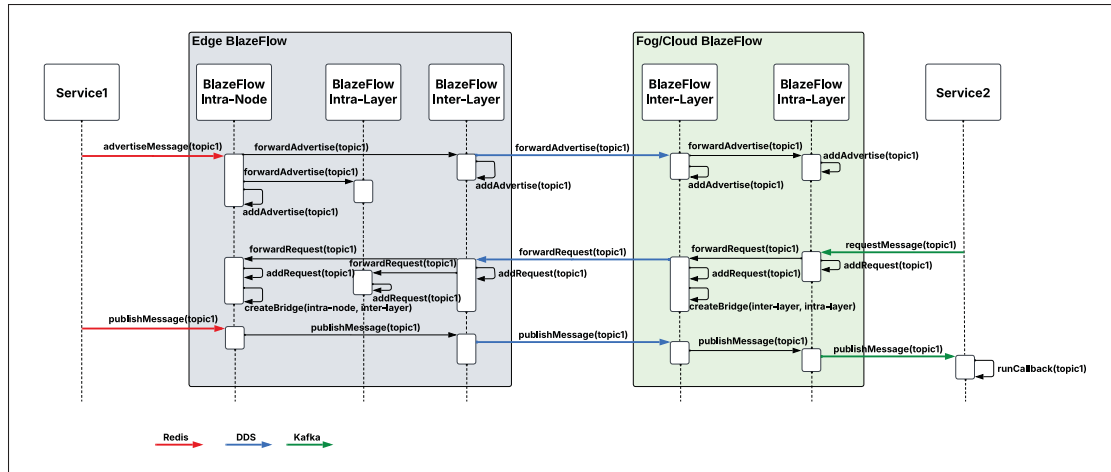


Figure 6.2 Multi-layer handshaking sequence

### 6.5.1.1 Bridge Creation Example

An example sequence for creating a bridge and the flow of a bridge message is shown in Fig. 6.2. The two BlazeFlow boxes include the various brokers that they support. Each broker instance is a subscriber to the remote BlazeFlow brokers. Each received message triggers a task to perform the BlazeFlow logic. In this example, Service1, hosted on an edge node, must send data to Service2, which is hosted on either the fog or the cloud layer (which makes no difference to the logic). The first action is for Service1 to announce the advertised topic 'topic1', given that Service1 wants to publish on that topic. The message is sent to the intra-node broker and received by the BlazeFlow service. The advertised message is forwarded to the intra-layer and inter-layer brokers and stored for each broker. At the next layer, the BlazeFlow service receives the advertised message via the inter-layer broker and forwards it to the intra-layer broker. It then stores the advertised topic for each broker.

The following action is for Service2 to announce the requested topic 'topic1' because Service2 is interested in receiving messages for that topic. The message is sent to the intra-layer broker and accepted by the BlazeFlow service. The requested message is forwarded to the inter-layer broker and stored for each broker. At the edge layer, the inter-layer broker receives the requested message and forwards it to the intra-layer and intra-node brokers, where each node stores it.

Since a pair of advertise/request is found, the BlazeFlow service needs to create bridges. On the edge node, a bridge is made to transfer data from the intra-node broker to the inter-layer broker. On the fog/cloud layer, a bridge is created to transfer data from the inter-layer broker to the intra-layer broker.

The final action is an example of a message sent by `Service1` on the topic 'topic1'. The BlazeFlow bridge receives the message and automatically publishes it to the inter-layer broker. On the other layer, the BlazeFlow bridge gets the message from the inter-layer broker and forwards it to the intra-layer broker. The `Service2` receives the message and invokes the action associated with the message to perform a task (*i.e.*, the callback).

The logic of bridge creation depends on the availability of an advertise/request pair for the same topic and its origin. The goal of BlazeFlow is to transparently establish a continuous data flow from the original service to the destination services. If the destination is within the same layer, the intra-layer communication mechanism is used (simple case), and no bridging is required; however, if the destination is outside the layer, inter-layer communication is used. Multiple destinations may be necessary, requiring BlazeFlow to route messages across various brokers.

#### 6.5.1.2 One-to-many Scenario Example

Fig. 6.3 represents an example of a complex scenario with one publisher (`Service1`) and multiple subscribers (`Service2` and `Service3`) dispersed on various layers. A flow diagram is shown in this example to abstract away some details. `Service1` provides data via the ROS2 broker.

`Service2` is deployed on the same node and on the cloud layer. `Service2` is using the default BlazeFlow broker. To receive data at the edge layer, a bridge is created to publish ROS2 messages to the intra-node broker (Redis Pub/Sub). For the `Service2` hosted in the cloud layer, a bridge is required to transmit ROS2 data to the inter-layer broker (DDS). In the cloud layer's BlazeFlow, a bridge is necessary to transfer data from inter-layer communication (DDS) to intra-layer communication (Kafka), enabling `Service2` to receive it.

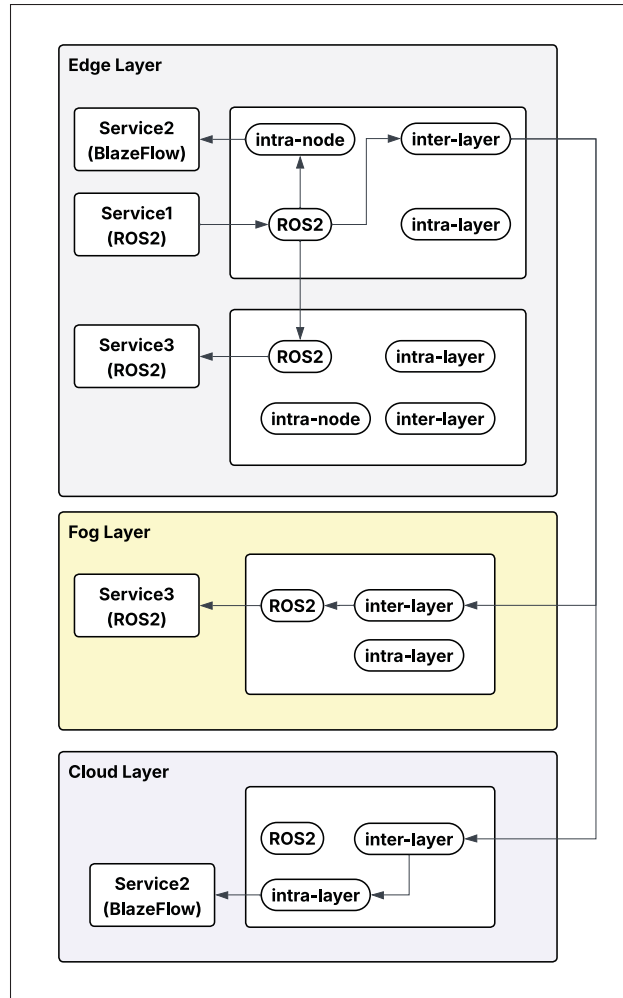


Figure 6.3 Multi-bridge scenario example

Service3 is deployed on a separate edge node and on the fog layer. Service3 is expecting data from the ROS2 broker. Since, as explained earlier, ROS2 is considered local within the same layer, the Service3 hosted on a separate edge node will receive data directly from the same ROS2 network without requiring a bridge. For the Service3 hosted on the fog layer, it will utilize the existing bridge on the edge layer (ROS2 → inter-layer). Additionally, a bridge will be created on the fog layer's BlazeFlow to transfer data from the inter-layer broker (DDS) to the ROS2 layer.

### 6.5.2 Network Considerations

Brokers that are not directly provided by the platform (intra-node, intra-layer, and inter-layer) are considered accessible from within the layer because all nodes are part of the same Kubernetes cluster. A macvlan interface is created on the cluster to share the data using Multus<sup>8</sup> and Whereabouts<sup>9</sup>. This ensures that all nodes are sharing the same virtual network.

Inter-layer communication is enabled by Submariner<sup>10</sup>, which establishes an IPsec tunnel between clusters. Other, more efficient networking solutions are available but are more complex to configure; therefore, the default configuration was used. IPsec tunneling is highly CPU-intensive, which limits network performance; however, it ensures that inter-layer traffic is encrypted. Due to the limited resources, the available bandwidth for inter-layer communication is about 160 Mbps. This could be improved for industrial deployment by using gateway nodes with a faster CPU, or by using a different cable driver technology.

### 6.5.3 Adaptive Data Distribution

To optimize data flow, all bridge actions are performed in separate tasks submitted to a thread pool sized according to the node's available resources. The same applies to the service callbacks. The cost of this parallelization is increased resource consumption (memory and compute), but it also enables better performance to meet real-time constraints. Without this parallelization, the support for large messages significantly slowed the overall data distribution, making it unusable in a distributed robotics scenario.

To support large messages, such as camera images, a rate-limiter logic was implemented in BlazeFlow. Small messages are prioritized, and large messages have their rate reduced by skipping frames to match the allowed rate within the bandwidth budget. For instance, if we were to replace IPsec with an unencrypted tunnel (at the cost of reduced data privacy), the allowed

---

<sup>8</sup> <https://github.com/k8snetworkplumbingwg/multus-cni>

<sup>9</sup> <https://github.com/k8snetworkplumbingwg/whereabouts>

<sup>10</sup> <https://submariner.io/>

bandwidth budget would increase, and the framework would automatically adapt its rate-limiting logic.

The rate-limiting logic implemented by BlazeAIoT is a two-tier hierarchical algorithm that combines client-level bandwidth allocation with publisher-level token-bucket regulation. At the client level, available bandwidth is defined in Eq. 6.1.

$$B_{\text{avail}} = L_{\text{client}} \cdot \frac{1024^2}{8} \text{ bytes/sec} \quad (6.1)$$

where  $L_{\text{client}}$  is the limit in Mbps.

Bandwidth is distributed among publishers using a priority-based allocation strategy: standard publishers receive their advertised rates first. In contrast, large-message publishers are allocated the remaining bandwidth, with a minimum threshold of  $r_{\min} = 2$  Hz to prevent starvation. The algorithm accounts for message overhead with a factor  $\alpha_m = 1.02$  and enforces a configurable per-message maximum bandwidth percentage  $\beta_b = 0.95$  (default 95%) to prevent monopolization. Publishers are sorted by bandwidth demand  $D_i = r_i^{\text{adv}} \cdot M_i \cdot \alpha_m$  in descending order, where  $r_i^{\text{adv}}$  is the advertised rate and  $M_i$  is the message size in bytes. For each publisher  $i$ , the allocated rate is computed with Eq. 6.2.

$$r_i^{\text{alloc}} = \min \left( r_i^{\text{adv}}, \frac{B_{\text{remaining}}}{M_i \cdot \alpha_m}, \frac{\beta_b \cdot B_{\text{avail}}}{M_i \cdot \alpha_m} \right) \quad (6.2)$$

where  $B_{\text{remaining}}$  is updated after each allocation with Eq. 6.3.

$$B_{\text{remaining}} \leftarrow B_{\text{remaining}} - r_i^{\text{alloc}} \cdot M_i \cdot \alpha_m \quad (6.3)$$

At the publisher level, each topic employs a token bucket filter with capacity  $C = 2$  tokens, as defined in Eq. 6.4.

$$T(t) = \min \left( C, T(t - \Delta t) + r_i^{\text{alloc}} \cdot \Delta t \right) \quad (6.4)$$

where  $T(t)$  is the token count at time  $t$  and  $\Delta t$  is the elapsed time in seconds.

A message is transmitted only if  $T(t) \geq 1$ , consuming one token per publish:  $T(t) \leftarrow T(t) - 1$ . This rate-limiting mechanism allows brief catch-up bursts above the nominal rate while maintaining long-term average compliance. It focuses on delivering small-payload messages at their publish rates, as much as possible, while reducing the rate of large-payload messages, which consume most of the bandwidth.

To reduce message size, the payload can be compressed using the LZ4 algorithm with a configurable compression level. The default level for large messages is 10, which provides a reduction of 40-60% depending on the image content. Since a compressed image has not a constant size, the rate-limiting algorithm always consider the maximum size for a given topic and will adjust the rate when a bigger message is received.

## 6.6 Evaluation Testbed

This section describes the experimental environment used to evaluate the BlazeAIoT platform. It presents the infrastructure topology, including the edge, fog, and cloud layers, and describes the robot and the various components needed for the evaluation.

### 6.6.1 Infrastructure

To validate the solution, the infrastructure detailed in Fig. 6.4 is used. Due to network restrictions, a separate server was configured to act as an on-premise cloud server.

All layers are providing a Kubernetes cluster. To facilitate configuration, the open-source solution Rancher was installed. The primary reason for selecting this application is its ability to deploy RKE2 Kubernetes clusters (for the fog and cloud layers) alongside K3S Kubernetes clusters (for the edge layer).

To allow multi-cluster communication, the application Submariner was configured. Each layer provides at least one gateway through which network communication flows. Each cluster's gateway shares a virtual network managed by the main deployment (configured as an operator).

Rancher provides a monitoring application that includes a Prometheus server that gathers metrics to monitor the cluster and Kubernetes performance. It also provides a Grafana deployment with multiple pre-bundled dashboards.

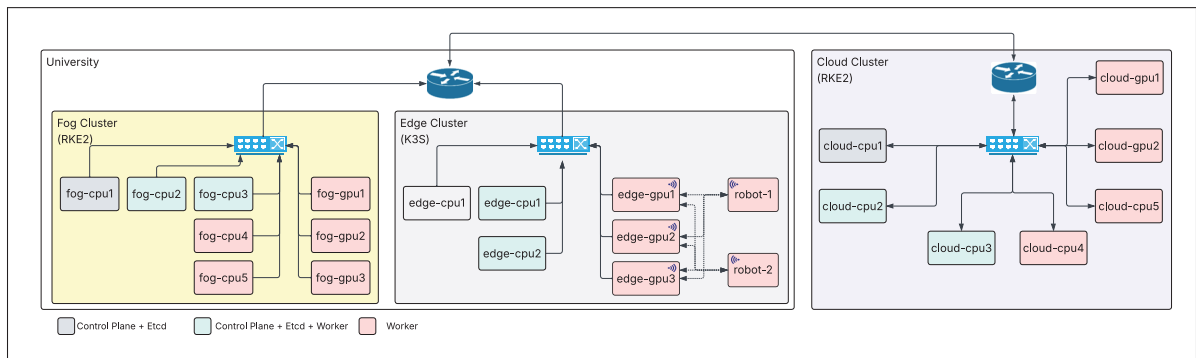


Figure 6.4 Experimental server infrastructure

The layer's default configuration, its nodes, and all services are stored in a Kubernetes ConfigMap that can be mounted as a volume in the services. This is useful when the BlazeConfig service is unavailable, enabling configuration of basic functionality, such as the broker, database, and cache addresses and ports.

#### 6.6.1.1 Cloud cluster

All the nodes in the cloud cluster are connected to the same network switch. The switch has access to the internet and the other clusters through a router. Since the cloud network is located near the other clusters, the scenario is made more realistic by configuring the router to introduce latency on the port connected to the router hosting the fog and edge clusters. A 50 ms latency with 10 ms jitter is added along with a packet loss of 0.01% (Microsoft (2025)).

The cloud cluster is an RKE2 cluster composed of seven nodes. Since using real servers with virtualization was not possible, each node is a computer with a different configuration. Table 6.1 describes the configuration of every node.

Table 6.1 Cloud cluster nodes

| Node name  | CPU | RAM    | GPU (Memory) |
|------------|-----|--------|--------------|
| cloud-cpu1 | 2   | 16 Gb  | 0            |
| cloud-cpu2 | 2   | 16 Gb  | 0            |
| cloud-cpu3 | 4   | 16 Gb  | 0            |
| cloud-cpu4 | 4   | 32 Gb  | 0            |
| cloud-cpu5 | 4   | 32 Gb  | 0            |
| cloud-gpu1 | 12  | 32 Gb  | 1 (8 Gb)     |
| cloud-gpu2 | 20  | 32 Gb  | 1 (24 Gb)    |
| Total      | 48  | 176 Gb | 2 (32 Gb)    |

A Kafka controller/broker application is deployed to serve as intra-layer communication. Inter-layer communication using DDS is enabled through a Kubernetes service exposed via Submariner. A Redis application is also deployed on the cluster.

The cloud-cpu3 has MongoDB installed outside Kubernetes and exposed its service for the other nodes.

The BlazeConfig Main service is deployed on the cloud cluster, and the REST API is exposed through a Kubernetes service accessible via Submariner. The BlazeMonitoring, BlazeFlow, and BlazeConfig worker applications are also deployed on the cluster.

#### 6.6.1.2 Fog cluster

The fog cluster is also an RKE2 cluster and is very similar in configuration to the cloud one. However, instead of having its own network, it shares the edge cluster's network via the same router. All nodes are connected to the same network switch, which is connected to the router that provides internet access and connectivity to the cloud cluster network. The Table 6.2 details the configuration of every node in the fog network.

Table 6.2 Fog cluster nodes

| Node name | CPU | RAM    | GPU (Memory) |
|-----------|-----|--------|--------------|
| fog-cpu1  | 8   | 8 Gb   | N/A          |
| fog-cpu2  | 8   | 16 Gb  | N/A          |
| fog-cpu3  | 8   | 16 Gb  | N/A          |
| fog-cpu4  | 2   | 4 Gb   | N/A          |
| fog-cpu5  | 8   | 16 Gb  | N/A          |
| fog-gpu1  | 8   | 16 Gb  | 1 (8 Gb)     |
| fog-gpu2  | 8   | 16 Gb  | 1 (8 Gb)     |
| fog-gpu3  | 12  | 85 Gb  | 1 (48 Gb)    |
| Total     | 62  | 177 Gb | 3 (64 Gb)    |

Similar to the cloud cluster, the fog cluster is configured with Redis and Kafka for intra-layer communication, an exposed service in Submariner for inter-layer communication, and the BlazeAIoT applications (BlazeFlow, BlazeConfig worker, and BlazeMonitoring).

### 6.6.1.3 Edge cluster

The edge cluster is a K3S cluster because of its node type. The nodes used for this work range from small computers, such as Raspberry Pi 5, to various Jetson nodes (Nano, NX, AGX). Additionally, the robots are equipped with an edge node, specifically a Jetson Orin AGX. The CPU nodes are installed near the network switch to minimize the latency. The GPU nodes, which are Jetsons, are dispersed throughout the test area and connected to the network switch via long Ethernet cables. They also serve as wireless gateways (configured as WiFi access points). The robot is a wireless node that connects to available access points to help minimize latency. The robot nodes also include GPUs, enabling them to perform critical AI tasks.

The Table 6.3 details the configuration of every node in the edge network.

Each edge node has a local installation of MongoDB and Redis (outside Kubernetes). The local Redis server is used for intra-node communication and caching. Intra-layer communication with DDS is facilitated by a Multus/Whereabouts network attachment definition that creates a MACVLAN network shared by all nodes in the cluster. The primary reason for this configuration

Table 6.3 Edge cluster nodes

| Node name | Node type | CPU | RAM    | GPU (Memory) |
|-----------|-----------|-----|--------|--------------|
| edge-cpu1 | Static    | 4   | 8 Gb   | N/A          |
| edge-cpu2 | Static    | 8   | 16 Gb  | N/A          |
| edge-cpu3 | Static    | 8   | 16 Gb  | N/A          |
| edge-cpu4 | Static    | 16  | 16 Gb  | N/A          |
| edge-gpu1 | Gateway   | 6   | 8 Gb   | 1 (8 Gb*)    |
| edge-gpu2 | Gateway   | 8   | 16 Gb  | 1 (16 Gb*)   |
| robot-1   | Mobile    | 8   | 32 Gb  | 1 (32 Gb*)   |
| Total     | -         | 66  | 144 Gb | 3 (56 Gb*)   |

\* Shared memory

is that it enables UDP multicast, which can reduce DDS bandwidth requirements when multiple nodes share the same topics. Inter-layer communication is set up the same way as the other clusters, via a Kubernetes service exported by Submariner.

The BlazeFlow, BlazeMonitoring, and BlazeConfig worker applications are deployed as DaemonSets, meaning each node runs one instance.

### 6.6.2 Robot

The robot is built on the Pioneer 3-AT platform, as described in Melancon *et al.* (2023), with an upgrade from the Jetson computer to the Jetson Orin AGX as represented in Fig. 6.5. The following subsections describe the ROS packages used for the robot navigation.

#### 6.6.2.1 Robot Localization Service

ROS2 uses a set of standard coordinate frames to localize robots on the map. These are the map frame, which represents the global area in which the robot operates, the odom frame, which is placed on the robot's starting location in the map frame and tracks the robot's movement relative to their starting position, and finally the robot's frame, which is connected to the robot's base link and represents the robot's position itself.



Figure 6.5 Modified Pioneer 3-AT

This standard frame separation allows separation of concerns when operating the robot platform. The map frame can be used to measure the robot's position within the area of interest. However, given the relatively slow update rates of global localization methods such as SLAM and UWB, which tend to produce discrete jumps, they cannot be used directly for low-level robot control. On the other hand, the odom frame is guaranteed to be continuous and thus suitable for robot control, but it suffers from drift due to odometry sensor errors.

Robot localization is performed using a pair of Kalman filters implemented in ROS2 with the Robot Localization package (Moore & Stouch (2016)). The local filter generates a transform from the odom frame to the base link using wheel odometry and the IMU's accelerometer and gyroscope.

The robot pose frequency calculated by the SLAM system is constrained by several factors, including the LiDAR scan rate and the local Kalman filter's position update rate, resulting in

a frequency of 1.5 Hz that reduces navigation reliability. To increase this frequency, a global Kalman filter fuses the same odometry data as the local Kalman filter, along with the SLAM pose, which is more discrete than the odometry and has a higher frequency. This Kalman filter interpolates positions between SLAM poses published by odometry, enabling it to publish a global position at 20 Hz.

To have a comparable ground truth, an Ultra Wideband (UWB) module is used to generate absolute position in the map frame. To filter the UWB's discrete, highly noisy data, an identical set of local and global Kalman filters is used, replacing the SLAM pose with the UWB pose to produce a more reliable UWB ground-truth signal at 20Hz.

#### **6.6.2.2 Pure Pursuit Service**

The path-following algorithm used is a pure-pursuit controller, which can be viewed as a carrot-and-stick method (Coulter (1992)). A line is drawn between the waypoints to build a path. The robot has a lookahead distance forming a circle around it. The controller identifies a goal point by finding the intersection of this circle with the line formed by the waypoints, favoring intersections closer to the next waypoint. The controller then uses the robot's localization data and the target point's position to compute a command for the current time step. This process is repeated at 25 Hz per iteration until the robot reaches the goal point, the last waypoint in the list. The controller computing the command from the localization and target point is a Proportional-Integral-Derivative (PID) controller.

#### **6.6.2.3 Vision-Based Safety Stop Service**

This service is a ROS2 package that detects a halt hand gesture (all fingers extended) and generates a safety stop.

The architecture of the AI model used for the halt detection is detailed in Fig. 6.6. The hand landmark extraction leverages the MediaPipe hand solution (Google LLC (2020)), which consists of a lightweight CNN pre-trained on a large dataset of hand images. The landmark detection

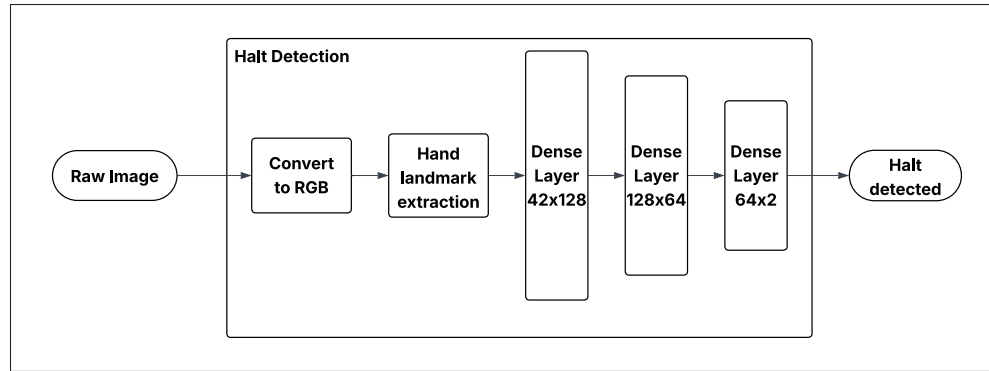


Figure 6.6 Safety stop detection model

pipeline passes through three dense layers before making a final decision on whether the halt gesture is detected. The ROS package provides a safety stop indication message that the robot can use to halt all motion. The model was trained with the HAGRID dataset (Alexander *et al.* (2024)).

## 6.7 Experimental Results

Based on the experimental infrastructure detailed in Section 6.6, the proposed BlazeAIoT platform is evaluated in two scenarios to assess the impact of computing distribution on a real-time application. Each scenario aims to validate one particular aspect of the platform. The first scenario focuses on small messages with a high data rate to assess the feasibility of running a typical navigation at any layer. The second scenario combines large messages with those from the first scenario to evaluate adaptive data distribution and its impact on a critical scenario, such as a safety stop.

