ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING

ARTIFICIAL INTELLIGENCE

MASTER THESIS

in

Architecture and Platforms for Artificial Intelligence

ROS-BASED VISUAL PERCEPTION USING AN M.2 EDGE AI ACCELERATOR FOR EMBEDDED ROBOTICS

CANDIDATE	SUPERVISOR
Jana Nikolovska	Prof. Davide Rossi

Academic year 2025-2026

Session 3rd

Abstract

Embedded robotic systems increasingly rely on deep neural networks to perform visual perception tasks such as object detection and scene understanding. These models impose significant computational demands that can exceed the capabilities of conventional embedded processors. Edge AI accelerators address this challenge by providing specialized hardware capable of executing neural network inference with high performance and improved energy efficiency. However, many accelerator software development kits are designed for high-throughput streaming pipelines, whereas robotics middleware such as the Robot Operating System (ROS) typically operates using frame-triggered, message-driven execution.

This thesis investigates the system-level integration of the Axelera Metis M.2 edge AI accelerator into a ROS-based perception pipeline running on an embedded NVIDIA Jetson Orin platform. A frame-triggered integration strategy is proposed that enables synchronous invocation of accelerator inference within ROS callbacks without modifying vendor runtime internals. This approach preserves the event-driven execution semantics of ROS while maintaining compatibility with the accelerator's stream-oriented software stack.

To evaluate the proposed integration, accelerator-backed inference nodes were implemented using both Python and C++ host-side environments. The perception pipeline was analyzed through detailed latency measurements that capture preprocessing, accelerator inference, and postprocessing stages within the frame-triggered execution flow. Experimental results demonstrate that the accelerator provides stable and predictable inference execution within the ROS perception pipeline. The analysis further shows that overall perception latency is dominated by host-side processing stages—such as preprocessing, data handling, and postprocessing—rather than by the accelerator inference time itself.

The integration strategy was further validated within a perception-driven robotic manipulation scenario in which object detections produced by the accelerator-based pipeline were used to guide pick-and-place actions. The results demonstrate that the perception subsystem can reliably provide detection outputs suitable for downstream motion planning and control, enabling consistent perception-driven task execution.

Overall, this work demonstrates that AI accelerators designed for stream-oriented execution can be effectively integrated into frame-triggered robotics middleware through appropriate system-level adaptation. The results highlight that accelerator performance must be evaluated within complete perception pipelines rather than solely through isolated neural network inference benchmarks.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Statement 2

1.3 System-Level Validation Scenario 3

1.4 Contributions 4

1.5 Thesis Structure 5

2 Background **7**

2.1 Robotics Middleware and Perception Pipelines 7

2.1.1 System Design Principles in Autonomous Robotics . . 7

2.1.2 ROS as Middleware Architecture 8

2.1.3 Perception–Planning–Control Interaction in Robotic Systems 10

2.1.4 Perception Pipelines in ROS-Based Systems 11

2.2 Embedded Deep Learning Inference 12

2.2.1 YOLO Object Detection Architectures 13

2.3 Hardware Acceleration for Embedded Robotics 15

3 Axelera AI M.2 Accelerator **17**

3.1 Architecture Overview 17

3.2 Voyager SDK Architecture 20

3.2.1 Compilation and Deployment Flow 21
3.2.2 SDK Components 21
3.3 Execution Model and Abstraction Levels 22
3.3.1 Levels of Abstraction 22
3.3.2 Execution Semantics 23
3.4 Architectural and SDK Implications for Robotic Integration 23

4 Adaptation of the Axelera SDK for ROS Perception Pipeline 25
4.1 Problem Statement and Scope 26
4.2 Baseline SDK Execution Model 26
4.3 Adaptation Goals 27
4.4 Adaptation Strategy 28
4.5 ROS Workspace and Node Topology 29
4.6 Auxiliary Nodes and Workspace Modularity 30
4.7 Inference Interface and Processing Stages 30
4.8 Inference Node Architecture 31
4.8.1 Common Inference Pipeline 31
4.8.2 Python Node Implementation 33
4.8.3 C++ Node Implementation 33
4.8.4 Implementation Comparison 34
4.9 Experimental Setup 34
4.9.1 Hardware Platform 35
4.9.2 Test Conditions 35
4.9.3 Metrics and Measurement Methodology 36
4.10 Performance Evaluation 37
4.10.1 End-to-End Latency Comparison 38
4.10.2 Latency Stability 39
4.10.3 Pipeline Stage Latency Breakdown 41
4.10.4 Throughput and Queue Behavior 42