### 6.7.1 Scenario 1: Navigation

The first scenario is a simple navigation scenario without active obstacle avoidance. The robot navigates the lab using the pure pursuit algorithm, moving from an office room to the end of an open area.

The experiment is divided into five steps. Step 1 is to run all computations locally on the robot edge node, directly in ROS2, rather than via the BlazeAIoT framework. This step is considered the baseline for experimentation. The goal is to assess the platform's overhead and the impact of latency on a typical navigation scenario when running services at any layer by comparing with the baseline execution.

The following steps are performed within the BlazeAIoT framework. Step 2 runs all the robot services on the robot's edge node. Step 3 keeps the P3AT services on the robot's edge node, while all other services run on the edge layer (remote to the robot). Steps 4 and 5 are similar to the third step, but service execution that is not related to the robot itself occurs on the fog and cloud layers, respectively.

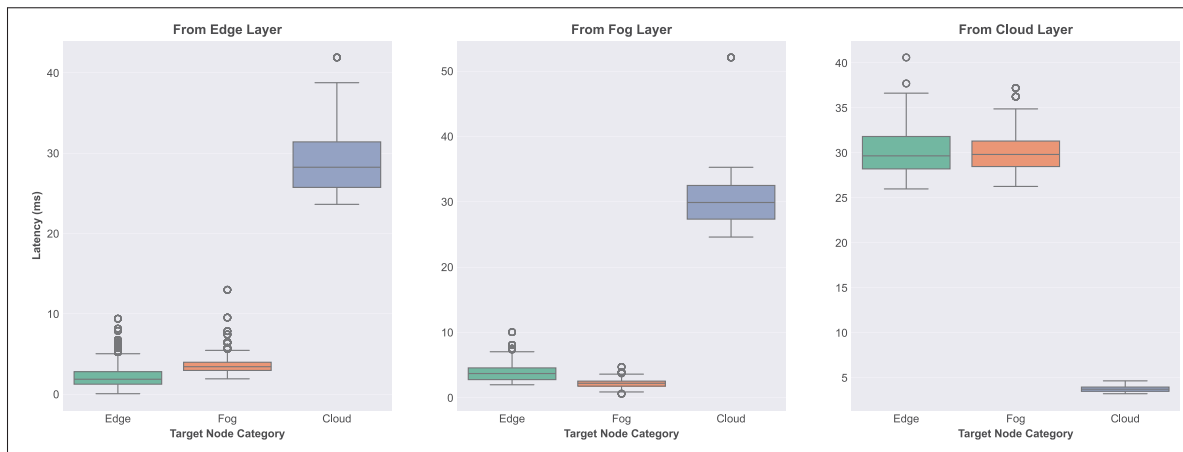


Figure 6.7 Latency distribution

To help interpret the results, Fig. 6.7 shows the latency distribution observed for each layer when pinged by every other layer using the BlazeMonitor ping service. The latency on the same layer is approximately 2.4 ms in the edge layer, 2.2 ms in the fog layer, and 3.8 ms in the cloud layer. Communication with the cloud is significantly affected by latency/jitter emulation, resulting in an average latency of approximately 30.1 ms. Communication with the fog exhibits an average latency of approximately 3.7 ms from the edge layer.

Table 6.4 shows the average latency of the ROS topics along with their rate of production. To achieve this latency, a subscriber probe is created at each layer, and each subscriber computes latency by comparing the header timestamp (generated by the ROS2 node) with the local time. All nodes are synchronized to the same Network Time Protocol (NTP) server, minimizing time drift.

Table 6.4 ROS topic latency

| Topic           | Rate (Hz) | Robot Latency (ms) | Edge Latency (ms) | Fog Latency (ms) | Cloud Latency (ms) |
|-----------------|-----------|--------------------|-------------------|------------------|--------------------|
| encoder_odom    | 49.71     | 3.26               | 12.11             | 10.76            | 29.26              |
| odometry/global | 19.09     | 7.68               | 18.98             | 16.22            | 32.81              |
| pose            | 1.09      | 218.68             | 285.18            | 219.71           | 216.00             |
| scan            | 4.56      | 142.84             | 148.70            | 147.37           | 155.08             |
| imu             | 49.17     | 3.30               | 11.26             | 9.38             | 28.14              |
| /tf             | 24.65     | 21.62              | 16.67             | 17.98            | 34.43              |

The latency of the scan (~145 ms) and pose (~280 ms) topics is relatively high, but this is not necessarily due to the framework. The LiDAR scan provides a header timestamp from the start of the full rotation, which takes 100 ms. The pose topic originates from SLAM, which depends on scan and odometry/global topics, and on the algorithm's processing requirements, which depend on the node's available resources.

The resulting navigation for scenario 1 is presented in Fig. 6.8, and a video is also available to show the robot behavior for each execution<sup>11</sup>. The base run is the one executed directly on the robot node without the BlazeAIoT framework. It is possible to see that the runs executed within the BlazeAIoT framework follow the shape of the base path, with a slight deviation. The error is more pronounced during rotations, which can be explained by network latency that slows the SLAM pose broadcast.

Fig. 6.9 shows the Euclidean distance error relative to the base run. The edge run shows a greater error (up to 10.7 cm), which is the opposite of what would be expected. Still, the reason for that is the increased traffic and CPU load introduced by the addition of two subscribers (one

<sup>11</sup> <https://youtu.be/4Bp8EqYhM5c>

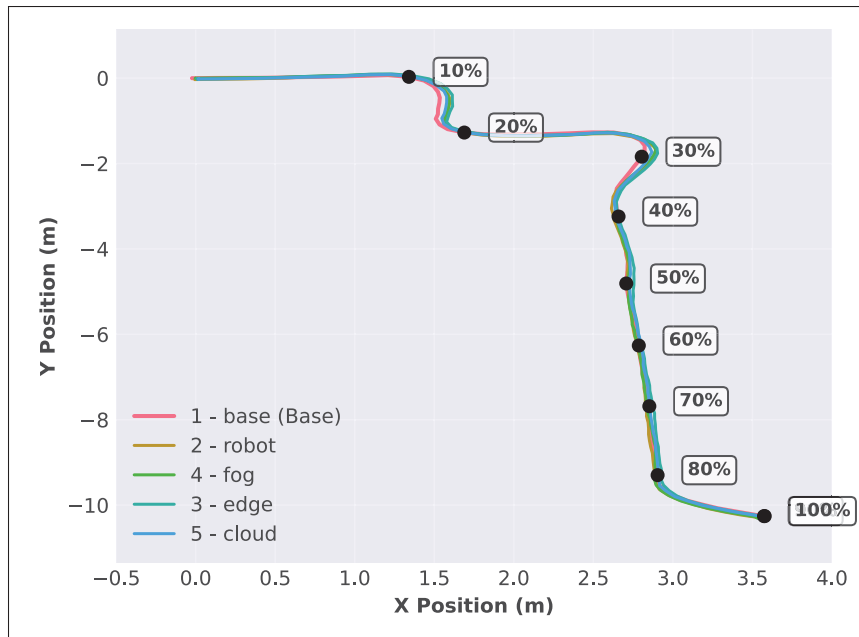


Figure 6.8 Spatial map

on the robot node and one on the edge node) to be able to do the latency monitoring. The fog run shows lower error than the edge run (at most 8.5 cm), attributable to the fog node's low latency combined with high computational power. The robot run shows better performance than the fog run with an error of at most 7.3 cm. The cloud presents the highest error (up to 12.9 cm), but follows the same path as the other runs. The main reason for this increased error is the greater network latency (+30 ms). When the robot receives a command, it executes almost two control steps (at 50 Hz). The robot is always trying to catch up for the delay.

Unsurprisingly, the best allocation for this scenario is to run everything on the robot's edge node. The communication latency is negligible, and because the entire communication runs directly on ROS2 without a bridge, there is no overhead. This is true when the robot's node resources are sufficient. Adding more services to the robot may yield a different conclusion. This is the objective of the following scenario, in which the robot's node resources are pushed to their limits.

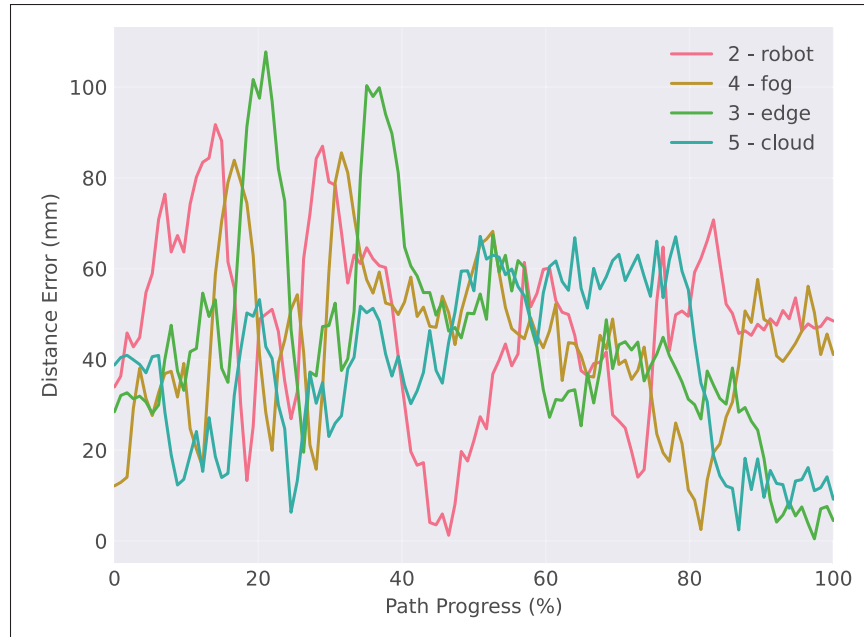


Figure 6.9 Total Distance Error (relative to Base)

### 6.7.2 Scenario 2: Vision-based safety stop

The second scenario is similar to that in Muratore & Tsagarakis (2023) and a video is available to visualize the scenario execution<sup>12</sup>. It involves moving the robot in a straight line and using hand gesture detection to detect a fully open hand, thereby triggering a safety stop. The same four steps described for scenario 1 are executed. The goal is to evaluate the effect of latency in a high-volume, high-throughput scenario. The RGB images transmitted by the camera and shared through the BlazeAIoT framework are significantly influenced by the size of the image message packet. Additionally, the safety stop service uses the AI model shown in Fig. 6.6, which is also resource-intensive relative to the other services. The relevant metric in this scenario is the time difference between the hand gesture's appearance and the robot's response.

To compare the network's effect on the robot's response, the safety-stop ROS node was launched simultaneously across all layers. A monitoring service subscribes to the safety stop topic, receives messages from all layers, and can compare their latencies. The layer nodes running

<sup>12</sup> [https://youtu.be/PYek5ER0\\_MM](https://youtu.be/PYek5ER0_MM)

the safety stop ROS node are robot-1, edge-gpu2, fog-gpu3, and cloud-gpu2. It is important to note that the MediaPipe hand-gesture algorithm runs on the CPU, as a GPU-based version was not natively available on the Jetson. Still, the results highlight an interesting aspect for resource-constrained nodes.

Table 6.5 presents the metrics obtained during execution.

Table 6.5 Emergency stop metrics

| Node       | Image Rate (Hz) | Image Latency (ms) | Processing Time (ms) | Safety Stop latency (ms) | Difference (%) |
|------------|-----------------|--------------------|----------------------|--------------------------|----------------|
| robot-1    | 27              | 38.6               | 430                  | 1287.3                   | -              |
| edge-gpu2  | 21              | 40.9               | 190                  | 581.6                    | -54.8          |
| fog-gpu3   | 7.9             | 44.7               | 23                   | 181.3                    | -85.9          |
| cloud-gpu2 | 7.9             | 172.5              | 19                   | 550.74                   | -57.2          |

The variation in image frame rate is due to rate-limiting logic that respects the allowed bandwidth budget. The camera images are the first topic to be bridged through BlazeFlow. The remaining messages are then added, including the scan, tf, encoder\_odometry, and imu topics, which together constitute the largest payloads after the camera images.

The service running on the robot receives images at 22 Hz, which is lower than the expected 30 Hz. The first justification for this difference is the node memory load. Since the Jetson shares memory with the CPU and GPU, the combination of the safety stop process, BlazeFlow execution, and all robot processes places a significant memory demand, ranging from 97% to 100%. The result is that safety stop service takes an average of 725 ms to execute on one image. Because the image callback is reentrant, multiple frames are processed in parallel (approximately 16 concurrent processes), thereby increasing the typical load. The other justification for the rate difference is based on the metrics-monitoring methodology, as in scenario 1. To be able to compute the topic latency, an additional subscriber is added on the robot node and on the edge node, which increases the network traffic and the necessary CPU on the robot since the multicast is not leveraged with the default ROS configuration, meaning that the robot node sends the topic message four times instead of 2, adding to the CPU burden.

The service running on the edge node receives the images at the same rate as the robot. Still, because the node is not running the robot and sensor services, the edge node's memory usage is approximately 85%, resulting in a 190 ms process time. The safety stop service remains slower than the input rate, resulting in an accumulation of process threads (approximately four concurrent processes).

To assess whether the hand gesture is detected, three consecutive positive detections are required. If the service is performed locally on the robot, without the network latency, it takes approximately 861 ms to generate the safety stop signal. On the edge node, the safety stop is triggered after 333 ms, whereas on the fog and cloud layers, it is triggered after 380 ms.

From the results, the safety stop executes faster on the fog layer, with an 859% reduction in time compared to running it on the robot directly, even though the image rate to the fog is about 26% of the expected 30 Hz. The cloud, even with a reduced rate and higher latency, offers a 572% reduction in time. Finally, remote execution of the safety stop on a different edge node results in a 548% decrease in execution time. While fog and cloud execution offer better response times, the frame-rate reduction could be problematic due to the risk of losing information from filtered frames.

It should be noted that the safety stop service runs in all scenarios, creating a worst-case timing issue due to high CPU/memory usage. If the service is allocated to a remote node, overall performance would improve.

## **6.8 Discussion**

Across the two scenarios, it is evident that when a solution handles small messages, latency has little impact on the robot's behavior for non-critical tasks. If a fast reaction is required, the time added by the network latency can be problematic. For a solution that handles large messages and computationally intensive tasks, the ability to deploy the service to another layer can be beneficial. If the local node is running multiple services, limited CPU availability can increase

processing time. While latency is higher in the other layers, a high-end CPU/GPU can be much faster than a typical edge node and compensate for the communication speed.

The BlazeAIoT framework facilitates communication by compressing payloads and rate-limiting, reducing the message rate to fit within the allowed bandwidth. Service allocation is essential, but orchestration is not part of the actual implementation. With BlazeFlow's dynamic capabilities, moving a service from one node to another is seamless, as all services broadcast their advertise/request topics.

### 6.8.1 BlazeAIoT vs. Objectives

After describing the BlazeAIoT components, it is possible to assess how the platform addresses the objectives enumerated in Section 6.2.1.

- **Cost-aware solution:** all the technologies used in the BlazeAIoT platform are Open-Source technologies (MongoDB, Kafka, RKE2, K3S, DDS, Redis, and Prometheus). Using Rancher as a Kubernetes management tool facilitates the creation of lightweight Kubernetes clusters with K3S and full-scale Kubernetes clusters with RKE2. This solution can be deployed on any cloud provider or replaced with other Kubernetes services, such as Azure Kubernetes Service, Amazon Elastic Kubernetes Service, and Google Kubernetes Engine. The result is a cloud-agnostic solution that can easily deploy within an enterprise's existing infrastructure.
- **Programming language flexibility:** BlazeFlow data technologies were selected based on this objective. The platform's clients do not directly access DDS. They would rather connect to Kafka or Redis, which provide client libraries in multiple languages. ROS2 packages can still communicate directly in the same layer. A converter is natively included in BlazeFlow to translate between any broker and ROS2. Client services typically use the SDK, which is available in multiple languages, and it is straightforward to add support for new languages as needed.
- **Multi-layer support:** BlazeFlow enables communication in all layers. In Kubernetes, a virtual private network is available to all nodes within the cluster. To enable multi-layer,

multi-cluster communication, IP tunneling, such as Submariner, bridge that gap through an encrypted tunnel.

- **Flexible service distribution:** With BlazeConfig, all services can fetch the layer, node, and service configuration. It is possible to have multiple instances of a service, each with its own configuration if needed.
- **Real-time performance:** BlazeFlow broker's performances were studied in a previous work (Melançon *et al.* (2023)). To monitor the metrics of the service within the platform, BlazeMonitor enables them for eventual improvement to place the services to ensure the real-time execution.
- **Robustness and security:** The connectivity loss scenario was not covered by this work, but all the actual development makes it easy to spawn the required services locally if the resources are sufficient to take over while waiting for the connectivity to come back. Security is a topic in its own right, but all selected technologies offer secure options. For all the communications, it is possible to encrypt the data. MongoDB also offers data-at-rest encryption. Using Kubernetes limits access to its internal network and provides authorization and role-based access controls to help secure the entire solution. Traffic between layers is also encrypted using an IPsec tunnel. It would also be possible to add monitoring services within BlazeMonitor to collect metrics for this objective and to take action. But this was not part of the current implementation.

## 6.9 Conclusion

This work presented BlazeAIoT, a modular multi-layer platform designed to unify distributed robotics across edge, fog, and cloud infrastructures. By extending the BlazeFlow framework, the proposed solution addresses critical challenges in distributed robotics, including dynamic data transfer, adaptive communication, real-time performance, and resilience against connectivity loss. The integration of BlazeConfig for multi-layer configuration management, BlazeFlow for dynamic and adaptive data distribution, and BlazeMonitoring for system health metrics provides a comprehensive foundation for robust and scalable deployments.

Experimental validation through navigation and AI-driven message handling scenarios demonstrated the platform's ability to meet stringent real-time constraints while maintaining flexibility across distributed topologies. The results confirm that BlazeAIoT can dynamically allocate services, optimize bandwidth usage, and ensure interoperability across heterogeneous environments.

Beyond robotics, the platform's design principles make it equally applicable to broader IoT domains, including smart cities, smart factories, and cyber-physical systems. Future work will focus on enhancing load balancing in BlazeFlow, exploring more efficient inter-cluster networking solutions, and proposing an orchestration solution to support service allocation. With these improvements, BlazeAIoT has the potential to serve as a foundation for distributed robotics and other distributed systems, such as IoT.

## CHAPTER 7

### BLAZEMAESTRO: A DIGITAL-TWIN FRAMEWORK FOR DISTRIBUTED ROBOTICS DYNAMIC ORCHESTRATION

Cédric Melançon<sup>1</sup> , Maarouf Saad<sup>1</sup> , Kuljeet Kaur<sup>1</sup> , Julien Gascon-Samson<sup>2</sup>

<sup>1</sup> Department of Electrical Engineering, École de Technologie Supérieure, Montréal, Québec, Canada

<sup>2</sup> Department of Software and IT Engineering, École de Technologie Supérieure, Montréal, Québec, Canada

Paper submitted to the journal « IEEE Transactions on Parallel and Distributed System » on April 12th, 2026

#### 7.1 Abstract

Autonomous mobile robots increasingly offload perception, localization, and navigation pipelines to heterogeneous compute resources spanning on-board computers, edge and fog nodes, and cloud servers. Managing these distributed resources for mobile robots is challenging due to dynamic wireless conditions, variable capacities, and strict end-to-end latency and energy constraints. This chapter presents BlazeMaestro, a digital-twin-driven orchestration framework that unifies a simulator with a real multi-cluster Kubernetes deployment. The simulator integrates models of network jitter, congestion, wireless mobility, and battery dynamics, while the real environment exposes live metrics and supports service placement. Within this digital twin, we formulate dynamic service placement as a constrained optimization problem with hard resource and timing constraints and soft objectives on energy consumption and end-to-end latency. We implement four orchestrators (a multiobjective genetic algorithm, a deep reinforcement learning agent, a greedy weighted heuristic, and an autoencoder-based method) that share a common constraint model and runtime metrics. Results in simulated and real scenarios with autonomous mobile robots show how these paradigms trade off energy, latency, and constraint satisfaction, and demonstrate that BlazeMaestro enables reproducible, metric-driven evaluation and transfer of orchestration strategies across simulation and deployment.

## 7.2 Introduction

Autonomous mobile robots increasingly rely on distributed compute resources across edge, fog, and cloud layers to execute perception, localization, control, and navigation pipelines in real time (Chaari *et al.* (2019)). These pipelines consist of interconnected services that can run on heterogeneous compute resources spanning on-board computers, nearby edge or fog nodes, and remote cloud servers in an Internet of Things (IoT)-edge-fog-cloud continuum (Ahmed *et al.* (2023)). The allocation of such heterogeneous resources, including Central Processing Unit (CPU), Random-Access Memory (RAM), and Graphics Processing Unit (GPU), as well as energy capacity and wireless connectivity, directly affects the timeliness and reliability of robotic perception and control loops (Muratore & Tsagarakis (2023)). Suboptimal allocation under fluctuating bandwidth and latency can significantly impact the robot's safety and autonomy, especially when key functions such as localization or obstacle avoidance are offloaded to remote nodes (Dordevic *et al.* (2023)). Managing these heterogeneous resources is therefore challenging due to dynamic wireless conditions, variable background load, and battery-powered robots operating under strict timing and energy constraints (Muratore & Tsagarakis (2023)).

Service placement and orchestration in edge-to-cloud (E2C) infrastructures have been widely studied using heuristic, metaheuristic, and optimization-based approaches, providing important building blocks for robotic workloads. Surveys and conceptual frameworks for the cloud-to-thing continuum highlight containerization, microservices, and service migration as key enablers of scalable orchestration across heterogeneous devices, edge nodes, and clouds as described in Ahmed *et al.* (2023); Cilic *et al.* (2021). Several works propose cooperative fog architectures and locality-aware schedulers that balance workloads between fog and cloud layers while respecting data locality and pub/sub patterns (Khezri *et al.* (2024); Kapsalis *et al.* (2017); Happ *et al.* (2021)), while complementary efforts focus on cost-optimized offloading between edge data centers and the cloud under data protection and regulatory constraints (Mann *et al.* (2023)). These studies establish the importance of locality, cost, and regulatory aspects in edge–cloud orchestration, but typically abstract away mobility, wireless variability, and battery-aware behavior that are central in autonomous mobile robotics.

Within this broader E2C context, metaheuristic-based schedulers have been used to address multiobjective placement problems in fog environments. Evolutionary approaches such as the Non-dominated Sorting Genetic Algorithm II (NSGA-II), introduced in Deb *et al.* (2002), have been evaluated for service placement optimization in fog architectures, demonstrating good solution quality across objectives such as latency, cost, and resource usage (Guerrero *et al.* (2019); Klimenko & Barinov (2023)). Other population-based techniques, including particle swarm optimization and ant colony optimization, have been applied to task scheduling in fog–cloud IoT scenarios, improving delay and energy metrics compared with baseline heuristics (Soe *et al.* (2025)). Another important aspect of microservice architecture is service composition, which induces transitive dependency latency and is addressed in Li *et al.* (2023). However, these techniques often rely on simplified constraint models that do not fully capture deadline or period requirements or explicit battery and mobility effects that arise when services are tightly coupled to moving robots.

In parallel, Deep Reinforcement Learning (DRL) has emerged as a powerful tool for dynamic resource allocation and microservice scheduling in E2C environments, offering policies that can adapt online to varying conditions. One example of a DRL-based scheduler is proposed in Jiang *et al.* (2025), modeling microservice placement on Kubernetes clusters as Markov decision processes, in which the learning policies adapt to time-varying workloads and heterogeneous resource capacities. Multi-objective DRL algorithms for mobile IoT-enabled edge computing optimize device latency and energy consumption (Khan *et al.* (2025)) or spatio-temporal latency under mobility and varying network conditions (Khoshvaght *et al.* (2025)), and prediction-enhanced approaches refine offloading decisions in IoT and fog scenarios (Alhameed (2024)). Another AI model is explored in Tian *et al.* (2026), which proposes a multiobjective scheduler to address this concern for resource optimization on a wearable device, but it has limited validation in dynamic real-world scenarios. These techniques illustrate the potential of DRL and predictive models to continuously adapt placement decisions, but they generally operate on abstract workloads or generic microservices, seldom include detailed models of wireless

handovers, jitter, or battery discharge, and rarely enforce a rich set of constraints that couple resource capacities, shared GPU memory, and end-to-end timing constraints.

When these orchestration ideas are applied to robotics, additional requirements arise due to mobility, safety, and energy constraints on robots. Hybrid cloud and fog architectures have been proposed to offload computation from robots to remote servers, sometimes combined with QoS-aware or resource-aware strategies that aim to preserve real-time control performance (Muratore & Tsagarakis (2023); Zhu *et al.* (2024)). Work on autonomous vehicles similarly studies resource management and task offloading in fog computing, with stringent latency and reliability requirements that mirror those of mobile robots (Zhou *et al.* (2020); Qingya *et al.* (2025)). At the same time, Kubernetes-based orchestration and mobility-aware scheduling have been explored in distributed systems that are closely related to robotics. Kubernetes has been used to orchestrate mixed-criticality software for autonomous mobile robots and modular edge robotics (Lumpp *et al.* (2024); Roy *et al.* (2022a)). Generalized orchestration has been integrated directly into Kubernetes distributions (Aslan *et al.* (2025)), and mobility-aware schedulers explicitly incorporate network conditions and microservice dependencies to perform dynamic, network-aware scheduling and descheduling in clusters with mobile nodes (Tsokov & Kostadinov (2025)). Complementary work on runtime orchestration of distributed control systems combines topology and orchestration specifications with Kubernetes and GitOp (Linsbauer *et al.* (2024)), and edge-mesh architectures with distributed intelligence provide building blocks for decentralized orchestration (Sahni *et al.* (2017)). These efforts underscore the feasibility of Kubernetes-based orchestration for latency-sensitive and mobility-affected workloads, but they are not specifically tailored to autonomous robotics scenarios that explicitly model energy and node mobility.

Digital twin architectures, on the other hand, have emerged as a promising approach to bridge the gap between simulation and deployment by coupling system models with their real-world counterparts. In edge and cloud contexts, digital twins and related emulation frameworks have been used to support E2C orchestration in distributed production lines, study the effect of orchestration policies on QoS, and enable impact analysis under different workloads and

network conditions (Nie *et al.* (2023); Akbari *et al.* (2022)). These approaches model robots and production lines and evaluate scheduling policies before deployment, reducing the risk of testing new policies on physical systems. However, to the best of our knowledge, no prior work provides a robotics-focused digital twin that simultaneously models network jitter, congestion, WiFi mobility, and battery dynamics for autonomous mobile robots and exposes a unified interface to both simulated and real Kubernetes-based deployments.

Most existing orchestration and scheduling frameworks for fog, edge, and cloud robotics are evaluated only in simulation or on emulated workloads, which can hide important effects of real-world wireless conditions, shared infrastructure, and robot energy constraints. In contrast, our work is explicitly designed to support both simulations and real Kubernetes-based deployments with physical robots, enabling us to validate whether scheduling policies that perform well in simulation remain effective under real network and energy dynamics.

This chapter introduces BlazeMaestro, a digital-twin-driven orchestration framework for autonomous mobile robots that unifies a complete simulator with a real Kubernetes-based multi-layer deployment via a shared orchestration facade. On the simulation side, BlazeMaestro combines network modeling with a discrete-event engine and incorporates network jitter, congestion, battery-aware energy consumption, and WiFi mobility models tailored to mobile robots, exposing orchestration algorithms to realistic wireless and energy dynamics. On the real side, it integrates with the BlazeAIoT platform (Melancon *et al.* (2026)), which comprises multiple Kubernetes clusters with Prometheus-based monitoring, and exposes the same high-level interface for node, service, latency, and metric queries in both environments. Within this digital twin, dynamic service placement is formulated as a constrained optimization problem governed by hard constraints on CPU, RAM, GPU, deadlines, and placement locks, and soft objectives that minimize energy consumption, resource usage, and end-to-end latency across service graphs.

The main contributions of this work are the following:

1. We present BlazeMaestro, a digital-twin architecture for distributed robotics that tightly couples a high-fidelity simulator with a real, multi-cluster Kubernetes deployment via a unified orchestration interface, together with a comprehensive constraint and energy model that captures mobility-induced wireless variability, transitive dependency latency, shared GPU memory, and battery dynamics on mobile robots.
2. We design and implement four constraint-aware, metric-driven schedulers (evolutionary NSGA-II, deep reinforcement learning, greedy weighted heuristic, and autoencoder-based representation learning) that all operate on this common digital twin and constraint model, enabling a fair comparison across paradigms.
3. We conduct an extensive simulation-based evaluation on realistic autonomous mobile robot workloads across heterogeneous edge–fog–cloud infrastructures, quantifying how each scheduler trades off energy consumption, end-to-end latency, and constraint satisfaction under varying network, mobility, and load conditions.
4. We deploy BlazeMaestro and the four schedulers on a real Kubernetes-based multi-cluster testbed with mobile robots, demonstrating that models and policies trained and tuned in simulation transfer to physical deployments, and revealing discrepancies between purely simulation-based results and behavior under real wireless and energy dynamics.

Overall, our aim is to compare how these four paradigms behave under a common constraint model and metric set, providing design guidance for future orchestration frameworks. The rest of the chapter is organized as follows. Section 7.3 details the digital twin implementation with the simulation and real environments. Section 7.4 presents the four orchestrators and the constraint model. Section 7.5 describes the evaluation methodology and results, followed by a discussion in Section 7.6.

### **7.3 Environments**

This section describes the simulation and real environments used by BlazeMaestro and explains how they are integrated into a unified digital twin.

### 7.3.1 Digital Twin Overview

BlazeMaestro couples a complete multi-cluster simulator with the BlazeAIoT Kubernetes-based deployment through a shared orchestration facade. Fig. 7.1 illustrates the digital twin architecture, where the orchestrators interact with a single abstraction layer that can attach to either the simulator or the real clusters, exposing the same node, service, and metric interfaces in both cases.

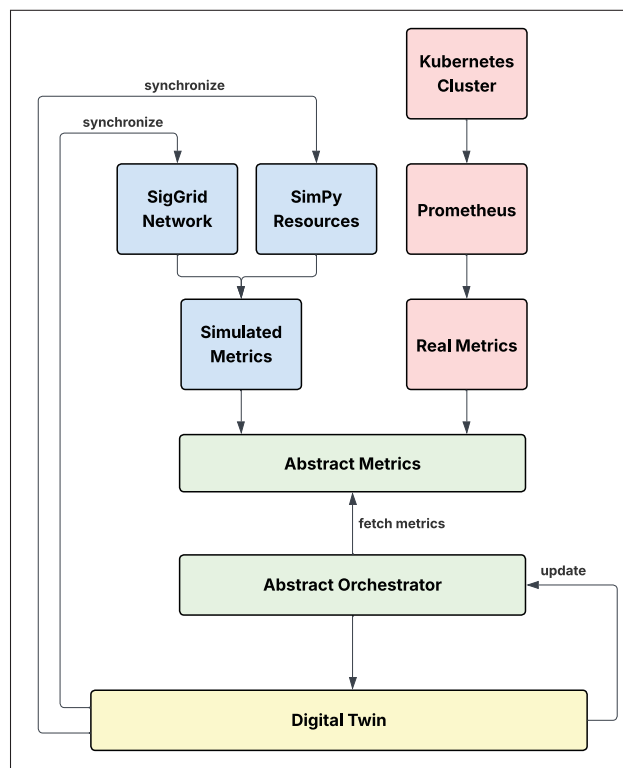


Figure 7.1 BlazeMaestro digital twin architecture

The digital twin concept provides a unified framework for integrating the simulation environment with real-world deployment and validation. Its core motivation is to enable seamless transfer of orchestration algorithms between simulated and production environments. This approach allows iterative training, validation, and refinement of orchestration strategies using both simulated and real infrastructure data.

To enable this closed loop between the simulation and the real environment, BlazeMaestro provides a unified abstraction layer that exposes the same node, service, latency, and metric interfaces to both the simulator and BlazeAIoT. This makes orchestrators environment-agnostic and allows policies, models, and constraints to be transferred from simulation to real clusters, while real metrics feed back into the simulator to refine its accuracy.

### 7.3.2 Simulation Environment

The simulator reproduces key factors that affect autonomous mobile robots across the E2C continuum, including heterogeneous resources, network variability, battery dynamics, and WiFi mobility. It combines SimGrid<sup>1</sup> to model the network topology, node-to-node routing computations, and network link characteristics (wireless or wired, latency, bandwidth), with SimPy<sup>2</sup> to model CPU, RAM, and GPU resources and to schedule service execution and contention. A single-framework implementation is technically feasible, but any such choice would either reduce network realism or significantly increase modeling effort and maintenance cost. The combined approach yields higher fidelity-to-effort in dynamic robotics orchestration experiments. Together, these components enable the replay of complex service graphs for an autonomous mobile robot scenario across heterogeneous nodes and links in a controlled, time-resolved manner.

To complete the simulation environment, four missing mechanisms critical for mobile robots were added, including the network jitter, the congestion model, battery management, and WiFi mobility.

---

<sup>1</sup> <https://simgrid.org/>

<sup>2</sup> <https://simpy.readthedocs.io/en/latest/>

### 7.3.2.1 Network jitter

The jitter is modeled by perturbing the base latency  $L_{\text{base}}(i, j)$  with a random factor, as shown below.

$$L_{\text{actual}}(i, j, t) = L_{\text{base}}(i, j) \times (1 + J(i, j, t)) \quad (7.1)$$

The jitter term  $J(i, j, t)$  is defined using the following equation:

$$J(i, j, t) = m_{\text{link}} \times \sigma_j \times \mathcal{N}(0, 1) \quad (7.2)$$

where  $\sigma_j$  controls the jitter amplitude,  $m_{\text{link}}$  scales it by link type (*e.g.*, wired vs. WiFi), and  $\mathcal{N}(0, 1)$  injects zero-mean randomness so that wireless links exhibit higher variability than wired ones.

### 7.3.2.2 Network Congestion

Congestion occurs when network traffic is high, leading to increased computational load on network routers and higher latency. It is modeled by a utilization-dependent factor  $L_{\text{cong}}(\rho)$ , as given below, which approximates the characteristic response-time-versus-utilization behavior of single-server queues.

$$L_{\text{cong}}(\rho) = \begin{cases} 1.0 + 0.375\rho & \text{if } \rho \leq 0.8 \\ 1.3 + 3.5 \left( \frac{\rho - 0.8}{0.2} \right)^2 & \text{if } \rho > 0.8 \end{cases} \quad (7.3)$$

This function places the knee of the latency–utilization curve at  $\rho = 0.8$ , with a modest linear increase below the knee and a quadratic growth above it, following empirical observations of  $M/M/1$  queues (Patel (2015); Novak & Kasera (2017)). The coefficients are chosen based on Patel (2015) so that  $L_{\text{cong}}(\rho)$  is continuous and has a continuous slope at  $\rho = 0.8$  while keeping the expression simple enough for efficient simulation, yielding a realistic "knee-shaped" response without requiring full queueing-theoretic computations.

The resulting total link latency combines jitter and congestion as given in the following equation:

$$L_{\text{total}}(i, j, t) = L_{\text{actual}}(i, j, t) \times L_{\text{cong}}(\rho_{ij}(t)) \quad (7.4)$$

Eqs. (7.1)-(7.4) therefore couple orchestrator decisions, which affect link usage, with the network latency seen by services.

### 7.3.2.3 Energy

Battery-powered robot nodes are modeled with a simple, first-order energy model in which power consumption is decomposed into an idle component and a dynamic component proportional to CPU/GPU usage. The instantaneous drain over a time step is expressed below.

$$E_{\text{drain}}(t) = P_{\text{idle}} \Delta t + \text{CPU}_n(t)(P_{\text{active}} - P_{\text{idle}})\Delta t \quad (7.5)$$

Here  $P_{\text{idle}}$  is idle power,  $P_{\text{active}}$  is active power, and  $\text{CPU}_n(t)$  is the node  $n$  CPU usage at time  $t$ . Active power is derived from resource usage as follows.

$$P_{\text{active}} = P_{\text{idle}} + \alpha_r (r_{\text{CPU}} + r_{\text{GPU}}) \quad (7.6)$$

where  $r_{\text{CPU}} = \text{CPU}_s / C_{\text{CPU}}(n)$  and  $r_{\text{GPU}} = \text{GPU}_s / C_{\text{GPU}}(n)$  are the fractional CPU and GPU usage of service  $s$  on node  $n$ , and  $\alpha_r$  is a single active-power scaling factor. RAM usage is not modeled as an independent power component since it has a negligible dynamic power impact compared with the processor and accelerator. This coarse-grained model does not aim to capture detailed hardware power states, but is sufficient to expose relative differences in energy consumption between placements in our orchestration experiments.

Battery capacity  $B_t$  in Joules is updated using the following equation:

$$B_{t+\Delta t} = B_t - E_{\text{drain}}(t) \quad (7.7)$$

This allows BlazeMaestro to measure the impact of different placements on robot battery life.

#### 7.3.2.4 WiFi Mobility

The WiFi mobility is modeled by updating the robot's positions and computing the signal strength to each access point using the standard log-distance path-loss model (Rappaport (2002)). Signal strength  $S$  in dBm is computed using Eq. (7.8).

$$S = S_{\max} - 10\beta \log_{10}\left(\frac{d}{d_{\max}}\right) \quad (7.8)$$

where  $d$  is the distance to the access point,  $d_{\max}$  is the maximum coverage radius,  $S_{\max}$  is the reference signal level at the coverage boundary  $d_{\max}$ , and  $\beta$  is the path-loss exponent representing the rate at which the path loss increases with the distance. Another essential factor to consider is the Signal-to-Noise Ratio (SNR), which affects network quality and, consequently, latency. Signal strength is mapped to discrete SNR levels that determine the effective bandwidth and latency on the wireless link (*e.g.*, high bandwidth and low latency for good SNR, reduced bandwidth and higher latency for poor SNR). At each step, the robot associates with the access point with the highest signal strength. When the best access point changes or the signal quality degrades, a handover is triggered, and the link parameters are updated.

As mobile robots move through the environment, the simulator dynamically evaluates the signal strength between each robot and all available WiFi access points. Algorithm 7.1 represents this logic which is executed at each simulation step.

This mechanism enables the study of how orchestration decisions behave under mobility-induced handovers and connectivity degradations typical of autonomous mobile robots.

#### 7.3.3 Real Environment

The real environment is based on the BlazeAIoT distributed platform, composed of multiple Kubernetes clusters spanning robot, edge, fog, and cloud nodes Melancon *et al.* (2026). Clusters

## Algorithm 7.1 WiFi Handover for Mobile Robot

```

Input: Robot position  $p_r$ , access points  $A = \{a_1, \dots, a_m\}$  with positions  $p_a$ 
Output: Current AP  $a_{\text{cur}}$ , SNR level  $\text{SNR}_{\text{cur}}$ 
1 foreach  $a \in A$  do
2    $d(a) \leftarrow \|p_r - p_a\|$ 
3    $S(a) \leftarrow S_{\text{max}} - 10\beta \log_{10}(d(a)/d_{\text{max}})$ 
4 end foreach
5  $a^* \leftarrow \arg \max_{a \in A} S(a)$ 
6  $\text{SNR}^* \leftarrow \text{MapToSNRLevel}(S(a^*))$  // good/medium/poor
7 if  $a^* \neq a_{\text{cur}}$  then
8    $a_{\text{cur}} \leftarrow a^*$  // handover to a new AP
9 end if
10 if  $\text{SNR}^* \neq \text{SNR}_{\text{cur}}$  then
11    $\text{SNR}_{\text{cur}} \leftarrow \text{SNR}^*$  // update link quality for current AP
12 end if

```

expose Prometheus metrics for resource usage and network performance, and the same abstract layer as the simulator is used to query nodes, services, and latencies. Robot services are deployed as containerized microservices orchestrated by Kubernetes, and BlazeMaestro interacts with the platform via the shared orchestration abstraction layer shown in Fig. 7.1. As a result, orchestrators developed and validated in simulation can be applied directly to the real environment without changing their logic, enabling controlled experiments and deployment under comparable conditions.

The following concepts were implemented to allow dynamic orchestration:

- **Node and service discovery:** The orchestrator queries Kubernetes to enumerate available nodes and running services, supporting dynamic topologies.
- **Multi-cluster coordination:** Orchestration decisions can span multiple clusters, enabling cross-layer optimization.
- **Real-time metrics:** Integration with Prometheus provides live resource usage data (CPU, RAM, GPU, network), allowing the orchestrator to react to changing conditions.
- **Service allocation:** Orchestration actions are applied by manipulating Kubernetes deployments, scaling replicas, and updating node affinities to control service placement.

- **Pod-to-node mapping:** The system tracks which services are running on which nodes, enabling precise control and monitoring. A pod is an instance of a deployment assigned to a specific node.

This setup mirrors the simulator in terms of observability and actuation interfaces but uses real robot hardware, edge nodes, and network links, which is essential for validating that learned or optimized policies remain feasible in real-world scenarios.

## 7.4 Dynamic Orchestration

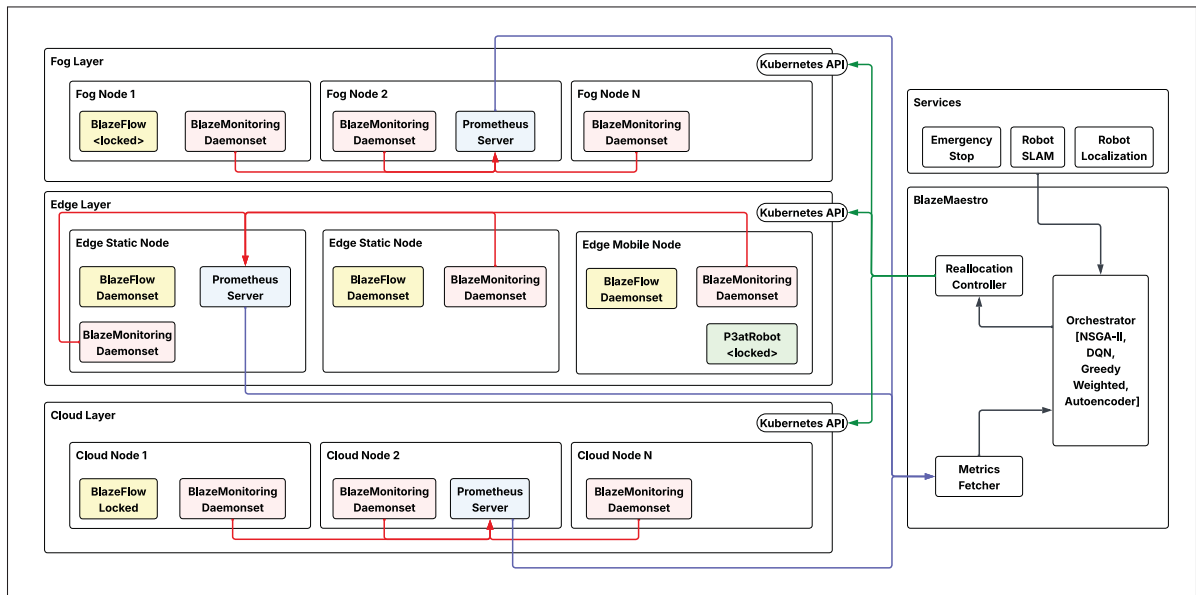


Figure 7.2 BlazeMaestro architecture within BlazeAIoT

This section presents the shared constraint model enforced by BlazeMaestro, the four orchestrators used for dynamic service placement, and the reallocation process. Fig. 7.2 represents the architecture of BlazeMaestro and how it integrates with BlazeAIoT.

### 7.4.1 Constraints

Dynamic service placement must satisfy hard constraints on resources and timing, while soft objectives guide optimization of resource usage, energy and latency. BlazeMaestro uses a

common constraint checker and objective structure shared by all orchestrators. The various constraint classes are described in Table 7.1. All four orchestrators described later use the same constraint checker and objective structure, enabling a fair comparison. Let  $S$  be the set of services and  $N$  the set of nodes (robots, edge, fog, cloud), and let  $x_{s,n} \in \{0, 1\}$  indicate whether service  $s \in S$  is placed on node  $n \in N$ . Each service must be assigned to exactly one node.

Table 7.1 Constraint Definition for Dynamic Orchestration

| Constraint                | Type      | Description                                                                     |
|---------------------------|-----------|---------------------------------------------------------------------------------|
| CPU Capacity              | Hard      | Each node has a fixed CPU capacity that cannot be exceeded                      |
| CPU Usage Factor          | Hard      | For services with periodic execution, CPU usage is adjusted                     |
| RAM Capacity              | Hard      | Total RAM usage on a node cannot exceed capacity                                |
| GPU Capacity              | Hard      | GPU memory must not exceed available capacity                                   |
| Memory Sharing Constraint | Hard      | For shared-memory nodes, the GPU memory is added to RAM requirements            |
| Period Constraint         | Hard      | Services must complete execution within their period, including network latency |
| Service Lock              | Hard      | Some services are locked to a node or a layer and cannot move                   |
| Usage Thresholds          | Objective | CPU and RAM usage should stay below thresholds                                  |
| Energy Minimization       | Objective | Minimize total energy consumption for battery-powered nodes                     |
| Latency Minimization      | Objective | Minimize total end-to-end latency                                               |

#### 7.4.1.1 CPU, RAM, and GPU Capacity

Each node has fixed CPU, RAM, and GPU capacities that cannot be exceeded. For each resource  $r \in \{\text{CPU, RAM, GPU}\}$  with service demand  $d_s^r$  and node capacity  $C_n^r$ , we enforce the resource

usage as defined below.

$$\sum_{s \in S} d_s^r x_{s,n} \leq C_n^r, \quad \forall n \in N, \forall r. \quad (7.9)$$

To reflect periodic execution, an effective CPU usage factor is expressed as follows:

$$\text{CPU}_{\text{eff}}(s) = \text{CPU}_s \times \min \left( 1.0, \frac{t_{\text{exec}}(s)}{T_{\text{period}}(s)} \right) \quad (7.10)$$

where  $t_{\text{exec}}(s)$  is the execution time and  $T_{\text{period}}(s)$  the period or deadline of service  $s$ . On platforms where the CPU and GPU share RAM, the total memory usage is the sum of the CPU and GPU memory and is capped at the node's RAM capacity.

#### 7.4.1.2 Period and Dependency Constraint

Services must meet their period constraints, including communication delays due to upstream dependencies. For a service  $s$  placed on node  $n$ , the period constraint is given by

$$t_{\text{exec}}(s) + L_{\text{dep}}(s, n) \leq T_{\text{period}}(s) \quad (7.11)$$

where  $L_{\text{dep}}(s, n)$  represents the whole service dependency latency.

The transitive dependency latency  $L_{\text{dep}}(s, n)$  is computed recursively as below.

$$L_{\text{dep}}(s, n) = \sum_{d \in D(s)} [L_{\text{edge}}(d, n) + L_{\text{dep}}(d, \text{place}(d))] \quad (7.12)$$

where  $D(s)$  is the set of direct dependencies of  $s$ ,  $L_{\text{edge}}(d, n)$  is the link latency from dependency  $d$  to node  $n$ , and  $\text{place}(d)$  is the node where  $d$  is placed. Eqs. 7.11 and 7.12 ensure that full dependency chains respect end-to-end timing constraints.

### 7.4.1.3 Service Locks and Soft Objectives

Some services are locked to specific nodes or layers to access hardware resources or satisfy visibility requirements (*e.g.*, drivers on the robot or web frontends in the cloud). Any attempt to violate these constraints is rejected, ensuring that all allocations remain consistent with them.

Soft objectives include node usage thresholds, energy minimization on battery-powered nodes, and end-to-end latency minimization. These terms appear in the orchestrators' objective functions, allowing different trade-offs while keeping the hard constraints in Eqs. (7.9) and (7.11) strictly enforced.

Period constraints require that end-to-end latency for each service (or chain) does not exceed the configured deadline  $D_s$ . Let  $L_s(x)$  be the latency induced by placement  $x$  in the digital twin, including network jitter, congestion, and WiFi mobility as defined in Eq. 7.13.

$$L_s(x) \leq D_s, \quad \forall s \in S. \quad (7.13)$$

We define the aggregate latency objective for a placement in Eq. 7.14.

$$L_{\text{tot}}(x) = \sum_{s \in S} L_s(x). \quad (7.14)$$

Placement locks and affinity/anti-affinity rules are expressed by filtering out the allowed node set for each service. For usage thresholds, we bound the utilization of each resource on each node by a maximum fraction  $\rho_{\text{max}}^r$  of its capacity as defined in Eq. 7.15.

$$\sum_{s \in S} d_s^r x_{s,n} \leq \rho_{\text{max}}^r C_n^r, \quad \forall n \in N, \forall r. \quad (7.15)$$

Given a placement  $x$ , the total energy objective  $E(x)$  sums the energy drained by all battery-powered nodes over the simulation horizon, as defined in Eq. (7.16).

$$E(x) = \sum_{n \in N} \sum_t E_{\text{drain}}^n(t \mid x) \quad (7.16)$$

where  $E_{\text{drain}}^n(t \mid x)$  is the per-step drain from Eq. (7.5) on node  $n$  under placement  $x$ . The nodes that are not battery-powered typically return 0, since they are not considered. The aggregate end-to-end latency  $L_{\text{tot}}(x)$  is defined in Eq. (7.14) and the number of hard-constraint violations  $V(x)$  counts the constraints in Table 7.1 that are not satisfied by placement  $x$ , as defined in Eq. (7.17).

$$V(x) = |\{c \in C \mid c \text{ is violated by } x\}| \quad (7.17)$$

where  $C$  is the set of all hard constraints in Table 7.1. The orchestration problem is formulated as a bi-objective minimization over feasible placements, as defined in Eq. (7.18).

$$\min_x (E(x), L_{\text{tot}}(x)) \quad \text{subject to} \quad V(x) = 0 \quad (7.18)$$

Only placements with  $V(x) = 0$  are considered feasible, so any placement that violates a hard constraint is rejected and never reported.