4.10.5 Discussion 43

5 System-Level Validation 47

5.1 Validation Scenario Overview 48

5.2 Logical System Architecture and Data Flow 50

5.2.1 Perception Pipeline and Trigger Generation 51

5.2.2 Pick-and-Place Control and Planning 53

5.2.3 Simulation Execution 54

5.3 Hardware Architecture 56

5.3.1 Embedded Host Platform 56

5.3.2 AI Acceleration Module 56

5.3.3 Vision Sensor 57

5.3.4 Simulation Host 57

5.3.5 Network Topology 58

5.4 Detection Model 58

5.4.1 Datasets 59

5.4.2 Model Selection 61

5.4.3 Training and Model Refinement 62

5.4.4 Selected Detector Performance 63

5.5 Simulation Environment 64

5.5.1 Robot Model and Kinematic Representation 64

5.5.2 Simulation World and Object Modeling 65

5.5.3 End-Effector and Grasping Mechanism 66

5.5.4 Trajectory Execution Interface 67

5.6 Experimental Setup 67

5.7 System-Level Metrics 68

5.7.1 Perception Throughput 69

5.7.2 Perception Pipeline Latency 70

5.7.3 Temporal Stability 70

5.7.4 Detection Stability 71

5.7.5 System Throughput and Manipulation Performance . . 71
5.7.6 Overall System Timing 72
5.8 Results 72
5.8.1 Perception Throughput 72
5.8.2 Perception Pipeline Latency 73
5.8.3 Temporal Stability of the Perception Pipeline 75
5.8.4 System Throughput and Manipulation Performance . . 76
5.8.5 Overall System Bottleneck 76
5.9 Interpretation of System-Level Results 77

6 Conclusion and Future Work 79
6.1 Conclusion 79
6.2 Future Work 81
6.2.1 Optimization of the Inference Pipeline 82
6.2.2 Frame-Oriented Execution Support within the SDK Workflow 82
6.2.3 Evaluation on Physical Robotic Systems 83
6.2.4 Energy Efficiency Analysis 83

Bibliography 85

Acknowledgements 89

List of Figures

3.1 Axelera Metis M.2 AI Inference Acceleration card. 19
3.2 Voyager SDK development and runtime architecture. 19
3.3 Voyager SDK architecture 20

4.1 Voyager SDK stream-based inference pipeline 27
4.2 ROS inference workspace architecture 29
4.3 Frame-triggered inference pipeline 32
4.4 End-to-end latency comparison of Python and C++ nodes . . . 39
4.5 Latency stability summary for YOLOv5n 40
4.6 Pipeline stage latency comparison 42

5.1 Perception-to-action pipeline 48
5.2 Validation workspace layout 49
5.3 Logical architecture of the perception and simulation system . 51
5.4 Perception-driven pick-and-place pipeline execution 55
5.5 Dataset 1 samples 60
5.6 Dataset 2 samples 61
5.7 Confusion matrix for the refined detector model. 64
5.8 Gazebo simulation environment for system validation 65
5.9 Perception pipeline latency breakdown 74
5.10 Inference latency stability over 30-minute evaluation 75
5.11 Perception pipeline latency over 30 minutes 76

List of Tables

2.1 Comparison of representative embedded AI hardware platforms. 16

3.1 Axelera Metis M.2 hardware specifications. 19

4.1 Implementation comparison of inference nodes 34
4.2 Embedded inference platform hardware configuration. 35
4.3 End-to-end latency comparison between Python and C++ inference nodes. 38
4.4 Pipeline stage latency breakdown for the C++ inference node (YOLOv5n). 41
4.5 Pipeline stage latency breakdown for the Python inference node (YOLOv5n). 41
4.6 Effective processing throughput of the inference nodes. 43

5.1 Hardware configuration of the simulation host. 57
5.2 Class distribution for Dataset 1 after filtering. 59
5.3 Dataset 1 split statistics after filtering. 60
5.4 Class distribution for Dataset 2. 61
5.5 Dataset 2 split statistics. 61
5.6 Detector architecture comparison on Dataset 1 62
5.7 Detection performance before and after dataset integration . . 63
5.8 Perception pipeline performance. 73
5.9 Perception pipeline latency breakdown. 73
5.10 End-to-end pipeline timing comparison. 77