### 7.4.2 Orchestrator Justification

The four orchestrators implemented in BlazeMaestro are deliberately chosen to approximate the same constrained multi-objective optimization problem defined in Eq. (7.18) using complementary paradigms. The first orchestrator is a multiobjective NSGA-II algorithm inspired by evolutionary service placement methods (Deb *et al.* (2002); Guerrero *et al.* (2019); Klimenko & Barinov (2023)) directly searches for a Pareto approximation of  $(E(x), L_{\text{tot}}(x))$  within the feasible region  $V(x) = 0$ , providing a population-based baseline that explores diverse trade-offs between energy and latency. The second is a DQN-based orchestrator building on DRL schedulers for E2C microservices (Jiang *et al.* (2025); Khoshvaght *et al.* (2025); Zhang *et al.* (2023a)) that learns a

placement policy to optimize a shaped reward aligned with  $E(x)$  and  $L_{\text{tot}}(x)$ , with constraint satisfaction enforced by penalizing violations during training, offering fast inference once trained and capturing temporal structure in sequential placement decisions. The third is a Greedy Weighted heuristic method that uses multi-objective scoring to balance energy, latency, and load (Bhamare *et al.* (2017); Patel (2015)) which constructs feasible placements in a single pass using a scalar score that combines incremental energy, dependency latency, and load balance, yielding very low runtime overhead while remaining consistent with the global objectives. Finally, the fourth orchestrator is an Autoencoder-based orchestrator that learns to reconstruct high-quality placements from observed states (Zhai *et al.* (2024)) using compact latent representations, exploiting patterns in feasible, low-energy solutions and generalizing them to similar states.

By grounding all four methods in the common formulation of Eq. (7.18) and the shared constraint model in Table 7.1, BlazeMaestro enables a fair, metric-driven comparison of evolutionary, reinforcement-learning, greedy, and representation-learning approaches for dynamic orchestration of autonomous mobile robots in E2C environments.

#### 7.4.2.1 NSGA-II Orchestrator

The NSGA-II orchestrator encodes a service placement as a binary assignment vector  $x = (x_{s,n})_{s \in S, n \in N}$ , where each service  $s$  is mapped to exactly one node  $n$  according to the constraints in Table 7.1. A population of candidate placements is evolved via crossover and mutation, and each individual is evaluated according to the multi-objective formulation in Eq. (7.18).

For each placement  $x$ , the digital twin simulator computes the total energy consumption  $E(x)$  of battery-powered nodes and the aggregate end-to-end latency  $L_{\text{tot}}(x)$  across the service graph. Feasibility is enforced by treating  $V(x) = 0$  as a hard requirement, meaning that any individual with  $V(x) > 0$  is assigned to a strictly worse Pareto front than any feasible individual, so that feasible individuals always dominate infeasible ones during selection, and only placements with  $V(x) = 0$  are retained in the reported results.

The initial population is generated by sampling feasible or near-feasible placements using randomized greedy heuristics that, whenever possible, respect CPU, RAM, GPU, period, and lock constraints. Crossover operates at the service level by exchanging subsets of service–node assignments between two parents, while mutation randomly reassigns a small fraction of services to different feasible nodes. New candidate placements that violate hard constraints are repaired by reassigning offending services to alternative nodes; any remaining violations are reflected in  $V(x)$  during evaluation.

Standard NSGA-II non-dominated sorting and crowding distance are then used to select the next generation, promoting convergence toward the Pareto front of  $(E(x), L_{\text{tot}}(x))$  within the feasible region  $V(x) = 0$ . Over successive generations, the algorithm explores a diverse set of trade-offs between energy and latency under the shared constraint and objective model.

#### 7.4.2.2 DQN Orchestrator

The DQN orchestrator models dynamic service placement as a Markov Decision Process, in which each episode constructs a complete placement  $x$  by assigning services sequentially to nodes. At each decision step, the state  $o$  encodes the service to be placed, its resource and timing characteristics, current node usage, aggregated dependency latencies, and slack and criticality flags derived from the shared constraint model. The transition kernel  $P$  is implicitly defined by the environment. In simulation, the digital twin simulator updates resource usage, latencies, and battery levels in response to each placement decision and in the real deployment, it is the live Kubernetes cluster, whose observed state after each service placement action constitutes the next state  $o'$ . The action space at state  $o$  consists only of nodes that satisfy all hard constraints in Table 7.1 for the current service, so feasibility is enforced by construction at each step as defined in Eq. (7.19), where a node is feasible for service  $s$  if it satisfies all hard constraints (capacity, period, and lock) under the current state  $o$ .

$$\mathcal{A}_s = \{ n \in N \mid n \text{ is feasible for } s \} \quad (7.19)$$

The DQN policy is trained to approximate the bi-objective formulation in Eq. (7.18) by shaping the reward around  $E(x)$  and  $L_{\text{tot}}(x)$ , with constraint violations heavily penalized to enforce  $V(x) = 0$ . The total reward decomposes into a step component and a terminal component, as follows.

$$R(o, a, o') = r_{\text{step}} + r_{\text{term}} \quad (7.20)$$

The step reward  $r_{\text{step}}$  is applied after each individual service placement and combines constraint penalties, slack, and energy terms as represented in Eq. (7.21).

$$\begin{aligned} r_{\text{step}} = & -w_{\text{res}} v_{\text{res}} - w_{\text{per}} \left( \frac{\text{per}}{T_{\text{period}}} \right)^2 - w_{\text{dep}} \left( \frac{L_{\text{dep}}}{T_{\text{period}}} \right)^2 \\ & - w_{\text{slack}} - w_e E_{s_{\text{idx}}, a}. \end{aligned} \quad (7.21)$$

where  $v_{\text{res}}$  counts CPU/RAM/GPU violations,  $\text{per}$  is the period violation magnitude,  $T_{\text{period}}$  is the service period,  $L_{\text{dep}}$  is the dependency latency, and  $E_{s_{\text{idx}}, a}$  is the incremental energy cost of placing the current service on a battery-powered node, computed using the same energy model as the other orchestrators. The period term is applied only when  $\text{per} > 0$ . Otherwise, it is set to zero. The slack penalty  $w_{\text{slack}}$  is applied only if  $\text{slack} < 0.15 T_{\text{period}}$ . Otherwise, this term is omitted from  $r_{\text{step}}$ .

The terminal reward  $r_{\text{term}}$  is applied only when all services are placed, and the episode ends, as defined by Eq. (7.22).

$$r_{\text{term}} = -w_{\text{viol}} |\mathcal{V}| - w_{\text{lat}} L_{\text{crit}} \quad (7.22)$$

where  $\mathcal{V}$  is the set of remaining constraint violations (so  $|\mathcal{V}|$  plays the role of  $V(x)$ ) and  $L_{\text{crit}}$  approximates the critical-path latency through the service graph, so that lower  $L_{\text{crit}}$  correlates with lower  $L_{\text{tot}}(x)$ .

The large weight  $w_{\text{viol}}$  implements a feasibility-first hierarchy consistent with Eq. (7.18), strongly penalizing any residual violations so that learned policies prefer placements with  $V(x) = 0$ .

The Q-network  $Q_\theta(o, a)$  is trained by minimizing the standard temporal-difference target loss as defined in Eq. (7.23) and the global loss is defined in Eq. (7.24).

$$y = r + \gamma \max_{a' \in \mathcal{A}_s} Q_{\bar{\theta}}(o', a') \quad (7.23)$$

$$\mathcal{L} = \mathbb{E} \left[ (y - Q_\theta(o, a))^2 \right] \quad (7.24)$$

where  $\gamma \in (0, 1)$  is the discount factor and  $Q_{\bar{\theta}}$  is a target network periodically synchronized with  $Q_\theta$  to stabilize training. After training, the DQN orchestrator computes new placements with a single forward pass, selecting at each step the action  $a \in \mathcal{A}_s$  that maximizes  $Q_\theta(o, a)$  while the global constraint checker still enforces feasibility and evaluates  $E(x)$ ,  $L_{\text{tot}}(x)$ , and  $V(x)$  for comparison with the other orchestrators.

### 7.4.2.3 Greedy Weighted Orchestrator

The Greedy Weighted orchestrator incrementally builds a complete placement  $x$  by visiting services in topological order and, for each service  $s$ , choosing the single *best* feasible node according to a scalar score. In each step, it first filters out nodes that would violate any hard constraint under the current partial placement, then evaluates the local impact of assigning  $s$  to each remaining node in terms of energy, transitive dependency latency, and load balance, and finally selects the node that minimizes the weighted sum of these three terms. This procedure repeats until all services are placed, so that all hard constraints are satisfied by construction and the resulting placement heuristically approximates the shared multi-objective formulation in Eq. (7.18).

Formally, for each service  $s$ , the candidate node set is reduced to the feasible subset as described below.

$$\mathcal{N}_s = \{ n \in N \mid n \text{ is feasible for service } s \}. \quad (7.25)$$

where  $\mathcal{N}_s$  contains only nodes that satisfy all hard constraints (capacity, period, and lock) for service  $s$ . For each  $n \in \mathcal{N}_{\text{feasible}}$ , the orchestrator estimates the incremental energy cost  $E_{s,n}$  using the shared energy model, the transitive dependency latency  $L_{\text{deps},n}$  using Eq. (7.12), and a CPU/RAM balance term  $B_{s,n}$  that favors balanced utilization, from Eq. (7.26).

$$B_{s,n} = \left| \frac{\text{availCPU}_n}{\text{capCPU}_n} - \frac{\text{availRAM}_n}{\text{capRAM}_n} \right| \quad (7.26)$$

These quantities are combined into a weighted score evaluated with Eq. (7.27) and the node selected for service  $s$  is defined by Eq. (7.28).

$$\text{score}_{s,n} = w_E E_{s,n} + w_L L_{\text{deps},n} + w_B B_{s,n} \quad (7.27)$$

$$n^* = \underset{n \in \mathcal{N}_{\text{feasible}}}{\text{argmin}} \text{score}_{s,n} \quad (7.28)$$

Formally, the Greedy Weighted orchestrator algorithm is presented in Algorithm 7.2.

#### Algorithm 7.2 Greedy Weighted Service Placement

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                      |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------|
| <p><b>Input:</b> Service graph <math>\mathcal{S}</math>, node set <math>N</math>, current resource usage</p> <p><b>Output:</b> Placement <math>x</math> with <math>V(x) = 0</math></p> <ol style="list-style-type: none"> <li>1 Compute a topological order <math>(s_1, \dots, s_{ \mathcal{S} })</math> of the service graph</li> <li>2 Initialize empty placement <math>x</math> and resource usage counters on all <math>n \in N</math></li> <li>3 <b>foreach</b> service <math>s</math> in <math>(s_1, \dots, s_{ \mathcal{S} })</math> <b>do</b></li> <li>4     <math>\mathcal{N}_s \leftarrow \{n \in N \mid n \text{ feasible for } s\}</math></li> <li>5     <b>foreach</b> <math>n \in \mathcal{N}_s</math> <b>do</b></li> <li>6         Compute <math>E_{s,n}</math> under the shared energy model</li> <li>7         Compute <math>L_{\text{deps},n}</math> using Eq. (7.12)</li> <li>8         Compute <math>B_{s,n}</math> using Eq. (7.26)</li> <li>9         Compute <math>\text{score}_{s,n}</math> using Eq. (7.27)</li> <li>10     <b>end foreach</b></li> <li>11     Compute <math>n^*</math> using Eq. (7.28)</li> <li>12     Assign <math>s</math> to <math>n^*</math> in <math>x</math> and update resource usage of <math>n^*</math></li> <li>13 <b>end foreach</b></li> </ol> | <p>// Eq. (7.25)</p> |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------|

By iterating this process over all services in the graph, the Greedy Weighted orchestrator produces a deterministic placement that respects all hard constraints (so  $V(x) = 0$  by construction) and

seeks a low-energy, low-latency, and well-balanced configuration, while incurring only negligible computational overhead compared with NSGA-II and DQN.

#### 7.4.2.4 Autoencoder Orchestrator

The Autoencoder orchestrator learns structural patterns in feasible placements and uses them to reconstruct candidate service–node assignments under the shared constraint model. It approximates the multi-objective optimization in Eq. (7.18) by training on samples that encode the same latency and energy objectives and by using the common feasibility checker to ensure that final placements satisfy all hard constraints ( $V(x) = 0$ ).

The input state  $x$  of the autoencoder is represented below.

$$x = [f_1^{\text{svc}}, \dots, f_S^{\text{svc}}, f^{\text{net}}, f^{\text{energy}}] \quad (7.29)$$

In Eq. (7.29),  $f_s^{\text{svc}}$  is the per-service features defined as follows.

$$f_s^{\text{svc}} = [f_s^{\text{res}}, f_s^{\text{time}}, f_s^{\text{logic}}, f_s^{\text{exec}}, f_s^{\text{dep}}] \quad (7.30)$$

with

$$f_s^{\text{res}} = [\text{CPU}_s, \text{RAM}_s, \text{GPU}_s] \quad (7.31)$$

$$f_s^{\text{time}} = [t_{\text{exec},s}, T_s, T_s^{-1}, \tau_s, \rho_s] \quad (7.32)$$

$$f_s^{\text{logic}} = [c_s, \delta_s] \quad (7.33)$$

$$f_s^{\text{exec}} = (\hat{t}_{s,n})_{n \in N} \quad (7.34)$$

$$f_s^{\text{dep}} = (\ell_{s,n}^{\min}, \ell_{s,n}^{\max}, \bar{\ell}_{s,n})_{n \in N} \quad (7.35)$$

Here  $\hat{t}_{s,n}$  is the estimated execution time of service  $s$  on node  $n$ ,  $c_s$  is a binary real-time criticality flag  $c_s$  indicating if the service has a specific period  $T_{\text{period}}(s)$  and that its execution time  $t_{\text{exec}}(s)$  fits in this period.  $\delta_s$  is a binary dependency indicator indicating if the service  $s$  has at least one

dependency, and  $\ell_{s,n}^{\min}/\ell_{s,n}^{\max}/\bar{\ell}_{s,n}$  are the minimum, maximum, and mean dependency latency for service  $s$  at node  $n$ .

The term  $f^{\text{net}}$  of Eq. (7.29), is the aggregated network features detailed below.

$$f^{\text{net}} = [f_{\text{tot}}^{\text{cap}}, f_{\text{avg}}^{\text{cap}}, \bar{L}, L_{\text{max}}, |N|] \quad (7.36)$$

This vector represents the total capacity of resources (CPU/RAM/GPU) across all nodes, as given in Eq. (7.37), aggregated with the average resource capacity per node, defined in Eq. (7.38), the average and maximum inter-node latency, and the node count.

$$f_{\text{tot}}^{\text{cap}} = [C_{\text{tot}}, R_{\text{tot}}, G_{\text{tot}}] \quad (7.37)$$

$$f_{\text{avg}}^{\text{cap}} = [\bar{C}, \bar{R}, \bar{G}] \quad (7.38)$$

Finally, the term  $f^{\text{energy}}$  in Eq. (7.29) represents the per-node energy features as defined by Eq. (7.39).

$$f^{\text{energy}} = [b_{n_j}, p_{n_j}]_{j=1}^{|N|}. \quad (7.39)$$

where  $b_{n_j}$  is a flag indicating whether the node is battery-powered, and  $p_{n_j}$  is the normalized active power of the node  $n_j$ .

The encoder and decoder share a single network that processes each service independently. The augmented feature vector  $x_s$  for service  $s$  (combining local service attributes with the shared network and energy statistics) is encoded to a per-service latent vector  $z_s$ , and the decoder maps  $z_s$  to a logit vector  $U_s$  over candidate nodes:

$$z_s = f_{\text{enc}}(x_s) \quad s = 1, \dots, S \quad (7.40)$$

$$U_s = f_{\text{dec}}(z_s) \quad (7.41)$$

where  $U_{s,n}$  is the unnormalized logit for placing service  $s$  on node  $n$ . Training samples are generated by random feasible placement exploration: services are assigned in topological order

to randomly selected feasible nodes, with biased sampling that favours nodes offering greater period-slack margin for real-time-constrained services. Each sample  $q$  pairs an input state  $x$  with the resulting one-hot assignment  $y$ . Samples are weighted by an energy efficiency factor defined in Eq. (7.42), combined with a penalty for constraint violations, so that energy-efficient and feasible placements exert greater influence during training. the sample weight is assigned as follows:

$$\alpha_q = \frac{1}{1 + 0.1 E_q} \quad (7.42)$$

where  $E_q$  represents the total estimated energy cost of sample  $q$ .

$$w_q = \begin{cases} \max(0.1, \alpha_q) & V(x) = 0 \\ \max\left(0.05, \frac{\alpha_q}{1 + 5 V(x)}\right) & V(x) > 0 \end{cases} \quad (7.43)$$

where  $V(x)$  is the number of hard-constraint violations in the sample.

The autoencoder loss, which combines weighted cross-entropy reconstruction and latent regularization, can be defined as below.

$$\mathcal{L}_{\text{AE}} = \mathbb{E}[w_q \mathcal{H}(U_s, y_s)] + \lambda \mathbb{E}_s[\|z_s\|_2^2] \quad (7.44)$$

where  $y_s \in \{0, 1\}^{|N|}$  denotes the one-hot target node assignment for service  $s$ . Then,  $\mathcal{H}(U, y)$  is the mean per-service cross-entropy loss between each decoder output  $U_s$  and the corresponding one-hot ground-truth assignment  $y_s$ . Cross-entropy is chosen because the placement output is a multinomial classification (one node per service), and the  $\ell_2$  regularization is averaged over all per-service latent vectors  $z_s$ .

At inference time, the decoder outputs per-service logit vectors  $U_s$ , which are converted to probability distributions over nodes via temperature-scaled softmax. Services are processed in topological order so that dependency latency can be accounted for during feasibility checking.

For each service  $s$ , all nodes that satisfy the hard constraints in Table 7.1 under the current accumulated resource usage are first identified. Among these feasible candidates, the node with the highest softmax probability is selected. This post-processing ensures that the final placement obeys the same feasibility conditions as the other orchestrators, so that  $V(x) = 0$  for all accepted configurations, while the autoencoder exploits learned structure to approximate low-energy, low-latency regions of the multi-objective space defined by Eq. (7.18) with relatively low runtime overhead.

### 7.4.3 Reallocation Process

The reallocation process is the final stage of each BlazeMaestro cycle, which translates orchestration decisions into safe Kubernetes placement updates. At each iteration, BlazeMaestro receives a candidate placement map from one of the orchestrators, compares it with the current Kubernetes deployment, and applies only the necessary differences as concrete updates to the running system. To avoid oscillations caused by small fluctuations in latency or energy, the controller applies a hysteresis mechanism that rejects placement changes whose improvement is below a user-defined threshold on the objective values.

More concretely, each control cycle performs the following steps to safely realize a new placement on Kubernetes while preserving application continuity:

1. **Change detection and planning:** The controller retrieves the current placement from the service discovery layer and compares it with the target placement, constructing a minimal migration set that contains only services whose node or layer assignment changes. Services whose targets are unchanged are left untouched, and services with fixed placement policies (e.g., DaemonSets or node-locked services) are excluded from the migration.
2. **Applying Kubernetes updates:** For each migration in the set, the controller patches the corresponding Deployment or StatefulSet with updated node constraints (e.g., `nodeSelector`, `nodeName`, or `nodeAffinity`), letting Kubernetes perform a rolling update that terminates old pods and schedules new pods on the selected nodes.

3. **Observation and state update:** After each patch, the controller waits for the updated pods to reach Ready status, using readiness probes to ensure that services only become active once their communication paths are established; on success, the in-memory allocation state is updated, while failed rollouts are logged and the last known good allocation is preserved.

This staged procedure ensures that high-level orchestration decisions produced by the orchestrator are applied consistently to the underlying Kubernetes platform, while limiting unnecessary restarts and avoiding unstable, rapid back-and-forth reallocations.

When a service moves between nodes in the same layer, the networking path and service dependencies typically remain within the same tier, but transient message duplication can occur during rollout. We mitigate this at the BlazeFlow (data distribution) level, where the various data protocols supported in the platform (Kafka, Redis Pub-Sub, and DDS) can use an exclusive publisher to filter out incoming messages from other publishers. ROS2 Humble distribution is used for the robot services in the evaluation, as described in Melancon *et al.* (2026). Unfortunately, the DDS version used by this distribution lacks the ownership QoS capability. Since only the control’s custom service is affected by potential message duplication, the filtering was applied directly to the ROS subscriber to process only one message within a time window defined by the expected message rate. Since all other ROS packages use sensor values from the robot, and the ROS robot node is locked to the robot, there is no possibility of message duplication.

## 7.5 Evaluation

This section describes how the four orchestrators are trained and evaluated within the BlazeMaestro digital twin and presents the resulting trade-offs among energy, latency, and stability.

### 7.5.1 Training Scenarios

Training uses a fixed platform topology and a set of scenario templates that apply background load and mobility variations. At each episode, an infrastructure scenario is combined with a service-level load pattern, yielding diverse yet repeatable conditions that cover edge cases and teach models to respond when constraints are stressed.

Infrastructure scenarios apply high CPU, RAM, or combined CPU/RAM background load on half of the nodes, stress GPU-capable nodes with intense GPU usage and moderate CPU/RAM usage, or saturate all nodes in a given layer. A mobility scenario moves robots among WiFi access points, requiring orchestrators to deal with varying signal strength, handovers, and wireless latency. Each non-WiFi scenario has a preassigned variant that preserves the initial allocations to test stability under changes in background load. Network jitter and packet loss are enabled to allow the models to learn their effects on effective latency and constraint satisfaction.

Service-level patterns modify execution frequencies and execution times. These patterns include a normal pattern using baseline rates with small random jitter, a high-rate pattern that increases frequencies by shortening periods, a heavy-execution pattern that increases execution times, and a light-load pattern that reduces frequencies to create slack. These perturbations stress period constraints and make the training distribution better match realistic robotic workloads.

### 7.5.2 Simulation Evaluation

The simulation evaluation leverages the digital twin simulator with background CPU/RAM load, network jitter, congestion, battery dynamics, and WiFi mobility, as described in Section 7.3.2. The ROS-inspired robotic application is deployed across heterogeneous simulated robot, edge, fog, and cloud nodes parameterized from the real topology. The dynamic scenario runs for 10 minutes, with orchestrators executed every 10 seconds, and is divided into six time cycles (10 samples per cycle). The workload sequence begins with a normal phase (I) for two cycles, then introduces a peak phase with higher execution rates (II), a phase with heavier execution times (III) to emulate degraded compute capacity, a congestion phase (IV) with normal rates

and increased contention, and finally a low-traffic phase (V) with reduced rates. The mobile robot node follows a linear trajectory between access points, whose SNR levels change and may trigger handovers, thereby affecting end-to-end latency along the service graph.

Table 7.2 Simulation Test Results

| Orchestrator    | Hysteresis: 0%     |                     |                         |      | Hysteresis: 5%     |                     |                         |      |
|-----------------|--------------------|---------------------|-------------------------|------|--------------------|---------------------|-------------------------|------|
|                 | Latency<br>Avg (s) | Energy<br>Total (J) | Realloc.<br>(out of 60) | $C$  | Latency<br>Avg (s) | Energy<br>Total (J) | Realloc.<br>(out of 60) | $C$  |
| NSGA-II         | 0.1846             | 25078.70            | 59                      | 0.50 | 0.1814             | 25189.78            | 59                      | 0.50 |
| DQN             | 0.2545             | 23813.21            | 3                       | 0.50 | 0.2555             | 23813.21            | 0                       | 0.39 |
| Greedy Weighted | 0.2296             | 23813.21            | 12                      | 0.35 | 0.2769             | 23813.81            | 0                       | 0.50 |
| Autoencoder     | 0.2541             | 24244.79            | 30                      | 0.70 | 0.2537             | 24244.79            | 0                       | 0.47 |

Dynamic scenario performance for each orchestrator is summarized in Table 7.2. For each hysteresis setting (0% and 5%), the table reports the average end-to-end latency, total energy consumption on battery-powered nodes, the number of service reallocations over the 60 decisions, and an aggregate cost  $C$  that combines normalized latency, energy, and reallocation counts. The aggregate cost is defined as follows:

$$C = w_L \hat{L} + w_E \hat{E} + w_R \hat{R} \quad (7.45)$$

where  $\hat{L}$ ,  $\hat{E}$ , and  $\hat{R}$  denote the average latency, total energy, and reallocation count, each min-max normalized to  $[0, 1]$  across the four orchestrators for each hysteresis setting. In other words, the normalization uses the minimum and maximum values observed within a given experiment, so that  $C$  compares solutions relative to one another for that specific run rather than providing an absolute performance value that can be compared across experiments with different metric ranges. In our experiments, we use  $w_L = 0.5$ ,  $w_E = 0.3$ , and  $w_R = 0.2$  to reflect an operating regime where latency is the primary concern, while energy usage and reallocation stability remain secondary but still important factors, which is typical of real-time robotic applications where timing constraints are primary but battery life and excessive reallocations cannot be ignored. These weights are not part of the optimization formulation itself. They are used only as

an illustrative aggregation to ease comparison between schedulers for a representative set of application requirements.

At 0% hysteresis, Greedy Weighted achieves the lowest aggregate cost  $C$  (0.35), limiting both energy usage and reallocation frequency despite moderate latency. NSGA-II and DQN are tied at  $C = 0.50$  for opposing reasons. NSGA-II achieves the lowest latency (0.1846 s) but incurs the highest energy and reallocation penalties (59/60), while DQN eliminates reallocation overhead and keeps energy near the minimum, yet produces the highest average latency. The Autoencoder incurs the worst aggregate cost (0.70), combining near-maximum latency with elevated energy consumption and frequent reassignments (30/60). At 5% hysteresis, DQN and Autoencoder reduce their cost to 0.39 and 0.47, respectively, by eliminating reallocations, while NSGA-II remains at 0.50, confirming that hysteresis has a limited impact on evolutionary search behavior. Greedy Weighted's cost rises from 0.35 to 0.50 at 5% hysteresis because suppressing reallocation locks it into an aggressive offloading placement that, while energy-efficient, yields the highest latency of the group.

Fig. 7.3 provides a time-resolved view of latency and energy over the six phases of the scenario. NSGA-II maintains consistently low latency across phases, but at the cost of systematically higher energy usage on the robot node, especially during periods with increased computational demand and contention (II-IV). In contrast, DQN and Greedy Weighted, which are superposed on the figure, exhibit more pronounced latency fluctuations when the scenario becomes more demanding (II-III), yet keep energy consumption closer to the minimum by offloading more aggressively to non-robot nodes. The Autoencoder lies between these extremes, with moderate sensitivity to phase changes in both latency and energy, illustrating how different scheduling paradigms trade off robustness to load/mobility variations versus energy efficiency.

### 7.5.3 Experimental Evaluation

To validate transfer from simulation to deployment, we evaluate the orchestrators on the BlazeAIoT distributed platform, which provides real robot, edge, fog, and cloud nodes organized

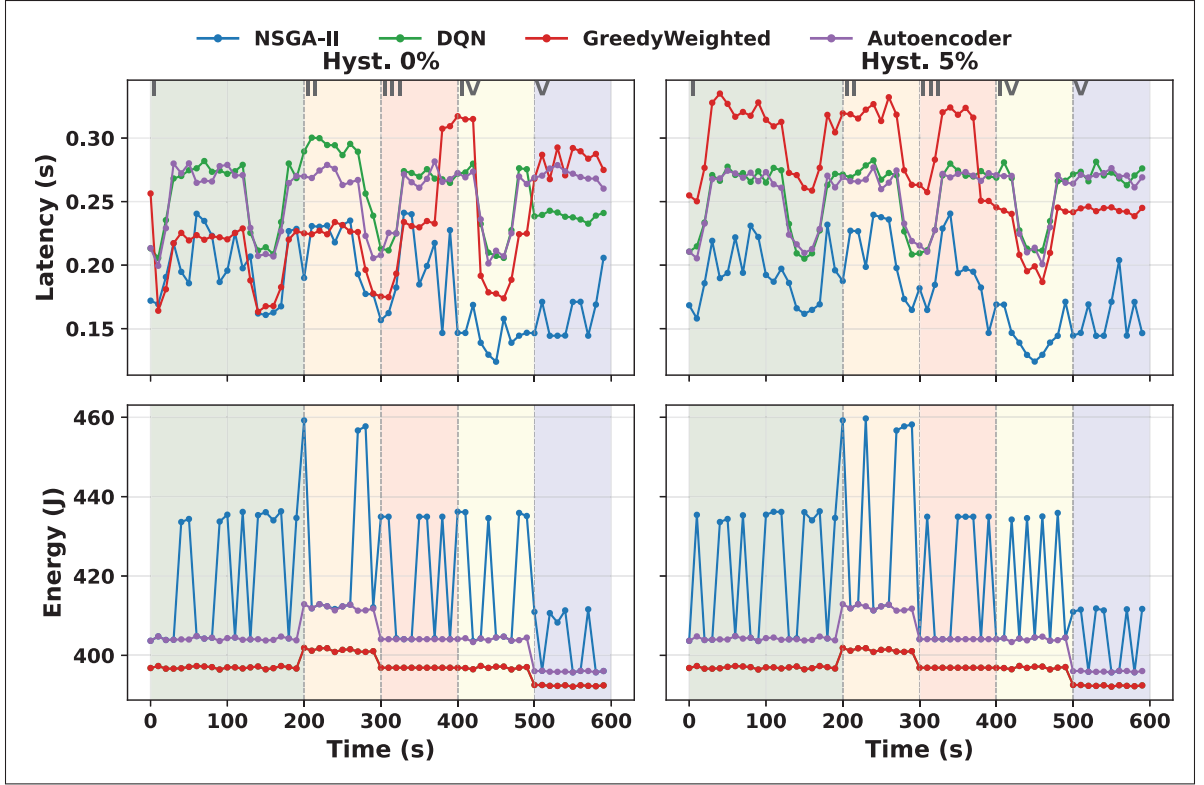


Figure 7.3 Simulation latency and energy over scenarios I-V

as multiple Kubernetes clusters with Prometheus-based monitoring. The goal of this experiment is not to obtain a one-to-one numerical match with the simulation results, but rather to assess whether the qualitative behaviors observed in the digital twin carry over to a real Kubernetes-based testbed with physical robots. We therefore focus on relative trends across orchestrators (*e.g.*, latency–energy trade-offs, reallocation patterns, and computational overhead) under controlled but realistic network perturbations, rather than on reproducing identical absolute latency and energy values. This allows us to validate that the scheduling policies remain effective and feasible when exposed to real wireless links, shared infrastructure effects, and hardware execution times. The same service graph is used on the real-robot infrastructure, and orchestrators interact with the real environment via the unified digital-twin abstract layer.

All orchestrator models were executed in parallel with the same input values to enable direct comparison under identical conditions. A hysteresis value of 0% was used to isolate base

placement behavior without any stabilization suppression. Since the physical lab environment is not large enough to produce natural WiFi signal degradation, the effect was emulated by injecting increasing latency and packet loss (0 to 70 ms artificial added latency, 0 to 3% packet loss) over the first part of the experiment, followed by a gradual recovery. To evaluate stability, all services except the navigation service were pre-deployed for approximately five minutes, after which the robot began its path-following operation for up to two minutes before the scenario ended, for a total experiment duration of approximately ten minutes.

Table 7.3 Experimental Test Results (Hysteresis: 0%)

| <b>Orchestrator</b> | <b>Latency<br/>Avg (s)</b> | <b>Energy<br/>Total (J)</b> | <b>Realloc.<br/>(out of 60)</b> | <b>C</b> | <b>Decision Time<br/>(s)</b> |
|---------------------|----------------------------|-----------------------------|---------------------------------|----------|------------------------------|
| NSGA-II             | 0.1163                     | 17676                       | 31                              | 0.20     | 0.604                        |
| DQN                 | 0.1171                     | 17676                       | 0                               | 0.01     | 0.011                        |
| Greedy Weighted     | 0.1366                     | 17676                       | 2                               | 0.30     | 0.001                        |
| Autoencoder         | 0.1519                     | 17676                       | 5                               | 0.53     | 0.003                        |

The results are summarized in Table 7.3. The relative stability ranking from the simulation is preserved. The DQN produces the fewest placement changes (0/60), followed by Greedy Weighted (2/60), Autoencoder (5/60), and NSGA-II (31/60). This consistency across environments confirms that the stability characteristics measured in the digital twin transfer to the real Kubernetes deployment. All orchestrators satisfy constraints throughout the ten-minute experiment with zero failures, validating that the constraint enforcement layer operates correctly across the simulation-to-deployment boundary.

Examining individual behaviors provides further insight. The Autoencoder performs five reallocations on the robot localization service, progressively repositioning it from the edge toward fog nodes before settling, and makes additional intermediate adjustments in the fog layer. Greedy Weighted also reassigns localization twice, cycling through the available Fog GPU nodes until finding a stable assignment, after which all services converge to the fog layer. NSGA-II exhibits the least stable behavior with 31 reallocations, and alternates repeatedly between two fog nodes for the localization service, generating a ping-pong pattern. This behavior indicates that

the Pareto front continues to shift under real-world metric noise, producing marginal competing solutions that alternate dominance. The DQN, by contrast, commits to a single placement decision from the beginning and never deviates, suggesting that its learned value function generalizes well to the real observation space.

Execution time differences are significant and carry direct operational implications. Greedy Weighted is the fastest at 1.2 ms, followed by Autoencoder at 3.4 ms. They both sit well within real-time decision budgets, even for rapid control loops. DQN requires 11.41 ms per decision, which remains acceptable for orchestration-level tasks. NSGA-II, however, averages 603.6 ms, which is nearly 500 times slower than Greedy Weighted and roughly 50 times slower than DQN. At this latency, NSGA-II cannot react quickly enough to transient events such as WiFi handovers or sudden spikes in resource contention, limiting its suitability for scenarios where the orchestration horizon is measured in seconds rather than milliseconds.

The energy consumption across the ten-minute experiment is represented in Fig. 7.4 for the DQN orchestrator. During the experimental test, none of the four schedulers assigns services to the robot node itself, so all of them induce essentially identical compute and energy usage on the robot, which is why only the DQN is represented. The first 310 seconds correspond to the energy consumed while the robot is at rest. In the range of 310-510 s, the motors are running, increasing energy consumption. Since all allocable services are pushed outside the robot's node, the energy curve represents the best-case scenario for battery autonomy.

## 7.6 Discussion

The results indicate that no single orchestrator dominates across all conditions and that the choice of strategy should reflect application priorities and system dynamics. In scenarios with relatively stable network conditions and moderate mobility, NSGA-II consistently finds high-quality latency–energy trade-offs, but at the cost of higher computational overhead and slower reaction time. As network jitter, congestion, or mobility-induced handovers become more frequent, the DQN-based orchestrator adapts more rapidly to changing conditions, yielding lower average

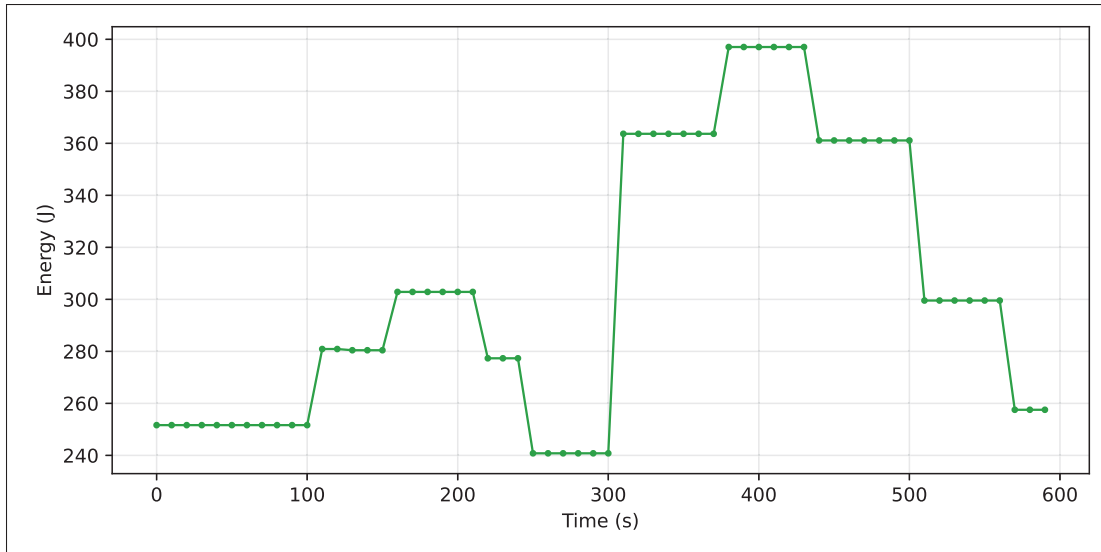


Figure 7.4 Experimental instantaneous energy consumption

latency and improved constraint satisfaction under tight timing constraints. Greedy Weighted and Autoencoder incur lower runtime overhead, making them attractive for resource-constrained deployments or as fallbacks when learning-based methods cannot be retrained online, but they offer less flexibility when the workload or infrastructure deviates from the patterns observed at design time.

BlazeMaestro’s digital-twin design brings two main benefits. By incorporating models of network jitter, congestion, WiFi mobility, and battery discharge, the simulator exposes orchestrators to behaviors representative of real robots traversing heterogeneous wireless zones with limited energy, leading to placement policies that are more robust than those trained on stationary or purely microservice-based models. At the same time, the unified interface across the simulator and the real Kubernetes deployment enables seamless transfer, allowing the same orchestration logic, state representation, and constraint evaluation to be reused without rewriting the policy for a new environment. In practice, this tight coupling shortens the iteration loop (policies are trained and stress-tested in simulation, and only promising candidates are deployed to real clusters, where live metrics can refine the models and provide realistic performance and safety feedback).

BlazeMaestro still approximates several phenomena, such as detailed CPU/GPU frequency scaling, temperature effects, and low-level network contention. The WiFi mobility model captures handovers and coverage variations at coarse granularity but does not represent all physical-layer effects or multi-radio configurations. Our evaluation also targets a specific class of autonomous mobile robots and a particular Kubernetes-based infrastructure. Different robot platforms, accelerators, or network technologies (*e.g.*, 5G) may exhibit behaviors only partially captured by the current models, and large-scale fleet behavior with multiple robots and mixed-criticality workloads remains only partially explored. Finally, BlazeMaestro’s centralized design is vulnerable to connectivity loss; combining it with a local, safety-first fallback orchestrator on the robot would provide an offline degraded mode that guarantees safe operation.

Although BlazeMaestro is designed for autonomous mobile robots, several lessons extend to other latency-sensitive E2C applications. The constrained optimization formulation that separates hard constraints (resources, deadlines, placement locks) from soft objectives (resource usage, latency, energy) applies to distributed systems where safety or QoS must not be violated while cost and efficiency remain important. The unified orchestration interface across simulated and real environments also suggests a general pattern for constructing digital twins that can host heterogeneous orchestrators with consistent instrumentation and constraint validation. More broadly, the comparative evaluation of NSGA-II, DQN, Greedy Weighted, and Autoencoder orchestrators underscores the importance of matching orchestration paradigms to workload structure and variability: predictable, stable demand may favor algorithmic optimization, whereas highly dynamic or partially observable environments may benefit more from learning-based or hybrid strategies.

Our evaluation pipeline relies on a staged process in which the simulation environment acts as a fast pre-filter before real-robot experiments. This keeps the cost and complexity of physical deployments manageable, but also means that only a subset of configurations observed in simulation is tested on the real infrastructure. Consequently, our real-world results characterize the most promising candidates under realistic wireless and hardware conditions, rather than exhaustively covering the entire configuration space explored in the digital twin.

## 7.7 Conclusion

This chapter presented BlazeMaestro, a digital-twin-driven orchestration framework that couples a complete simulator with a real multi-cluster Kubernetes deployment for autonomous mobile robots via a unified orchestration facade. Dynamic service placement is formulated as a constrained optimization problem with hard resource, timing, and placement constraints and soft objectives on energy consumption and end-to-end latency, and four orchestrators (multiobjective NSGA-II, DQN-based deep reinforcement learning, greedy weighted heuristic, and autoencoder-based method) are implemented on a common constraint model and metric interface. Evaluation on scenarios with mobility-induced wireless variability and battery dynamics shows that these paradigms exhibit distinct trade-offs among energy, latency, and constraint satisfaction, and that the digital twin enables consistent characterization of these trade-offs across simulated and real Kubernetes-based infrastructures. By integrating robotics-aware models of network jitter, congestion, mobility, and battery behavior into a cloud-agnostic stack, BlazeMaestro improves the fidelity and transferability of orchestration policies and is applicable to other latency-sensitive edge-to-cloud applications beyond robotics. Future work includes enriching the constraint and energy models, and extending the framework to multi-robot coordination and fleet-level objectives.

## CONCLUSION AND RECOMMENDATIONS

This thesis studied how to operationalize autonomous mobile robots over heterogeneous edge-fog-cloud infrastructures while satisfying strict real-time and energy constraints. Building on a modernized mobile platform and a multi-layer testbed, the work addressed three tightly coupled challenges: low-latency data distribution, real-time DL localization, and dynamic orchestration of distributed robotic services. Together, these contributions form an integrated, IoT Edge-enabled collaborative robotics framework tailored to smart healthcare and similar safety-critical domains.

### 8.1 Contributions

The first contribution of this thesis is BlazeFlow, a multi-layer communication middleware designed for real-time distributed IoT and robotic applications across edge, fog, and cloud. BlazeFlow combines topic-based publish/subscribe semantics with explicit support for inter-layer and intra-layer links, enabling flexible data distribution topologies that span robots, gateways, and servers. Experimental results demonstrated that BlazeFlow can maintain low end-to-end latency and bounded bandwidth usage under diverse traffic patterns, while preserving acceptable CPU and memory footprints on constrained nodes. These properties make BlazeFlow a suitable foundation for time-sensitive robotic services that must continuously exchange high-rate sensor data and control messages across the E2C continuum.

The second contribution is a detailed impact analysis of algorithmic and implementation optimizations on a DL visual odometry model deployed on embedded GPU platforms. By progressively refactoring the execution pipeline through ROS integration, parallelization, and loop unrolling, the work quantified how each optimization phase affects frame rate, resource usage, power consumption, and temperature. The measurements showed that carefully restructuring the model and its runtime leads to substantial improvements in execution time and energy efficiency,

without degrading localization quality. This analysis provides practical guidelines for deploying DL localization on mobile robots under real-time constraints and tight energy budgets, and it highlights the importance of joint model, software, and hardware co-design.

The third contribution is BlazeAIoT, a modular, multi-layer platform for real-time distributed robotics across edge, fog, and cloud infrastructures. BlazeAIoT integrates communication, configuration, monitoring, and service management into a unified framework that exposes a standard SDK for robotic services. The platform leverages BlazeFlow as a dynamic data bridge to flexibly route streams between layers, and it relies on a monitoring subsystem to observe resource usage and network performance in real time. Experiments with realistic navigation and emergency stop scenarios on a heterogeneous testbed showed that BlazeAIoT can satisfy latency and safety requirements while adapting data flows and service placement to infrastructure conditions. These results confirm that a tightly integrated E2C platform can support complex autonomous robot workloads in smart environments such as hospitals.

The final contribution is BlazeMaestro, a digital-twin framework for dynamic orchestration of distributed robotic services. BlazeMaestro couples a simulator with a real Kubernetes-based multi-cluster deployment through a unified orchestration abstraction, enabling schedulers to operate consistently in both environments. It models heterogeneous compute resources, wireless variability, network congestion, battery dynamics, and dependency-induced latency, and it exposes a shared constraint and objective structure that all orchestrators must satisfy. On top of this model, four constraint-aware, metric-driven schedulers were designed and implemented: a multi-objective NSGA-II algorithm, a Deep Q-Network (DQN) reinforcement learning policy, a greedy weighted heuristic, and an autoencoder-based orchestrator. Simulation and experimental results showed how each paradigm trades off energy consumption, end-to-end latency, and constraint satisfaction under varying mobility, load, and network conditions. This comparison

provides design guidance for future orchestration frameworks targeting autonomous robots in E2C environments.

Collectively, the proposed communication middleware, real-time AI optimizations, E2C platform, and digital-twin-based orchestration framework advance the state of the art in distributed robotics. They show that closing the loop between measurement, modeling, and orchestration is essential to meet the real-time and energy requirements of mobile robots that rely on DL and multi-layer infrastructures. At the same time, the observations in this thesis underline the importance of exposing clear abstractions to developers so that complex distributed deployments remain manageable as fleets and workloads scale.

## **8.2 Recommendations**

Based on the findings of this thesis, several recommendations can be formulated for the deployment of autonomous mobile robots over edge-fog-cloud infrastructures and for future research in this area. From a practical standpoint, safety-critical localization and control services should be deployed as close as possible to the robot. In concrete terms, on-board GPUs or edge nodes are preferable locations for DL-based visual odometry and low-level control loops. Higher-level analytics, logging, and decision-support tasks can be offloaded to fog and cloud layers. The experimental results obtained with the visual odometry model and the BlazeAIoT platform show that even modest changes in service placement can significantly affect end-to-end latency, energy consumption, and thermal behavior. These effects become more pronounced when multiple AI components are chained in the control pipeline. Accordingly, on-board and edge resources should be allocated primarily to services that directly support system stability, safety, and emergency response. Offloading decisions should rely on clear and explicitly defined criteria. These criteria should take into account control-loop periods, acceptable latency bounds, and the available computational and energy budgets.

Communication between robotic services should rely on a unified multi-layer middleware rather than on ad hoc bridges between protocols. A middleware such as BlazeFlow simplifies the management of heterogeneous protocols and provides consistent abstractions for intra-node, intra-layer, and inter-layer links. It also enables low-latency data paths to be configured explicitly for critical topics. In practice, high-priority data flows, such as localization, control, and emergency stop, should be clearly distinguished from bulk traffic used for monitoring, logging, and analytics. They should assign appropriate Quality of Service settings to each category of flow. Latency and jitter on critical streams should be monitored continuously. Thresholds and reconfiguration rules should be specified in advance so that the system can adapt routes or reduce data rates before quality of service degrades to unsafe levels.

For deployments that span multiple clusters, container-based orchestration platforms such as Kubernetes should be used not only for scalability but also for modularity and observability. Robotic services should be packaged with standardized interfaces for configuration, health reporting, and metrics. The platform should integrate monitoring and logging stacks that collect both infrastructure-level and robot-level metrics. This integrated observability helps operators relate anomalies observed on the robot to resource bottlenecks or network issues in the underlying infrastructure. Orchestration policies should also include safeguards for safety-critical services. For example, some services may be pinned to specific nodes or restricted to a subset of placements. Reallocation frequency should be limited, for instance through hysteresis mechanisms, to avoid oscillations in placement that could destabilize control loops.

Finally, this work suggests several methodological guidelines. Evaluations of distributed robotic systems should rely on multi-dimensional metrics rather than on single indicators. Latency, CPU and GPU utilization, memory usage, temperature, power consumption, and task-level metrics such as localization error, navigation success rate, or emergency stop responsiveness should be measured together. Considering only one or two metrics can lead to incomplete or misleading

conclusions. A combined use of digital twins and real-world experiments is also recommended. Simulation allows rapid exploration of scenarios and parameter spaces. Real deployments reveal integration issues, measurement noise, and rare events that are difficult to model, especially in wireless and human-populated environments. Iterating between simulated and real environments can increase confidence in the robustness of proposed solutions. Reproducibility and reusability should also be treated as first-class goals. Publishing containerized services, orchestration policies, configuration files, and anonymized measurement datasets would enable other researchers to validate and extend the results of this thesis and to apply similar methodologies to different robotic platforms and application domains.

### **8.3 Future Research**

From a research perspective, the results obtained with the four contributions of this thesis point to several directions. First, there is a need for tighter co-design between AI models and the distributed infrastructures that host them. Model architectures and deployment strategies are often optimized independently. Future work should instead investigate lighter and more energy-efficient DL backbones, quantization-aware training, model compression, and model partitioning schemes that explicitly target the latency, energy, and temperature budgets observed on edge devices and on-board platforms. Extending the type of detailed measurements conducted here from visual odometry to other critical tasks, such as perception, human detection, intent prediction, and navigation, would help derive broader design rules for real-time AI in robotics.

Second, the comparative analysis of NSGA-II, deep Q-learning, greedy, and autoencoder-based orchestrators indicates that no single approach dominates across all scenarios. Hybrid orchestration strategies appear promising. One option is to combine the predictability and interpretability of heuristic or metaheuristic methods with the adaptivity of learning-based agents. In such schemes, a heuristic orchestrator could provide a safe baseline placement, while learning-based components propose incremental improvements when conditions change. As

robots are deployed in safety-critical domains, there is also a growing need for orchestration decisions to be explainable to operators and certification bodies. Future work should therefore explore mechanisms that attach human-understandable rationales to placement decisions. These rationales should make explicit which constraints or objectives, such as latency, energy, or resource balance, have motivated a given reallocation.