List of Abbreviations

AI Artificial Intelligence

AIPU Artificial Intelligence Processing Unit

API Application Programming Interface

CNN Convolutional Neural Network

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

CV Computer Vision

DDS Data Distribution Service

DNN Deep Neural Network

DRAM Dynamic Random Access Memory

FPS Frames Per Second

GPU Graphics Processing Unit

INT8 8-bit Integer Precision

ML Machine Learning

MLOps Machine Learning Operations

NMS Non-Maximum Suppression

ONNX Open Neural Network Exchange

OpenCV Open Source Computer Vision

PC Personal Computer

PCIe Peripheral Component Interconnect Express

ROS Robot Operating System

SDK Software Development Kit

SIMT Single Instruction Multiple Threads

TOPS Tera Operations Per Second

USB Universal Serial Bus

YOLO You Only Look Once

Chapter 1

Introduction

1.1 Motivation

Robotic systems increasingly operate in complex real-world environments where reliable visual perception is essential for enabling higher-level planning and control. Traditional rule-based vision methods often struggle under such conditions, particularly in the presence of clutter, illumination changes, and object variability. The emergence of convolutional neural networks (CNNs) has significantly improved the performance of perception systems, enabling robust object detection, segmentation, and scene understanding. These advances allow robotic systems to interpret their surroundings with a level of robustness and semantic richness that was previously difficult to achieve. However, the improved perception capabilities offered by CNN-based models come at the cost of substantially increased computational demands.

Robotic systems operate under strict constraints on power consumption, computational resources, and latency. Unlike cloud-based training environments, embedded platforms must deliver real-time inference with bounded response times while operating within limited energy budgets. As a result, deploying computationally intensive CNN-based perception models on embedded robotic systems presents a significant system-level deployment challenge. These constraints have accelerated the development of specialized edge

AI accelerators that provide significantly higher performance per watt compared to general-purpose CPUs and GPUs.

However, raw accelerator performance alone does not guarantee successful deployment in robotic systems. The integration of hardware accelerators into middleware-based software architectures introduces system-level challenges that are often overlooked in accelerator deployment. In particular, robotics frameworks such as the Robot Operating System (ROS) are built around message-driven, frame-triggered execution semantics. In this execution model, each incoming sensor frame, such as an image message, triggers a discrete processing cycle within a callback function, and the perception pipeline is executed once per frame. In contrast, many accelerator software development kits (SDKs) adopt stream-oriented execution models in which data is processed continuously as part of a runtime-controlled pipeline rather than being explicitly invoked for individual frames. The semantic gap between these execution paradigms necessitates a dedicated system-level integration strategy.

1.2 Problem Statement

The Axelera Metis M.2 edge AI accelerator provides an SDK optimized for high-throughput inference using a stream-based execution paradigm. In contrast, ROS-based perception systems typically rely on discrete, callback-driven processing of individual sensor frames.

The central challenge addressed in this thesis is therefore not model accuracy, but system-level integration. Accordingly, this thesis investigates the following research question:

> How can a stream-oriented AI accelerator be integrated into a frame-triggered ROS perception pipeline without modifying vendor runtime internals, while preserving deterministic behavior and bounded processing latency?

Beyond enabling functional integration, it is necessary to evaluate whether such an accelerator-based perception pipeline can operate reliably within the timing constraints of robotic systems. In particular, the integration must support stable real-time operation while maintaining predictable latency and avoiding message backlog within the ROS execution framework.

In frame-triggered perception pipelines, each incoming sensor frame initiates a new processing cycle. To maintain stable operation, the end-to-end processing time must remain below the frame period of the input stream so that each frame can be processed before the arrival of the next one. Formally, if the sensor operates at a frame rate f, the perception pipeline must satisfy

$$T_{\text{processing}} < \frac{1}{f} \tag{1.1}$$

to avoid message backlog within the ROS callback execution model [1]. This requirement defines the bounded processing latency considered in this work.

To examine these aspects, this work implements and evaluates accelerator-backed inference nodes using two host-side implementations, one in Python and one in C++. By analyzing their latency characteristics, execution stability, and throughput within a ROS-based perception pipeline, this study examines how host-side software design influences the overall behavior of accelerator-accelerated perception in embedded robotic systems.

1.3 System-Level Validation Scenario

To evaluate the practical implications of the proposed integration strategy, this thesis further investigates its behavior within a complete perception–planning–control pipeline. Once the accelerator is integrated into ROS, the developed inference nodes are deployed in a representative embedded robotics scenario designed and fully implemented as part of this thesis.