Third, digital twin frameworks for distributed robotics should be extended both in scope and in openness. On the modeling side, richer descriptions of wireless interference, mobility patterns, and failure modes would enable more realistic stress tests for orchestration policies and service placements. Future work could also generalize the simulator developed in the capstone project into a reusable scenario generator for the digital twin. Such a generator would allow designers to specify, in a structured way, robot trajectories, human presence, network perturbations, and failure events, and to automatically produce both simulation runs and corresponding orchestration workloads. This would facilitate systematic stress-testing of placement strategies and make it easier to reproduce complex experiments across laboratories. The combination of both simulator would offer a full framework to validate distributed robotics solutions from the robot to the infrastructure.

Extending BlazeAIoT and the digital twin simulation to heterogeneous fleets that combine manipulators, aerial robots, and ground vehicles, and to broader application domains such as industrial logistics or smart cities, would further validate the generality of the proposed approaches.

## **8.4      Synthesis**

In summary, this thesis contributes an end-to-end view of IoT Edge-enabled collaborative robotics for smart healthcare, from low-level communication and real-time AI execution to platform design and dynamic orchestration via digital twins. The proposed frameworks and

experimental insights offer concrete tools and guidelines for researchers and practitioners who seek to deploy autonomous mobile robots over heterogeneous E2C infrastructures. By combining systematic measurement, principled modeling, and adaptive orchestration, this work lays the groundwork for future robotic systems that are both more capable and more dependable in complex, networked environments.



## BIBLIOGRAPHY

- Adafruit. (2023). Adafruit Grand Central M4 Express featuring the SAMD51. Retrieved from: <https://www.adafruit.com/product/4064>.
- Ahmed, S. F., Shuravi, S., Afrin, S., Rafa, S. J., Hoque, M. & Gandomi, A. H. (2023, 9). The Power of Internet of Things (IoT): Connecting the Dots with Cloud, Edge, and Fog Computing. Retrieved from: <https://arxiv.org/abs/2309.03420v1>.
- Aiswarya, S., Ramesh, K., Prabha, B., Sasikumar, S. & Vijayakumar, K. (2021). A time optimization model for the Internet of Things-based Healthcare system using Fog computing. *Proceedings of the 2021 IEEE International Conference on Innovative Computing, Intelligent Communication and Smart Electrical Systems, ICSES 2021*, pp. 1–6.
- Akbari, M. R., Barati, H. & Barati, A. (2022). An overlapping routing approach for sending data from things to the cloud inspired by fog technology in the large-scale IoT ecosystem. *Wireless Networks*, 28(2), 521–538.
- Alexander, K., Karina, K., Alexander, N., Roman, K. & Andrei, M. (2024). HaGRID - HAnd Gesture Recognition Image Dataset. *Proceedings - 2024 IEEE Winter Conference on Applications of Computer Vision, WACV 2024*, 4560–4569.
- Alhameed, M. H. (2024). Adaptive Scheduling Architecture for IoT Environment. *ACM International Conference Proceeding Series*, 295–300.
- Alladi, T., Kohli, V., Chamola, V., Yu, F. R. & Guizani, M. (2021). Artificial Intelligence (AI)-Empowered Intrusion Detection Architecture for the Internet of Vehicles. *IEEE Wireless Communications*, 28(3), 144–149.
- Amirloo, E., Rasouli, A., Lakner, P., Rohani, M. & Luo, J. (2022). LatentFormer: Multi-Agent Transformer-Based Interaction Modeling and Trajectory Prediction. 1–10. Retrieved from: <http://arxiv.org/abs/2203.01880>.
- Anshul, R. & Mohan, A. (2022). Miniaturized Electronics: Advantages, Disadvantages and its Applications. *Proceedings of National Seminar on Emerging Trends of Technologies in Current Era*, pp. 28–33.
- Antevski, K., Bernardos, C. J., Groshev, M., Mourad, A., Cominardi, L. & Gazda, R. (2018). Enhancing Edge robotics through the use of context information. *EM-5G 2018 - Proceedings of the 2018 Workshop on Experimentation and Measurements in 5G, Part of CoNEXT 2018*, 7–12.

- Asghari, A. & Sohrabi, M. K. (2024). Server placement in mobile cloud computing: A comprehensive survey for edge computing, fog computing and cloudlet. *Computer Science Review*, 51, 100616.
- Aslan, H. I., Muhammad, S., Haque, M., Witzke, J. & Kao, O. (2025, 10). QONNECT: A QoS-Aware Orchestration System for Distributed Kubernetes Clusters. *Service-Oriented Computing*, pp. 149–156. Retrieved from: <https://arxiv.org/abs/2510.09851v1>.
- Atwal, S., Singh, D. & Chhabra, A. (2025). Resource Allocation Strategies for Edge and Fog Computing in the Cloud Continuum. *2025 International Conference on Intelligent Control, Computing and Communications, IC3 2025*, 208–213.
- Avramovic, A., Sluga, D., Tabernik, D., Skocaj, D., Stojnic, V. & Ilc, N. (2020). Neural-network-based traffic sign detection and recognition in high-definition images using region focusing and parallelization. *IEEE Access*, 8, 189855–189868.
- Bach, S. H., Khoi, P. B. & Yi, S. Y. (2024). Global UWB System: A High-Accuracy Mobile Robot Localization System With Tightly Coupled Integration. *IEEE Internet of Things Journal*, 11(9), 16618–16626.
- Badidi, E. (2019). A Fog Node Architecture for Real-Time Processing of Urban IoT Data Streams. *Advances in Intelligent Systems and Computing*, 984, 330–341. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-030-19807-7\\_32](https://link.springer.com/chapter/10.1007/978-3-030-19807-7_32).
- Bakshi, G., Kumar, A. & Puranik, A. N. (2022). Adoption of Robotics Technology in Healthcare Sector. *Lecture Notes in Electrical Engineering*, 776, 405–414. Retrieved from: [https://link.springer.com/chapter/10.1007/978-981-16-2911-2\\_42](https://link.springer.com/chapter/10.1007/978-981-16-2911-2_42).
- Balouek-Thomert, D. & Parashar, M. (2021, 8). Platforms for edge computing and internet of things applications: A survey. *ACM International Conference Proceeding Series*, pp. 140–149.
- Barbosa, R., Ogobuchi, O. D., Joy, O. O., Saadi, M., Rosa, R. L., Otaibi, S. A. & Rodríguez, D. Z. (2023). IoT based real-time traffic monitoring system using images sensors by sparse deep learning algorithm. *Computer Communications*, 210, 321–330.
- Barodi, A., Bajit, A., Zemmouri, A., Benbrahim, M. & Tamtaoui, A. (2022). Improved Deep Learning Performance for Real-Time Traffic Sign Detection and Recognition Applicable to Intelligent Transportation Systems. *International Journal of Advanced Computer Science and Applications*, 13(5), 712–723. Retrieved from: <https://thesai.org/Publications/ViewPaper?Volume=13&Issue=5&Code=IJACSA&SerialNo=82>.

- Barreto-Cubero, A. J., Gómez-Espinosa, A., Escobedo Cabello, J. A., Cuan-Urquizo, E. & Cruz-Ramírez, S. R. (2022). Sensor data fusion for a mobile robot using neural networks. *Sensors*, 22(1), 1–33.
- Baxi, A., Mageshkumar, V., Murthy, G. S., Merwaday, A., Raju, A., Hosamane, G., Prabhu, S., Natarajan, V., Sudhakaran, S., Cavalcanti, D., Hatakar, A. N., Maciocco, C., Iyer, R. & Vannithamby, R. (2025). Achieving time-critical control in Edge Robotics Systems over 5G. *International Conference on Communication Systems and Networks, COMSNETS*, (2025), 739–747.
- Bédard, C., Lajoie, P. Y., Beltrame, G. & Dagenais, M. (2023). Message flow analysis with complex causal links for distributed ROS 2 systems. *Robotics and Autonomous Systems*, 161, 104361.
- Bedolla-Martinez, D., Kali, Y., Saad, M., Ochoa-Luna, C. & Rahman, M. H. (2023). Learning human inverse kinematics solutions for redundant robotic upper-limb rehabilitation. *Engineering Applications of Artificial Intelligence*, 126, 106966.
- Belkahla, M., Hains, G. J. D. R. & Hains, G. (2025). Measurements and optimizations of the energy consumption of AI systems. *Authorea Preprints*, 1–12. Retrieved from: <https://www.authorea.com/users/962337/articles/1331174-measurements-and-optimizations-of-the-energy-consumption-of-ai-systems?commit=b631a21be651f897a0fc12992f5909ac4fe60db1>.
- Benmerar, T. Z., Theodoropoulos, T., Fevereço, D., Rosa, L., Rodrigues, J., Taleb, T., Barone, P., Tserpes, K. & Cordeiro, L. (2023). Intelligent Multi-Domain Edge Orchestration for Highly Distributed Immersive Services: An Immersive Virtual Touring Use Case. *Proceedings - IEEE International Conference on Edge Computing*, 2023-July, 381–392.
- Bhamare, D., Samaka, M., Erbad, A., Jain, R., Gupta, L. & Chan, H. A. (2017, 7). Multi-objective scheduling of micro-services for optimal service function chains. *IEEE International Conference on Communications*, pp. 1–6.
- Blake, M. B., Remy, S. L., Wei, Y. & Howard, A. M. (2011). Robots on the web. *IEEE Robotics and Automation Magazine*, 18(2), 33–43.
- Blechar, L., Zalewska, P., Filozooi Socjologii, W., Socjologii Zdrowia, Z., Rodziny, M. & Lublinie, U. (2019). The Role of robots in the improving work of nurses. *Nursing in the 21st Century*, 18(3(68)), 174–182. Retrieved from: <https://czasopisma.umlub.pl/piel21w/article/view/311>.

- Bonghi, R. (2007). GitHub - rbonghi/jetson\_stats: Simple package for monitoring and control your NVIDIA Jetson [Orin, Xavier, Nano, TX] series · GitHub. Retrieved from: [https://github.com/rbonghi/jetson\\_stats](https://github.com/rbonghi/jetson_stats).
- Bültemann, M., Rzepka, N., Junger, D., Simbeck, K. & Müller, H. G. (2023). Energy Consumption of AI in Education: A Case Study. *Lecture Notes in Informatics (LNI), Proceedings - Series of the Gesellschaft für Informatik (GI)*, P-338, 219–224.
- Cao, N., Yi, G., Zhang, S. & Qiu, L. (2023). A multiobjective path-smoothing algorithm based on node adjustment and turn-smoothing. *Measurement and Control*, 56(7-8), 1187–01.
- Carastan-Santos, D. & Pham, T. H. T. (2022). Understanding the Energy Consumption of HPC Scale Artificial Intelligence. *Communications in Computer and Information Science*, 1660 CCIS, 131–144.
- Casamayor Pujol, V. & Dustdar, S. (2021). Fog Robotics—Understanding the Research Challenges. *IEEE Internet Computing*, 25(5), 10–17. Retrieved from: <https://ieeexplore.ieee.org/document/9565454/>.
- Casini, D., Pazzaglia, P. & Becker, M. (2025). Managing real-time constraints through monitoring and analysis-driven edge orchestration. *Journal of Systems Architecture*, 163, 103403.
- Caspart, R., Ziegler, S., Weyrauch, A., Obermaier, H., Raffener, S., Schuhmacher, L. P., Scholtyssek, J., Trofimova, D., Nolden, M., Reinartz, I., Isensee, F., Götz, M. & Debus, C. (2022). Precise Energy Consumption Measurements of Heterogeneous Artificial Intelligence Workloads. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13387 LNCS, 108–121. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-031-23220-6\\_8](https://link.springer.com/chapter/10.1007/978-3-031-23220-6_8).
- Cassales, G., Gomes, H. M., Bifet, A., Pfahringer, B. & Senger, H. (2023). Balancing Performance and Energy Consumption of Bagging Ensembles for the Classification of Data Streams in Edge Computing. *IEEE Transactions on Network and Service Management*, 20(3), 3038–3054.
- Chaari, R., Cheikhrouhou, O., Koubaa, A., Youssef, H. & Hmam, H. (2019). Towards a distributed computation offloading architecture for cloud robotics. *2019 15th International Wireless Communications and Mobile Computing Conference, IWCMC 2019*, 434–441.

- Chen, C. C., Guan, Y. H., Novianda, N. R., Teng, C. C. & Yen, M. H. (2022a). Real-Time Traffic Sign Detection for Self-Driving and Energy-Saving Driving Based on YOLOv4 Neural Network. *Proceedings - 2022 IEEE/ACIS 24th International Winter Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2022*, 208–211.
- Chen, C. C., Guan, Y. H., Novianda, N. R., Teng, C. C. & Yen, M. H. (2022b). Real-Time Traffic Sign Detection for Self-Driving and Energy-Saving Driving Based on YOLOv4 Neural Network. *Proceedings - 2022 IEEE/ACIS 24th International Winter Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2022*, 208–211.
- Chen, J. I.-Z., Huang, P.-F. & Pi, C. S. (2023a). The implementation and performance evaluation for a smart robot with edge computing algorithms. *Industrial Robot*, 50(4), 581–594.
- Chen, L. B., Huang, X. R. & Chen, W. H. (2023b). Design and Implementation of an Artificial Intelligence of Things-Based Autonomous Mobile Robot System for Pitaya Harvesting. *IEEE Sensors Journal*, 23(12), 13220–13235.
- Chen, L. B., Huang, X. R. & Chen, W. H. (2023c). Design and Implementation of an Artificial Intelligence of Things-Based Autonomous Mobile Robot System for Pitaya Harvesting. *IEEE Sensors Journal*, 23(12), 13220–13235.
- Chen, L. B., Huang, X. R., Chen, W. H., Pai, W. Y., Huang, G. Z. & Wang, W. C. (2023d). Design and Implementation of an Artificial Intelligence of Things-Based Autonomous Mobile Robot System for Cleaning Garbage. *IEEE Sensors Journal*, 23(8), 8909–8922.
- Chen, M., Islam, M. T., Read, M. R. & Buyya, R. (2026). TraDE: Network and Traffic-Aware Adaptive Scheduling for Microservices Under Dynamics. *IEEE Transactions on Parallel and Distributed Systems*, 37(1), 76–89.
- Chen, Z., Jiang, C. & Guo, Y. (2018). Pedestrian-Robot Interaction Experiments in an Exit Corridor. *2018 15th International Conference on Ubiquitous Robots, UR 2018*, 29–34.
- Chinmay, K. S. & Nisha, G. K. (2024). Review of Autonomous Mobile Robots. *International Conference on E-Mobility, Power Control and Smart Systems: Futuristic Technologies for Sustainable Solutions, ICEMPS 2024*, pp. 1–5.
- Chitta, K., Prakash, A., Jaeger, B., Yu, Z., Renz, K. & Geiger, A. (2023). TransFuser: Imitation with Transformer-Based Sensor Fusion for Autonomous Driving. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(11), 12878–12895. Retrieved from: <https://github.com/autonomousvision/transfuser>.

- Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H. & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. *EMNLP 2014 - 2014 Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference*, 1724–1734. Retrieved from: <https://aclanthology.org/D14-1179/>.
- Chouinard, T. (2020, 5). *CHSLD: manque de personnel et d'équipement, dénonce l'armée*. La Presse. Retrieved from: <https://www.lapresse.ca/actualites/covid-19/2020-05-27/chsld-manque-de-personnel-et-d-equipement-denonce-l-armee.php>.
- Christoforou, E. G., Avgousti, S., Ramdani, N., Novales, C. & Panayides, A. S. (2020). The Upcoming Role for Nursing and Assistive Robotics: Opportunities and Challenges Ahead. *Frontiers in Digital Health*, 2, 585656.
- Chuprov, S., Marinenkov, E., Viksnin, I., Reznik, L. & Khokhlov, I. (2020). Image Processing in Autonomous Vehicle Model Positioning and Movement Control. *IEEE World Forum on Internet of Things, WF-IoT 2020 - Symposium Proceedings*, 1–6.
- Cilic, I., Zarko, I. P. & Kusek, M. (2021, 9). Towards service orchestration for the cloud-to-thing continuum. *2021 6th International Conference on Smart and Sustainable Technologies, SpliTech 2021*, pp. 1–7.
- Clark, R. (2017). *Visual-inertial odometry, mapping and re-localization through learning*. (Ph.D. thesis).
- Clark, R., Wang, S., Wen, H., Markham, A. & Trigoni, N. (2017). VINet: Visual-Inertial Odometry as a Sequence-to-Sequence Learning Problem. *31st AAAI Conference on Artificial Intelligence, AAAI 2017*, 3995–4001. Retrieved from: <https://arxiv.org/abs/1701.08376v2>.
- Cloud Native Computing Foundation. (2026). K3s. Retrieved from: <https://k3s.io/>.
- Coulter, R. C. (1992). *Implementation of the Pure Pursuit Path Tracking Algorithm* (Report n°CMU-RI-TR-92-01). Pittsburgh, PA. Retrieved from: <https://api.semanticscholar.org/CorpusID:62550799>.
- Cousins, S. (2012). Is ROS good for robotics? *IEEE Robotics and Automation Magazine*, 19(2), 13–14.
- Dalton, B. & Taylor, K. (2000). Distributed robotics over the internet. *IEEE Robotics and Automation Magazine*, 7(2), 22–27.

- Danapal, G., Santos, G. A., Da Costa, J. P. C., Praciano, B. J. & Pinheiro, G. P. (2020, 11). Sensor fusion of camera and LiDAR raw data for vehicle detection. *2020 Workshop on Communication Networks and Power Systems, WCNPS 2020, November 12, 2020 - November 13, 2020*.
- Deb, K., Pratap, A., Agarwal, S. & Meyarivan, T. (2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2), 182–197.
- Dey, S. & Mukherjee, A. (2016). Robotic SLAM-A review from fog computing and mobile edge computing perspective. *ACM International Conference Proceeding Series*, 28-November-2016, 153–158.
- Di, X. & Shi, R. (2021). A survey on autonomous vehicle control in the era of mixed-autonomy: From physics-based to AI-guided driving policy learning. *Transportation Research Part C: Emerging Technologies*, 125, 103008.
- Din, I. U., Almogren, A. & Rodrigues, J. J. (2024). AIoT Integration in Autonomous Vehicles: Enhancing Road Cooperation and Traffic Management. *IEEE Internet of Things Journal*, 11(22), 35942–35949.
- Dinar, A. E., Ghouali, S., Guellil, M. S. & Onyema, E. M. (2023). Energy-Efficient and Traffic-Aware Function Analysis of Network Service Orchestration. *Lecture Notes in Networks and Systems*, 591 LNNS, 269–280. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-031-21216-1\\_29](https://link.springer.com/chapter/10.1007/978-3-031-21216-1_29).
- Dordevic, M., Albonico, M., Lewis, G. A., Malavolta, I. & Lago, P. (2023). Computation offloading for ground robotic systems communicating over WiFi – an empirical exploration on performance and energy trade-offs. *Empirical Software Engineering*, 28(6), 140-. Retrieved from: <https://link.springer.com/article/10.1007/s10664-023-10351-6>.
- Dosovitskiy, A., Fischer, P., Ilg, E., Hausser, P., Hazirbas, C., Golkov, V., Smagt, P. v. d., Cremers, D. & Brox, T. (2015, 12). FlowNet: Learning Optical Flow with Convolutional Networks. *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2758–2766. Retrieved from: <https://ieeexplore.ieee.org/document/7410673/>.
- Elfes, A. (1989). Using Occupancy Grids for Mobile Robot Perception and Navigation. *Computer*, 22(6), 46–57.
- Estefo, P., Simmonds, J., Robbes, R. & Fabry, J. (2019). The Robot Operating System: Package reuse and community dynamics. *Journal of Systems and Software*, 151, 226–242.

- Fallon, M., Johannsson, H., Kaess, M. & Leonard, J. J. (2013). The MIT Stata Center dataset. *The International Journal of Robotics Research*, 32(14), 1695–1699.
- Furnadzhiev, R., Shopov, M. & Kakanakov, N. (2025). Efficient Orchestration of Distributed Workloads in Multi-Region Kubernetes Cluster. *Computers 2025*, Vol. 14, Page 114, 14(4), 114. Retrieved from: <https://www.mdpi.com/2073-431X/14/4/114/htmhttps://www.mdpi.com/2073-431X/14/4/114>.
- Gangrade, P., Kushwah, V. S., Date, S. S., Jagdale, A. D., Upreti, K. & Singh, V. K. (2025). Edge and Fog Computing in Cyber-Physical Systems. *2025 International Conference on Intelligent Control, Computing and Communications, IC3 2025*, 172–176.
- Geiger, A., Lenz, P. & Urtasun, R. (2012). Are we ready for autonomous driving? the KITTI vision benchmark suite. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 3354–3361.
- Goh, P. J., Hoe, Z. Y., Low, C. Y., Koh, C. T., Mohammad, U., Lee, K. & Tan, C. F. (2022). Conceptual Design of Cloud-Based Data Pipeline for Smart Factory. *Lecture Notes in Mechanical Engineering*, 25, 29–39. Retrieved from: [https://link.springer.com/chapter/10.1007/978-981-16-8954-3\\_4](https://link.springer.com/chapter/10.1007/978-981-16-8954-3_4).
- Google LLC. (2020). Hands - mediapipe. Retrieved from: <https://chuoling.github.io/mediapipe/solutions/hands.html>.
- Google LLC. (2023, 9). Protocol Buffers Documentation. Retrieved from: <https://protobuf.dev/>.
- Guerrero, C., Lera, I. & Juiz, C. (2019). Evaluation and efficiency comparison of evolutionary algorithms for service placement optimization in fog architectures. *Future Generation Computer Systems*, 97, 131–144.
- Gunasekaran, K. U., Krell, E., Sheta, A. & King, S. A. (2018). Map Generation and Path Planning for Autonomous Mobile Robot in Static Environments Using GA. *2018 8th International Conference on Computer Science and Information Technology, CSIT 2018*, 91–96.
- Hachijo, T., Igarashi, S., Tani, T. & Indo, M. (2024). Development of Autonomous Robots for Construction. *Proceedings of the International Symposium on Automation and Robotics in Construction*, 199–205.
- Haddadin, S. (2024). The Franka Emika Robot: A Standard Platform in Robotics Research. *IEEE Robotics and Automation Magazine*, 31(4), 136–148.

- Han, Y., Zhan, I. H., Zhao, W., Pan, J., Zhang, Z., Wang, Y. & Liu, Y.-J. (2022). Deep Reinforcement Learning for Robot Collision Avoidance With Self-State-Attention and Sensor Fusion. *IEEE Robotics and Automation Letters*, 7(3), 6886–6893.
- Happ, D., Bayhan, S. & Handziski, V. (2021). JOI: Joint placement of IoT analytics operators and pub/sub message brokers in fog-centric IoT platforms. *Future Generation Computer Systems*, 119, 7–19.
- Hardouin, G., Moras, J., Morbidi, F., Marzat, J. & Mouaddib, E. M. (2023). A Multirobot System for 3-D Surface Reconstruction with Centralized and Distributed Architectures. *IEEE Transactions on Robotics*, 39(4), 2623–2638.
- Hejazi, H., Rajab, H., Cinkler, T. & Lengyel, L. (2018). Survey of platforms for massive IoT. *2018 IEEE International Conference on Future IoT Technologies, Future IoT 2018*, 2018-January, 1–8.
- Hochreiter, S. & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735–1780.
- Hockstein, N. G., Gourin, C. G., Faust, R. A. & Terris, D. J. (2007). A history of robots: From science fiction to surgical robotics. *Journal of Robotic Surgery*, 1(2), 113–118. Retrieved from: <https://link.springer.com/article/10.1007/s11701-007-0021-2>.
- Hossucu, A. G. & Özdemir, S. (2025). Context Aware Task Orchestration with Deep Reinforcement Learning in Real Time Fog Computing Simulation Environment. *IEEE Access*, 13, 85004–85025. Retrieved from: <https://papers.ssrn.com/abstract=5009732>.
- Hu, G., Tay, W. P. & Wen, Y. (2012). Cloud robotics: Architecture, challenges and applications. *IEEE Network*, 26(3), 21–28.
- Ilg, E., Mayer, N., Saikia, T., Keuper, M., Dosovitskiy, A. & Brox, T. (2017). FlowNet 2.0: Evolution of optical flow estimation with deep networks. *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, 2017-January, 1647–1655.
- Institut de la statistique du Québec. (2026, 1). Bilan démographique des régions du Québec : la population continue d’augmenter presque partout, mais la croissance ralentit. Retrieved from: <https://statistique.quebec.ca/fr/communiqu/bilan-demographique-regions-quebec-population-continue-augmenter-presque-partout-mais-croissance-ralentit>.
- Intel. (2023). Depth Camera D435i – Intel® RealSense™ Depth and Tracking Cameras. Retrieved from: <https://www.intelrealsense.com/depth-camera-d435i/>.

- Isenko, A. (2023). GitHub - circuit/py-hardware-monitor: A basic, single-file Python hardware monitor · GitHub. Retrieved from: <https://github.com/circuit/py-hardware-monitor>.
- Ivanov, D. (2020). Fog Robotics Distributed Computing in a Monitoring Task. *Advances in Intelligent Systems and Computing*, 1225 AISC, 554–562. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-030-51971-1\\_45](https://link.springer.com/chapter/10.1007/978-3-030-51971-1_45).
- Janji, S. & Kliks, A. (2022). Neural Networks for Path Planning. 1–6. Retrieved from: <http://arxiv.org/abs/2207.00874>.
- Jiang, J., Li, Q., Wang, P. & Liu, Y. (2025). DRKC: Deep Reinforcement Learning Enhanced Microservice Scheduling on Kubernetes Clusters in Cloud-Edge Environment. *IEEE Transactions on Cloud Computing*, 13(4), 1472–1486.
- Jiang, Z., Jovan, F., Moradi, P., Richardson, T., Bernardini, S., Watson, S., Weightman, A. & Hine, D. (2023). A multirobot system for autonomous deployment and recovery of a blade crawler for operations and maintenance of offshore wind turbine blades. *Journal of Field Robotics*, 40(1), 73–93.
- Joglekar, A., Sathe, S., Misurati, N., Srinivasan, S., Schmid, M. J. & Krovi, V. (2022). Deep Reinforcement Learning Based Adaptation of Pure-Pursuit Path-Tracking Control for Skid-Steered Vehicles. *IFAC-PapersOnLine*, 55(37), 400–407.
- Josyula, A., Anand, B. & Rajalakshmi, P. (2019). Fast Object Segmentation Pipeline for Point Clouds Using Robot Operating System. *IEEE 5th World Forum on Internet of Things, WF-IoT 2019 - Conference Proceedings*, 915–919.
- Kabir, H., Tham, M. L. & Chang, Y. C. (2023). Internet of robotic things for mobile robots: Concepts, technologies, challenges, applications, and future directions. *Digital Communications and Networks*, 9(6), 1265–1290.
- Kainova, T. D. (2023). Overview of the Accelerated Platform for Robotics and Artificial Intelligence NVIDIA Isaac. *Proceedings of the Seminar on Information Computing and Processing, ICP 2023*, 89–93.
- Kaneko, Y., Yokoyama, Y., Monma, N., Terashima, Y., Teramoto, K., Kishimoto, T. & Saito, T. (2020). A Microservice-Based Industrial Control System Architecture Using Cloud and MEC. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 12407 LNCS, 18–32. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-030-59824-2\\_2](https://link.springer.com/chapter/10.1007/978-3-030-59824-2_2).

- Kang, M. & Park, M. (2026). Energy Consumption and Performance of AI Accelerators for Edge Devices. 474–482. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-032-13056-3\\_36](https://link.springer.com/chapter/10.1007/978-3-032-13056-3_36).
- Kang, P., Lama, P. & Khan, S. U. (2021). SLO-aware Virtual Rebalancing for Edge Stream Processing. *Proceedings - 2021 IEEE International Conference on Cloud Engineering, IC2E 2021*, 126–135.
- Kang, Z., Canady, R., Dubey, A., Gokhale, A., Shekhar, S. & Sedlacek, M. (2020). A study of publish/subscribe middleware under different iot traffic conditions. *M4IoT 2020 - Proceedings of the 2020 International Workshop on Middleware and Applications for the Internet of Things, Part of Middleware 2020 Conference*, 7–12.
- Kapsalis, A., Kasnesis, P., Venieris, I. S., Kaklamani, D. I. & Patrikakis, C. Z. (2017). A Cooperative Fog Approach for Effective Workload Balancing. *IEEE Cloud Computing*, 4(2), 36–45.
- Khalid, A., Shahbaz, M. & Khan, I. A. (2019). Intelligent use of fog devices in edge-cloud paradigm to assist in E-polling. *Concurrency and Computation: Practice and Experience*, 31(8), e4957.
- Khan, A. & Anwar, Y. (2020). Robots in Healthcare: A Survey. *Advances in Intelligent Systems and Computing*, 944, 280–292. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-030-17798-0\\_24](https://link.springer.com/chapter/10.1007/978-3-030-17798-0_24).
- Khan, S., Avgeris, M., Gascon-Samson, J. & Leivadreas, A. (2025). EMDTORA: Energy-Aware Multi-User Dependent Task Offloading and Resource Allocation in MEC Using Graph-Enabled DRL. *IEEE Transactions on Green Communications and Networking*, 9(3), 1453–1469.
- Khezri, E., Yahya, R. O., Hassanzadeh, H., Mohaidat, M., Ahmadi, S. & Trik, M. (2024). DLJSF: Data-Locality Aware Job Scheduling IoT tasks in fog-cloud computing environments. *Results in Engineering*, 21, 101780.
- Khoshvaght, P., Haider, A., Rahmani, A. M., Gharehchopogh, S., Anka, F., Lansky, J. & Hosseinzadeh, M. (2025). A multi-objective deep reinforcement learning algorithm for spatio-temporal latency optimization in mobile IoT-enabled edge computing networks. *Simulation Modelling Practice and Theory*, 143, 103161.
- Kim, B., Lee, S., Trivedi, A. R. & Song, W. J. (2020). Energy-Efficient Acceleration of Deep Neural Networks on Realtime-Constrained Embedded Edge Devices. *IEEE Access*, 8, 216259–216270.

- Kinova. (2020). Discover our Gen3 lite robot. Retrieved from: [https://www.kinovarobotics.com/product/gen3-lite-robots?gad\\_source=1&gad\\_campaignid=23705055354&gclid=Cj0KCQjwy\\_fOBhC6ARIsAHKFB7\\_h94hqhGoojPUNzPT4ZFsXUaqHjMEAhBkAT40hwvFR4hG3cUd0fQwaAkdOEALw\\_wcB](https://www.kinovarobotics.com/product/gen3-lite-robots?gad_source=1&gad_campaignid=23705055354&gclid=Cj0KCQjwy_fOBhC6ARIsAHKFB7_h94hqhGoojPUNzPT4ZFsXUaqHjMEAhBkAT40hwvFR4hG3cUd0fQwaAkdOEALw_wcB).
- Klimenko, A. (2022). Model and Method of Resource-Saving Tasks Distribution for the Fog Robotics. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13719 LNCS, 210–222. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-031-23609-9\\_19](https://link.springer.com/chapter/10.1007/978-3-031-23609-9_19).
- Klimenko, A. & Barinov, A. (2023). Resource-Saving Multiobjective Task Distribution in the Fog- and Edge-Robotics. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 14214 LNAI, 279–288.
- Kounte, M. R., Rishitha, J., Setty, S. S. & Shivani, S. (2023). Implementation of Realtime design of crowd Enumeration via tracking using AI system. *Proceedings of the International Conference on Intelligent and Innovative Technologies in Computing, Electrical and Electronics, ICIITCEE 2023*, pp. 270 - 274.
- Kramer, M., Lichtenfeld, J., Daun, K. & Von Stryk, O. (2025). Intention-Aware Multi-Target Human Tracking in Sparse Heterogeneous Sensor Networks for Monitoring Outdoor Facilities. *2025 IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2025*, 190–195.
- Krishna Chand Gudi, S. L., Ojha, S., Johnston, B., Clark, J. & Williams, M. A. (2018, 11). Fog robotics for efficient, fluent and robust human-robot interaction. *NCA 2018 - 2018 IEEE 17th International Symposium on Network Computing and Applications*, pp. 1–5.
- Krishnan, K. S. & Sahin, F. (2019). ORBDeepOdometry - A feature-based deep learning approach to monocular visual odometry. *2019 14th Annual Conference System of Systems Engineering, SoSE 2019*, 296–301.
- Krishnappa, M. S., Jayabalan, D., Harve, B. M., Jayaram, V., Bidkar, D. M., Veerapaneni, P. K. & Gejjegondanahalli, V. Y. (2024). Building the Future of IoT: Cloud Platforms, Integration Challenges, and Emerging Applications. *2024 International Conference on Computer and Applications, ICCA 2024*, 1–6.
- Kruse, T., Pandey, A. K., Alami, R. & Kirsch, A. (2013). Human-aware robot navigation: A survey. *Robotics and Autonomous Systems*, 61(12), 1726–1743.

- Ksentini, A., Jebalia, M. & Tabbane, S. (2020). How much can Fog Computing enhance performances of heterogeneous delay-sensitive services in Smart Cities? *2020 8th International Conference on Communications and Networking, ComNet2020 - Proceedings*, 1–8.
- Leela, S., Gudi, K. C., Ojha, S., Johnston, B., Clark, J. & Williams, M.-A. (2017, 9). Fog Robotics: An Introduction. *IEEE/RSJ International Conference on Intelligent Robots and Systems*. Retrieved from: <https://opus.lib.uts.edu.au/handle/10453/120748>.
- Legault Thuot, C. (2024, 5). *Système de santé: pénurie de main-d'œuvre ou exode vers le privé? – Institut de recherche et d'informations socioéconomiques*. Institut de recherche et d'informations socioéconomiques. Retrieved from: <https://iris-recherche.qc.ca/communiqués/main-doeuvre-sante-services-sociaux/>.
- Li, H., Ng, J. K. & Abdelzaher, T. (2022a). Enabling Real-time AI Inference on Mobile Devices via GPU-CPU Collaborative Execution. *Proceedings - 2022 IEEE 28th International Conference on Embedded and Real-Time Computing Systems and Applications, RTCSA 2022*, 195–204.
- Li, K., Xu, Y., Wang, J. & Meng, M. Q. (2019). SARL : DDeep reinforcement learning based human-aware navigation for mobile robot in indoor environments. *IEEE International Conference on Robotics and Biomimetics, ROBIO 2019*, 688–694.
- Li, N., Tan, Y., Wang, X., Li, B. & Luo, J. (2022b). SCORE: A Resource-Efficient Microservice Orchestration Model Based on Spectral Clustering in Edge Computing. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 13740 LNCS, 186–202. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-031-20984-0\\_13](https://link.springer.com/chapter/10.1007/978-3-031-20984-0_13).
- Li, X., Zhou, J., Wei, X., Li, D., Qian, Z., Wu, J., Qin, X. & Lu, S. (2023). Topology-Aware Scheduling Framework for Microservice Applications in Cloud. *IEEE Transactions on Parallel and Distributed Systems*, 34(5), 1635–1649.
- Li, Y., Park, J., Manogharan, G., Ju, F. & Kovalenko, I. (2024, 8). A Mobile Additive Manufacturing Robot Framework for Smart Manufacturing Systems. *Proceedings of ASME 2024 19th International Manufacturing Science and Engineering Conference, MSEC 2024*, 2, V002T05A010.
- Lin, H. I., Shodiq, M. A. F. & Hsieh, M. F. (2025). Robot path planning based on three-dimensional artificial potential field. *Engineering Applications of Artificial Intelligence*, 144, 110127.

- Lindsay, D., Gill, S. S., Smirnova, D. & Garraghan, P. (2021). The evolution of distributed computing systems: from fundamental to new frontiers. *Computing*, 103(8), 1859–1878. Retrieved from: <https://link.springer.com/article/10.1007/s00607-020-00900-y>.
- Linsbauer, S., Hark, R., Koziolk, H. & Eskandani, N. (2024). Runtime Orchestration of Distributed Control System Services with TOSCA, Kubernetes, and GitOps. *Proceedings - IEEE 21st International Conference on Software Architecture Companion, ICSA-C 2024*, 40–47.
- Liu, B., Wang, L. & Liu, M. (2022a). ElasticROS: An Elastically Collaborative Robot Operation System for Fog and Cloud Robotics. 1–23. Retrieved from: <https://arxiv.org/abs/2209.01774v1>.
- Liu, D., Lyu, Z., Zou, Q., Bian, X., Cong, M. & Du, Y. (2022b). Robotic Navigation Based on Experiences and Predictive Map Inspired by Spatial Cognition. *IEEE/ASME Transactions on Mechatronics*, 27(6), 4316–4326.
- Liu, E., Wang, D., Peng, Z., Liu, L. & Gu, N. (2022c). Data-driven Path Following of Unmanned Surface Vehicles based on Model-Based Reinforcement Learning and Model Predictive Path Integral Control. *Proceedings - 2022 37th Youth Academic Annual Conference of Chinese Association of Automation, YAC 2022*, pp. 1045–1049.
- Liu, J., Yang, P. & Chen, C. (2023a). Intelligent energy-efficient scheduling with ant colony techniques for heterogeneous edge computing. *Journal of Parallel and Distributed Computing*, 172, 84–96.
- Liu, R., Shen, J., Chen, C. & Yang, J. (2019). SLAM for Robotic Navigation by Fusing RGB-D and Inertial Data in Recurrent and Convolutional Neural Networks. *2019 IEEE 5th International Conference on Mechatronics System and Robots, ICMSR 2019*, 1–6.
- Liu, T., Ni, S., Li, X., Zhu, Y., Kong, L. & Yang, Y. (2023b). Deep Reinforcement Learning Based Approach for Online Service Placement and Computation Resource Allocation in Edge Computing. *IEEE Transactions on Mobile Computing*, 22(7), 3870–3881.
- Long, Y., Chakraborty, I., Srinivasan, G. & Roy, K. (2021). Complexity-aware adaptive training and inference for edge-cloud distributed ai systems. *Proceedings - International Conference on Distributed Computing Systems*, 2021-July, 573–583.
- Loukatos, D., Arapostathis, V., Karavas, C. S., Arvanitis, K. G. & Papadakis, G. (2024). Power Consumption Analysis of a Prototype Lightweight Autonomous Electric Cargo Robot in Agricultural Field Operation Scenarios. *Energies 2024, Vol. 17, Page 1244*, 17(5), 1244. Retrieved from: <https://www.mdpi.com/1996-1073/17/5/1244/html><https://www.mdpi.com/1996-1073/17/5/1244>.

- Lu, M., Bui, M., Lyu, X. & Chen, M. (2026, 3). Fast Confidence-Aware Human Prediction via Hardware-accelerated Bayesian Inference for Safe Robot Navigation. Retrieved from: <https://arxiv.org/abs/2603.01122v1>.
- Lumpp, F., Fummi, F., Patel, H. D. & Bombieri, N. (2022). 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-October, 9708–9713.
- Lumpp, F., Fummi, F., Patel, H. D. & Bombieri, N. (2024). Enabling Kubernetes Orchestration of Mixed-Criticality Software for Autonomous Mobile Robots. *IEEE Transactions on Robotics*, 40, 540–553.
- Macenski, S., Martín, F., White, R. & Clavero, J. G. (2020). The Marathon 2: A Navigation System. *IEEE International Conference on Intelligent Robots and Systems*, 2718–2725.
- Macenski, S., Foote, T., Gerkey, B., Lalancette, C. & Woodall, W. (2022). Robot Operating System 2: Design, architecture, and uses in the wild. 1–13. Retrieved from: <https://arxiv.org/pdf/2211.07752v1>.
- Manivannan, J., Sowmiya, M. & Bhagyaveni, M. A. (2023). An Energy Efficient Scheduling in Industrial Internet of Things (IIoT). *2023 2nd International Conference on Advances in Computational Intelligence and Communication, ICACIC 2023*, pp. 1–5.
- Mann, Z. A., Metzger, A., Prade, J., Seidl, R. & Pohl, K. (2023). Cost-Optimized, Data-Protection-Aware Offloading Between an Edge Data Center and the Cloud. *IEEE Transactions on Services Computing*, 16(1), 206–220.
- Marinho, M. M., Quiroz-Omana, J. J. & Harada, K. (2024). A Multiarm Robotic Platform for Scientific Exploration: Its Design, Digital Twins, and Validation. *IEEE Robotics and Automation Magazine*, 31(4), 10–20.
- Massoud, M. M., Abdellatif, A. & Atia, M. R. (2022). Different Path Planning Techniques for an Indoor Omni-Wheeled Mobile Robot: Experimental Implementation, Comparison and Optimization. *Applied Sciences (Switzerland)*, 12(24), 12951–12978.
- Matsumoto, K., Yamaguchi, A., Oka, T., Yasumoto, M., Hara, S., Iida, M. & Teichmann, M. (2020, 10). Simulation-based Reinforcement Learning Approach towards Construction Machine Automation. *Proceedings of the 37th International Symposium on Automation and Robotics in Construction, ISARC 2020: From Demonstration to Practical Use - To New Stage of Construction Robot*, pp. 457–464. Retrieved from: [http://www.iaarc.org/publications/2020\\_proceedings\\_of\\_the\\_37th\\_isarc/simulation\\_based\\_reinforcement\\_learning\\_approach\\_towards\\_construction\\_machine\\_automation.html](http://www.iaarc.org/publications/2020_proceedings_of_the_37th_isarc/simulation_based_reinforcement_learning_approach_towards_construction_machine_automation.html).

- Melancon, C., Saad, M. & Zhu, W. H. (2009). Feedback linearization control of mobile robots with obstacle avoidance. *Transactions on Systems, Signals and Devices*, 4(1), 41–53.
- Melancon, C., Kaur, K., Gascon-Samson, J. & Saad, M. (2023). Towards Smart Distributed Robotics Solution using Digital Twin. *Proceedings of the International Conference on Power Electronics and Drive Systems*, 2023-August, 1–6.
- Melancon, C., Saad, M., Kaur, K. & Gascon-Samson, J. (2025). Impact Analysis of Algorithm Optimization on Robot Deep Learning Localization Model for Real-Time Execution. *IEEE Access*, 13, 203253–203267. Retrieved from: <https://ieeexplore.ieee.org/document/11269673>.
- Melancon, C., Gascon-Samson, J., Saad, M., Kaur, K. & Savard, S. (2026, 1). BlazeAIoT: A Modular Multi-Layer Platform for Real-Time Distributed Robotics Across Edge, Fog, and Cloud Infrastructures.
- Melançon, C., Simard, G., Saad, M., Kaur, K. & Gascon-Samson, J. (2023). BlazeFlow: a Multi-Layer Communication Middleware for Real-Time Distributed IoT Applications. *Mid4CC 2023 - Proceedings of the 2023 1st International Workshop on Middleware for the Computing Continuum, Part of: Middleware 2023*, 30–35.
- Mello, R. C., Ribeiro, M. R. & Frizera-Neto, A. (2023). Introduction to Cloud Robotics. *Springer Tracts in Advanced Robotics*, 152, 1–11. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-031-16908-3\\_1](https://link.springer.com/chapter/10.1007/978-3-031-16908-3_1).
- Mendes, E., Koch, P. & Lacroix, S. (2016). ICP-based pose-graph SLAM. 195–200. Retrieved from: <https://hal.archives-ouvertes.fr/hal-01522248>.
- Messaoud, K. (2021). *Deep Learning based Trajectory Prediction for Autonomous Vehicles*. (Ph.D. thesis, Sorbonne Université).
- Messaoud, K., Yahiaoui, I., Verroust-Blondet, A. & Nashashibi, F. (2019). *Relational Recurrent Neural Networks For Vehicle Trajectory Prediction; Relational Recurrent Neural Networks For Vehicle Trajectory Prediction*.
- Microsoft. (2025). Azure network round-trip latency statistics | Microsoft Learn. Retrieved from: <https://learn.microsoft.com/en-us/azure/networking/azure-network-latency?tabs=Americas%2CCanada>.
- Mirus, F., Pasch, F., Singhal, N. & Scholl, K.-U. (2024). A generic approach for reactive stateful mitigation of application failures in distributed robotics systems deployed with Kubernetes. 1–7. Retrieved from: <https://arxiv.org/abs/2410.18825v2>.

- MobileRobots. (2010). Aria: ARIA Overview. Retrieved from: [https://www.eecs.yorku.ca/course\\_archive/2010-11/W/4421/doc/pioneer/aria/main.html](https://www.eecs.yorku.ca/course_archive/2010-11/W/4421/doc/pioneer/aria/main.html).
- Mohanarajah, G., Hunziker, D., D'Andrea, R. & Waibel, M. (2015). Rapyuta: A Cloud Robotics Platform. *IEEE Transactions on Automation Science and Engineering*, 12(2), 481–493.
- MohandSaidi, S. & Mellah, R. (2022). Real-Time Speed Control of a Mobile Robot Using PID Controller. *Lecture Notes in Networks and Systems*, 413 LNNS, 548–556. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-030-96311-8\\_51](https://link.springer.com/chapter/10.1007/978-3-030-96311-8_51).
- Montella, R., Di Luccio, D., Kosta, S., Giunta, G. & Foster, I. (2018). Performance, Resilience, and Security in Moving Data from the Fog to the Cloud: The DYNAMO Transfer Framework Approach. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11226 LNCS, 197–208. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-030-02738-4\\_17](https://link.springer.com/chapter/10.1007/978-3-030-02738-4_17).
- Moore, T. & Stouch, D. (2016). A generalized extended Kalman filter implementation for the robot operating system. *Advances in Intelligent Systems and Computing*, 302, 335–348. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-319-08338-4\\_25](https://link.springer.com/chapter/10.1007/978-3-319-08338-4_25).
- Mukuhi, D. K. & Outtagarts, A. (2023). Distributed Intelligent Resource Orchestration. *International Conference on 6G Networking, 6GNet 2023*, pp. 1–3.
- Munim, M. A. & Rahman Kohinoor, M. S. (2023). Performance Evaluation of Deep Learning-Based Facial Recognition Models on Mobile Computing Environments. *IEEE Region 10 Humanitarian Technology Conference, R10-HTC*, 13–18.
- Muratore, L. & Tsagarakis, N. (2023). XBot2D: towards a robotics hybrid cloud architecture for field robotics. *Frontiers in Robotics and AI*, 10, 1168694.
- Nagabandi, A. (2020). *Model-based Deep Reinforcement Learning for Robotic Systems*. (Ph.D. thesis). Retrieved from: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-158.html>.
- Nakamura, Y., Suwa, H., Arakawa, Y., Yamaguchi, H. & Yasumoto, K. (2016). Middleware for Proximity Distributed Real-Time Processing of IoT Data Flows. *Proceedings - International Conference on Distributed Computing Systems*, 2016-August, 771–772.
- Natali, T., Kurucz, L. M., Fusaglia, M., Mertens, L. S., Ruers, T. J., van Leeuwen, P. J. & Dashtbozorg, B. (2025). Automatic Prostate Volume Estimation in Transabdominal Ultrasound Images. *European Journal of Radiology*, 191, 112274. Retrieved from: <https://arxiv.org/abs/2502.07859v1>.

- Nezami, Z., Zamanifar, K., Djemame, K. & Pournaras, E. (2020). Decentralized Edge-to-Cloud Load-balancing: Service Placement for the Internet of Things. *IEEE Access*, 9, 64983–65000.
- Nie, Q., Tang, D., Liu, C., Wang, L. & Song, J. (2023). A multi-agent and cloud-edge orchestration framework of digital twin for distributed production control. *Robotics and Computer-Integrated Manufacturing*, 82, 102543.
- Nimura, M., Kanai, K. & Katto, J. (2023). Accuracy evaluations of real-time LiDAR-based indoor localization system. *Digest of Technical Papers - IEEE International Conference on Consumer Electronics*, 2023-January, 1–5.
- Niu, W., Kong, Z., Yuan, G., Jiang, W., Guan, J., Ding, C., Zhao, P., Liu, S., Ren, B. & Wang, Y. (2020). Real-Time Execution of Large-scale Language Models on Mobile. 1–4. doi: 10.18653/v1/2020.acl-main.195.
- Novak, J. H. & Kasera, S. K. (2017). Auto-tuning active queue management. *2017 9th International Conference on Communication Systems and Networks, COMSNETS 2017*, 136–143.
- NVIDIA. (2023). Jetson Orin for Next-Gen Robotics. Retrieved from: <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/>.
- Object Management Group. (2023). *What is DDS?* Last accessed: 18/09/2023. Retrieved from: <https://www.dds-foundation.org/what-is-dds-3/>.
- Omron. (2023). Home. Retrieved from: <https://www.adept.com/>.
- Open Robotics. (2025). Executors — ROS 2 Documentation: Humble documentation. Retrieved from: <https://docs.ros.org/en/humble/Concepts/Intermediate/About-Executors.html>.
- Ouster. (2017). High-resolution OS1 lidar sensor: robotics, trucking, mapping | Ouster. Retrieved from: <https://ouster.com/products/scanning-lidar/os1-sensor/>.
- Ouster. (2023). GitHub - ouster-lidar/ouster-ros: Official ROS drivers for Ouster sensors (OS0, OS1, OS2, OSDome) · GitHub. Retrieved from: <https://github.com/ouster-lidar/ouster-ros>.
- Ozturk, C., Tasyurek, M. & Turkdamar, M. U. (2023). Transfer learning and fine-tuned transfer learning methods' effectiveness analyse in the CNN-based deep learning models. *Concurrency and Computation: Practice and Experience*, 35(4), 1–26.

- Paniego, S., Paliwal, N. & Canas, J. M. (2024). Model Optimization in Deep Learning Based Robot Control for Autonomous Driving. *IEEE Robotics and Automation Letters*, 9(1), 715–722.
- Pareigis, S., Tiedemann, T., Schönherr, N., Mihajlov, D., Denecke, E., Tran, J., Koch, S., Abdelkarim, A. & Mang, M. (2023). Artificial Intelligence in Autonomous Systems. A Collection of Projects in Six Problem Classes. *Lecture Notes in Networks and Systems*, 674 LNNS, 123–145. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-031-32700-1\\_14](https://link.springer.com/chapter/10.1007/978-3-031-32700-1_14).
- Park, J. & Kim, T. (2016). A method of logically time synchronization for safety-critical distributed system. *International Conference on Advanced Communication Technology, ICACT*, 2016-March, 356–359.
- Park, M., Park, C. & Kwon, N. K. (2024). Autonomous Driving of Mobile Robots in Dynamic Environments Based on Deep Deterministic Policy Gradient: Reward Shaping and Hindsight Experience Replay. *Biomimetics 2024, Vol. 9, Page 51*, 9(1), 51. Retrieved from: <https://www.mdpi.com/2313-7673/9/1/51/htm><https://www.mdpi.com/2313-7673/9/1/51>.
- Patel, N. M. (2015). Half-Latency Rule for Finding the Knee of the Latency Curve. *ACM SIGMETRICS Performance Evaluation Review*, 43(2), 28–29.
- Peszor, D., Paszkuta, M., Wojciechowska, M. & Wojciechowski, K. (2018). Optical Flow for Collision Avoidance in Autonomous Cars. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10752 LNAI, 482–491. Retrieved from: [https://link.springer.com/chapter/10.1007/978-3-319-75420-8\\_46](https://link.springer.com/chapter/10.1007/978-3-319-75420-8_46).
- Petri, I., Rana, O. F., Bittencourt, L. F., Balouek-Thomert, D. & Parashar, M. (2021). Autonomics at the Edge: Resource Orchestration for Edge Native Applications. *IEEE Internet Computing*, 25(4), 21–29.
- Pololu. (2023). Pololu Dual G2 High-Power Motor Driver 24v14 Shield for Arduino. Retrieved from: <https://www.pololu.com/product/2516>.
- Poojara, S. R., Dehury, C. K., Jakovits, P. & Srirama, S. N. (2021). Serverless data pipeline approaches for IoT data in fog and cloud computing. *Future Generation Computer Systems*, 130, 91–105. Retrieved from: <https://arxiv.org/abs/2112.09974v1>.

- Popic, S., Pezer, D., Mrazovac, B. & Teslic, N. (2016). Performance evaluation of using Protocol Buffers in the Internet of Things communication: Protobuf vs. JSON/BSON comparison with a focus on transportation's IoT. *Proceedings of 2016 International Conference on Smart Systems and Technologies, SST 2016*, 261–265.
- Portugal, D., Rocha, R. P. & Castilho, J. P. (2024). Inquiring the robot operating system community on the state of adoption of the ROS 2 robotics middleware. *International Journal of Intelligent Robotics and Applications*, 9(2), 454–479. Retrieved from: <https://link.springer.com/article/10.1007/s41315-024-00393-4>.
- Poulose, A. & Han, D. S. (2021). Feature-Based Deep LSTM Network for Indoor Localization Using UWB Measurements. *3rd International Conference on Artificial Intelligence in Information and Communication, ICAIIC 2021*, 298–301.
- Puck, L., Keller, P., Schnell, T., Plasberg, C., Tanev, A., Heppner, G., Roennau, A. & Dillmann, R. (2021). Performance Evaluation of Real-Time ROS2 Robotic Control in a Time-Synchronized Distributed Network. *IEEE International Conference on Automation Science and Engineering*, 2021-August, 1670–1676.
- Pudasaini, N., Hanif, M. A. & Shafique, M. (2024). SPAQ-DL-SLAM: Towards Optimizing Deep Learning-Based SLAM for Resource-Constrained Embedded Platforms. *2024 18th International Conference on Control, Automation, Robotics and Vision, ICARCV 2024*, 972–978.
- Qingqing, L., Penia Queralta, J., Gia, T. N., Tenhunen, H., Zou, Z. & Westerlund, T. (2019). Visual Odometry Offloading in Internet of Vehicles with Compression at the Edge of the Network. *2019 12th International Conference on Mobile Computing and Ubiquitous Network, ICMU 2019*, 1–2.
- Qingya, Y., Zhaolin, M. & Kumar, B. V. D. (2025). Empowering Autonomous Driving: Adaptive AI-Orchestrated Distributed System Framework for Balancing Perception Accuracy and Latency. *International Conference on Computer and Communication Engineering Technology, CCET*, (2025), 191–195.
- Qorvo. (2025). MDEK1001 - Qorvo. Retrieved from: <https://www.qorvo.com/products/p/MDEK1001>.
- Quintas, J. M., Menezes, P. J. & Dias, J. M. (2011). Cloud robotics: Towards context aware robotic networks. *Proceedings of the 2nd IASTED International Conference on Robotics, Robo 2011*, 420–427. Retrieved from: [https://www.engineeringvillage.com/share/document.url?mid=cpx\\_6e3d60135a650a68fM748c2061377553&database=cpx&view=detailed](https://www.engineeringvillage.com/share/document.url?mid=cpx_6e3d60135a650a68fM748c2061377553&database=cpx&view=detailed).

- Rajendran, L. & Shyam Shankaran, R. (2021). Bigdata enabled realtime crowd surveillance using artificial intelligence and deep learning. *Proceedings - 2021 IEEE International Conference on Big Data and Smart Computing, BigComp 2021*, 129–132.
- Rancher. (2026). Introduction | RKE2. Retrieved from: <https://docs.rke2.io/>.
- Rappaport, T. S. (2002). *Wireless communications : principles and practice* (ed. 2nd ed). Upper Saddle River, N.J.: Prentice Hall PTR.
- Ray, P. P. (2016). Internet of Robotic Things: Concept, Technologies, and Challenges. *IEEE Access*, 4, 9489–9500.
- RealSenseAI. (2004). GitHub - realsenseai/realsense-ros: ROS Wrapper for RealSense™ Cameras · GitHub. Retrieved from: <https://github.com/realsenseai/realsense-ros>.
- Riedlinger, M. A., Ghafarian Tamizi, M., Tikekar, J. & Redeker, M. (2022). Concept for a Distributed Picking Application Utilizing Robotics and Digital Twins. *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2022-September, 1–4.
- Robotis. (2023a). DYNAMIXEL Protocol 2.0. Retrieved from: <https://emanual.robotis.com/docs/en/dxl/protocol2/>.
- Robotis. (2023b). OpenCR 1.0. Retrieved from: <https://emanual.robotis.com/docs/en/parts/controller/opencr10/>.
- Rossi, M. & Jokic, A. (2023). ROS Framework for Distributed Control of Networks of Dynamical Systems. *2023 46th ICT and Electronics Convention, MIPRO 2023 - Proceedings*, 356–362.
- Roy, N., Dhullipalla, M. H., Sharma, G. P., Dimarogonas, D. V. & Gross, J. (2025). Quality of Control Based Resource Dimensioning for Collaborative Edge Robotics. *Proceedings - IEEE Consumer Communications and Networking Conference, CCNC*, 1–7.
- Roy, S., Baruah, D., Hernandez, S. & Kalafatis, S. (2022a). Distributed Computation and Dynamic Load balancing in Modular Edge Robotics. *Proceedings - 2022 6th IEEE International Conference on Robotic Computing, IRC 2022*, 58–61.
- Roy, S., Baruah, D., Hernandez, S. & Kalafatis, S. (2022b). Distributed Computation and Dynamic Load balancing in Modular Edge Robotics. *Proceedings - 2022 6th IEEE International Conference on Robotic Computing, IRC 2022*, 58–61.

- Rudenko, A., Palmieri, L., Herman, M., Kitani, K. M., Gavrila, D. M. & Arras, K. O. (2020). Human Motion Trajectory Prediction: A Survey. *The International Journal of Robotics Research*, 39(8), 895–935. Retrieved from: <https://arxiv.org/abs/1905.06113>.
- Sahni, Y., Cao, J., Zhang, S. & Yang, L. (2017). Edge Mesh: A New Paradigm to Enable Distributed Intelligence in Internet of Things. *IEEE Access*, 5(5), 16441–16458.
- Sakai, T., Hata, K., Wada, T. & Naito, K. (2022). IoT platform using information flow to reduce load on cloud. *Proceedings - 2022 IEEE 46th Annual Computers, Software, and Applications Conference, COMPSAC 2022*, 1211–1216.
- Samara, G., Hussein, A., Matarneh, I. A., Alrefai, M. & Al-Safarini, M. Y. (2022, 3). Internet of Robotic Things: Current Technologies and Applications. *2021 22nd International Arab Conference on Information Technology, ACIT 2021*, pp. 1–6.
- Sanchez-Gallegos, D. D., Gonzalez-Compean, J. L., Carretero, J., Marin, H., Tchernykh, A. & Montella, R. (2022). PuzzleMesh: A puzzle model to build mesh of agnostic services for edge-fog-cloud. *IEEE Transactions on Services Computing*, 16(2), 1334–1345.
- Sanchez-Lopez, J. L., Wang, M., Olivares-Mendez, M. A., Molina, M. & Voos, H. (2019). A Real-Time 3D Path Planning Solution for Collision-Free Navigation of Multirotor Aerial Robots in Dynamic Environments. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 93(1-2), 33–53.
- Saravanan, M., Perepu, S. K. & Sharma, A. (2018). Exploring collective behavior of internet of robotic things for indoor plant health monitoring. *Proceedings - 2018 IEEE International Conference on Internet of Things and Intelligence System, IOTAIS 2018*, 148–154.
- Schreiber, M., Belagiannis, V., Glaeser, C. & Dietmayer, K. (2020). Motion Estimation in Occupancy Grid Maps in Stationary Settings Using Recurrent Neural Networks. *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 8587–8593. Retrieved from: <https://arxiv.org/abs/1909.11387>.
- Sciavicco, L. & Siciliano, B. (2000). *Modelling and Control of Robot Manipulators*. London: Springer London. Retrieved from: <http://link.springer.com/10.1007/978-1-4471-0449-0>.
- Seal, A. & Mukherjee, A. (2018). On the Emerging Coexistence of Edge, Fog and Cloud Computing paradigms in Real-Time Internets-of-EveryThings which operate in the Big-Squared Data space. *Conference Proceedings - IEEE SOUTHEASTCON*, 2018-April, 1–9.

- Sedghani, H., Filippini, F. & Ardagna, D. (2024). SPACE4AI-D: A Design-Time Tool for AI Applications Resource Selection in Computing Continua. *IEEE Transactions on Services Computing*, 17(6), 4324–4339. Retrieved from: <https://ieeexplore.ieee.org/document/10715700/>.
- Seisa, A. S., Damigos, G., Satpute, S. G., Koval, A. & Nikolakopoulos, G. (2022). Edge Computing Architectures for Enabling the Realisation of the Next Generation Robotic Systems. *2022 30th Mediterranean Conference on Control and Automation, MED 2022*, 487–493.
- Serdaroglu, K. C., Inanc, B. & Baydere, S. (2022). Effect of Messaging Model on the Reliable Data Transfer Latency in a Fog System. *Journal of Network and Systems Management*, 30(4), 1–19. Retrieved from: <https://link.springer.com/article/10.1007/s10922-022-09685-1>.
- Shah, M., Gandhi, K., Pandhi, B. M., Padhiyar, P. & Degadwala, S. (2023). Computer Vision & Deep Learning based Realtime and Pre-Recorded Human Pose Estimation. *Proceedings of the 2nd International Conference on Applied Artificial Intelligence and Computing, ICAAIC 2023*, 313–319.
- Sharma, A., Kaur, H. & Prashar, D. (2023). Maze Solver: A Federated Reinforcement Learning Approach. *2023 14th International Conference on Computing Communication and Networking Technologies, ICCCNT 2023*, 1–5.
- Shilpa, Kaur, T. & Garg, R. (2023). Digital healthcare: A topical and futuristic review of technological and robotic revolution. *Paladyn*, 14(1), 20220108–20220124.
- Simard, G., Melançon, C., Cardinal, P. & Gascon-Samson, J. (2023). Performance Characterization of MQTT Brokers in a Device-Local Edge Deployment. *MiddleWEdge 2023 - Proceedings of the 2nd International Workshop on Middleware for the Edge, Part of: ACM/IFIP Middleware 2023*, 13–18.
- Singh, B., Kumar, R. & Singh, V. P. (2022). Reinforcement learning in robotic applications: a comprehensive survey. *Artificial Intelligence Review*, 55(2), 945–990. Retrieved from: <https://link.springer.com/article/10.1007/s10462-021-09997-9>.
- Singh, K. U., Rai, A. K., Kumar, A., Kumar, S., Singh, T. & Raja, L. (2024). Artificial Intelligent Techniques for Disease Detection in Tomato. *Proceedings of InC4 2024 - 2024 IEEE International Conference on Contemporary Computing and Communications*.
- Slamtec. (2018). GitHub - Slamtec/rplidar\_ros · GitHub. Retrieved from: [https://github.com/slamtec/rplidar\\_ros](https://github.com/slamtec/rplidar_ros).

- Soe, Z. M. M., Thaw, T. Z. & Zaw, Y. M. (2025). Effective Task Scheduling Using Particle Swarm Optimization (PSO) with Various Offloading Strategies in Fog-Cloud Computing Environment. *Lecture Notes in Electrical Engineering*, 1322 LNEE, 211–220.
- Srikonda, S., Norris, W. R., Nottage, D. & Soylemezoglu, A. (2022). Deep Reinforcement Learning for Autonomous Dynamic Skid Steer Vehicle Trajectory Tracking. *Robotics*, 11(5), 95–115.
- Suneel, S., Yadav, R. K., Nirmala Devi, K., Teja sree, B., Aarthi, R. & Upadhyay, S. (2025). Edge Intelligence Hybrid Framework for Real-Time IoT Applications using Multi-Stage Adaptive Processing Methods. *6th International Conference on Mobile Computing and Sustainable Informatics, ICMCSI 2025 - Proceedings*, 380–386.
- Superdroid Robots. (2023). Dual LS3766R Quadrature Encoder Buffer. Retrieved from: <https://www.superdroidrobots.com/store/product=1523>.
- SY AHMAD UBAIDILLAH, S. H., AHMAD, N. & Sahabudin, N. A. (2023). A survey on potential reactive fault tolerance approach for distributed systems in big data. 21.
- Tamar, A., Wu, Y., Thomas, G., Levine, S. & Abbeel, P. (2016). Value Iteration Networks. 1–16. Retrieved from: <https://arxiv.org/abs/1602.02867>.
- Tamiru, M. A., Pierre, G., Tordsson, J. & Elmroth, E. (2021). Mck8s: An orchestration platform for geo-distributed multi-cluster environments. *Proceedings - International Conference on Computer Communications and Networks, ICCCN*, 2021-July, 1–10.
- Tank, B. & Gandhi, V. (2023). A Comparative Study on Cloud Computing, Edge Computing and Fog Computing. *Advances in Transdisciplinary Engineering*, 32, 665–670.
- Technical City. (2024). Jetson AGX Orin 32 GB vs GTX 1660. Retrieved from: <https://technical.city/en/video/GeForce-GTX-1660-vs-Jetson-AGX-Orin-32-GB>.
- Terven, J., Córdova-Esparza, D. M. & Romero-González, J. A. (2023). A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS. *Machine Learning and Knowledge Extraction*, 5(4), 1680–1716. Retrieved from: <https://arxiv.org/abs/2304.00501v7>.
- Thomas, H., Agro, B., Gridseth, M., Zhang, J. & Barfoot, T. D. (2021). Self-Supervised Learning of Lidar Segmentation for Autonomous Indoor Navigation. *IEEE International Conference on Robotics and Automation*, pp. 14047–4053. Retrieved from: <https://arxiv.org/abs/2012.05897>.

- Thomas, H., De Saint Aurin, M. G., Zhang, J. & Barfoot, T. D. (2022). Learning Spatiotemporal Occupancy Grid Maps for Lifelong Navigation in Dynamic Scenes. *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 484–490.
- Thomas, H., Zhang, J. & Barfoot, T. D. (2023). The Foreseeable Future: Self-Supervised Learning to Predict Dynamic Scenes for Indoor Navigation. *IEEE Transactions on Robotics*, 4581–4599.
- Tian, F., Zhang, L., Pan, Q., Liu, J., Zhao, Q. & Hu, B. (2026). MAMILS: A Memory-Aware Multiobjective Scheduler for Real-Time Embedded EEG Depression Diagnosis. *IEEE Transactions on Parallel and Distributed Systems*, 37(3), 600–617. Retrieved from: <https://www.ieee.org/publications/rights/index.html>.
- Tianfield, H. (2019). Towards Edge-Cloud Computing. *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*, 4883–4885.
- Tsokov, T. & Kostadinov, H. (2025). Kinitos: Dynamic network-aware scheduling and descheduling in Kubernetes clusters with mobile nodes. *Journal of Network and Computer Applications*, 238, 104157.
- Tuli, S., Poojara, S. R., Srirama, S. N., Casale, G. & Jennings, N. R. (2022). COSCO: Container Orchestration Using Co-Simulation and Gradient Based Optimization for Fog Computing Environments. *IEEE Transactions on Parallel and Distributed Systems*, 33(1), 101–116.
- Tummala, R. L., Xi, N. & Mukherjee, R. (2002). Distributed robotics special issue. *IEEE Robotics and Automation Magazine*, 9(4), 3.
- TurtleBot3. (2017). TurtleBot3. Retrieved from: <https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/>.
- Valtronic. (2020, 9). Valtronics | Miniaturization advantages and challenges - Valtronic. Retrieved from: <https://valtronic.com/miniaturization-advantages-and-challenges/>.
- Vu, T. D., Nguyen, D. T., Nguyen, D. T., Vu Toan, T. & Tran, V. T. (2024). Real-time Detection to Avoid Accidents of Interaction Between Humans and Collaborative Robot Using Image Processing and Machine Learning. *ICCC 2024 - IEEE 11th International Conference on Computational Cybernetics and Cyber-Medical Systems, Proceedings*, 229–232.
- Waibel, M., Beetz, M., Civera, J., D'Andrea, R., Elfring, J., Gálvez-López, D., Häussermann, K., Janssen, R., Montiel, J. M., Perzylo, A., Schießle, B., Tenorth, M., Zweigle, O. & Van De Molengraft, R. (2011). Robo earth. *IEEE Robotics and Automation Magazine*, 18(2), 69–82.

- Wang, H., Xu, H., Huang, H., Chen, M. & Chen, S. (2023). Robust Task Offloading in Dynamic Edge Computing. *IEEE Transactions on Mobile Computing*, 22(1), 500–514.
- Wang, H., Wang, Q., Li, Y., Chen, G. & Tang, Y. (2018a). Application of Fog Architecture Based on Multi-agent Mechanism in CPPS. *2nd IEEE Conference on Energy Internet and Energy System Integration, EI2 2018 - Proceedings*, 1–6.
- Wang, H., Wang, Q., Li, Y., Chen, G. & Tang, Y. (2018b). Application of Fog Architecture Based on Multi-agent Mechanism in CPPS. *2nd IEEE Conference on Energy Internet and Energy System Integration, EI2 2018 - Proceedings*, 1–6.
- Wang, J., Xu, H., Narasimhan, M. & Wang, X. (2021). Multi-Person 3D Motion Prediction with Multi-Range Transformers. 1–17. Retrieved from: <http://arxiv.org/abs/2111.12073>.
- Wang, S., Clark, R., Wen, H. & Trigoni, N. (2017). DeepVO: Towards End-to-End Visual Odometry with Deep Recurrent Convolutional Neural Networks. *Proceedings - IEEE International Conference on Robotics and Automation*, 2043–2050.
- WitMotion. (2026). Accelerometer +inclinometer Bluetooth5.0 sensor, 9-axis IMU. Retrieved from: <https://witmotion-sensor.com/collections/bluetooth-5-0/products/bluetooth-5-0-accelerometer-inclinometer-wt901blecl-mpu9250-9-axis-imu-sensor>.
- Xianjia, Y., Queralta, J. P., Heikkonen, J. & Westerlund, T. (2021). Federated Learning in Robotic and Autonomous Systems. *Procedia Computer Science*, 191, 135–142.
- Xiao, X., Liu, B., Warnell, G. & Stone, P. (2022). Motion planning and control for mobile robot navigation using machine learning: a survey. *Autonomous Robots*, 46(5), 569–597. Retrieved from: <https://link.springer.com/article/10.1007/s10514-022-10039-8>.
- Yin, H., Sun, Q., Ren, X., Guo, J., Yang, Y., Wei, Y., Huang, B., Chai, X. & Zhong, M. (2023). Development, integration, and field evaluation of an autonomous citrus-harvesting robot. *Journal of Field Robotics*, 40(6), 1363–1387.
- Yu, T., Wang, X. & Shami, A. (2017). A novel fog computing enabled temporal data reduction scheme in iot systems. *Proceedings - IEEE Global Communications Conference, GLOBECOM*, 2018-January, 1–5.
- Yu, X., Queralta, J. P. & Westerlund, T. (2022). Towards Lifelong Federated Learning in Autonomous Mobile Robots with Continuous Sim-to-Real Transfer. *Procedia Computer Science*, 210(C), 86–93. Retrieved from: <https://arxiv.org/abs/2205.15496v1>.

- Zachiotis, G. A., Andrikopoulos, G., Gornez, R., Nakamura, K. & Nikolakopoulos, G. (2018). A Survey on the Application Trends of Home Service Robotics. *2018 IEEE International Conference on Robotics and Biomimetics, ROBIO 2018*, 1999–2006.
- Zambianco, M., Cretti, S. & Siracusa, D. (2024). Cost Minimization in Multi-cloud Systems with Runtime Microservice Re-orchestration. *Proceedings of the 27th Conference on Innovation in Clouds, Internet and Networks, ICIN 2024*, 65–72.
- Zang, X., Yin, M., Huang, L., Yu, J., Zonouz, S. & Yuan, B. (2022, 10). Robot Motion Planning as Video Prediction: A Spatio-Temporal Neural Network-based Motion Planner. *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 12492–12499. Retrieved from: <https://ieeexplore.ieee.org/document/9981769/>.
- Zhai, J., Bi, J., Yuan, H., Wang, M., Zhang, J., Wang, Y. & Zhou, M. C. (2024). Cost-Minimized Microservice Migration With Autoencoder-Assisted Evolution in Hybrid Cloud and Edge Computing Systems. *IEEE Internet of Things Journal*, 11(24), 40951–40967.
- Zhang, S., Zhang, J. & Tao, D. (2022). Information-Theoretic Odometry Learning. *International Journal of Computer Vision*, 130(11), 2553–2570. Retrieved from: <https://link.springer.com/article/10.1007/s11263-022-01659-9>.
- Zhang, S., Wang, C. & Zomaya, A. Y. (2023a). Robustness Analysis and Enhancement of Deep Reinforcement Learning-Based Schedulers. *IEEE Transactions on Parallel and Distributed Systems*, 34(1), 346–357.
- Zhang, W. (2021a). *Learning To Navigate Without a Map*. (Ph.D. thesis, National University of Singapore).
- Zhang, W. (2021b). *Distributed Placement and Resource Orchestration of Real-Time Edge Computing Applications*. (Ph.D. thesis, Rutgers, The State University of New Jersey).
- Zhang, Y., Wurll, C. & Hein, B. (2023b). KubeROS: A Unified Platform for Automated and Scalable Deployment of ROS2-based Multi-Robot Applications. *Proceedings - IEEE International Conference on Robotics and Automation*, 2023-May, 9097–9103.
- Zhao, T., Li, F. & He, L. (2023). DRL-Based Secure Aggregation and Resource Orchestration in MEC-Enabled Hierarchical Federated Learning. *IEEE Internet of Things Journal*, 10(20), 17865–17880.
- Zheng, X., Li, Y., Duan, D., Yang, L., Chen, C. & Cheng, X. (2022). Multivehicle Multisensor Occupancy Grid Maps (MVMS-OGM) for Autonomous Driving. *IEEE Internet of Things Journal*, 9(22), 22944–22957.

- Zhou, Z., Liao, H., Wang, X., Mumtaz, S. & Rodriguez, J. (2020). When Vehicular Fog Computing Meets Autonomous Driving: Computational Resource Management and Task Offloading. *IEEE Network*, 34(6), 70–76.
- Zhu, A., Lu, H., Guo, S., Zeng, Z., Ma, M. & Zhou, Z. (2024). SyRoC: Symbiotic robotics for QoS-aware heterogeneous applications in IoT-edge-cloud computing paradigm. *Future Generation Computer Systems*, 150, 202–219.
- Zhuang, B., Liu, J., Pan, Z., He, H., Weng, Y. & Shen, C. (2023, 8). A Survey on Efficient Training of Transformers. *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, {IJCAI-23}*, pp. 6823–6831. Retrieved from: <https://arxiv.org/abs/2302.01107>.