In this scenario, visual perception is performed using a USB camera, and

object detection is executed on the embedded accelerator using a fine-tuned YOLO-based model. The detected objects are then incorporated into a simulated robotic manipulation environment. Motion planning and control are realized using a robotic arm model with established ROS-based tools for planning and visualization. This setup enables closed-loop interaction between perception and manipulation while maintaining full observability of system latency and execution behavior.

The objective of this validation scenario is not to position the application itself as the primary contribution, but to evaluate how the proposed integration strategy behaves within a complete robotic software stack. In particular, it allows evaluation of end-to-end timing behavior, data flow consistency, and robustness under realistic perception workloads.

1.4 Contributions

The main contributions of this thesis are:

- A structured analysis of the execution-model mismatch between stream-based accelerator SDKs and frame-triggered ROS perception pipelines.
- The design and implementation of a frame-triggered accelerator invocation strategy that enables integration without modification of vendor runtime internals.
- The development of two host-side integration variants (Python and C++) to analyze the impact of software abstraction level on perception pipeline performance.
- Experimental evaluation of the proposed integration on an embedded robotic platform, including a detailed analysis of end-to-end latency and host-side processing overhead within a ROS perception pipeline.

- System-level validation of the proposed integration within a closed-loop perception–planning–control pipeline.

- An empirical system-level insight demonstrating that accelerator inference latency is not the dominant factor in overall perception pipeline performance. The experimental results show that once neural network inference is accelerated, host-side stages such as preprocessing, data handling, and postprocessing become the primary contributors to end-to-end latency. This finding indicates that the accelerator successfully removes the traditional inference bottleneck and shifts the performance limitation to host-side pipeline components, highlighting the importance of holistic system-level optimization when integrating edge AI accelerators into robotic perception pipelines.

1.5 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 provides background on robotic middleware and embedded AI acceleration. Chapter 3 introduces the Axelera Metis hardware platform and its associated software stack. Chapter 4 presents the proposed system integration strategy and its performance evaluation. Chapter 5 describes the system-level validation scenario and analyzes end-to-end behavior within a perception-driven robotic manipulation setup. Finally, Chapter 6 summarizes the findings and outlines directions for future work.

Chapter 2

Background

2.1 Robotics Middleware and Perception Pipelines

2.1.1 System Design Principles in Autonomous Robotics

Modern autonomous robotic systems are designed as distributed computational architectures that integrate sensing, perception, planning, and control under strict operational constraints. Such systems must operate in dynamic environments while maintaining responsiveness, robustness, and adaptability. To manage increasing system complexity, robotic software is typically structured according to three fundamental design principles: modularity, scalability, and reusability [2].

Modularity allows individual subsystems—such as perception, localization, planning, and control—to be developed and tested independently. This separation of concerns reduces system complexity and facilitates debugging and maintenance.

Scalability ensures that new sensors, algorithms, or hardware components can be integrated without redesigning the entire system architecture.

Reusability enables software components to be deployed across different

robotic platforms and applications, reducing development effort and improving reliability through standardized interfaces.

These design principles necessitate a communication framework capable of coordinating heterogeneous hardware and software components while preserving clear abstraction boundaries between subsystems. Middleware solutions have therefore become central to modern robotic system design.

2.1.2 ROS as Middleware Architecture

The Robot Operating System (ROS) has emerged as the de facto middleware framework in academic and industrial robotics [3]. Despite its name, ROS is not a standalone operating system but rather a middleware layer that operates on top of conventional operating systems, most commonly Linux-based distributions such as Ubuntu. It relies on standard operating system services including process scheduling, networking, threading, and memory management, while providing higher-level abstractions tailored to robotic applications.

The original ROS architecture, commonly referred to as ROS 1, introduced a modular approach to robotic software development through a distributed node-based architecture. Functionality is encapsulated in independent computational units known as *nodes*, each executing as a separate process. Nodes communicate through standardized message interfaces using asynchronous publish-subscribe mechanisms (topics) or synchronous request-response interactions (services). This communication model promotes loose coupling between components, enabling modular system design and scalability.

However, several limitations of ROS 1—particularly regarding scalability, real-time capabilities, and distributed deployment—motivated the development of its successor, ROS 2. While maintaining the core architectural concepts of ROS, ROS 2 introduces significant improvements to the underlying communication infrastructure.

In ROS 2, communication between nodes is implemented using the Data

Distribution Service (DDS), a distributed publish-subscribe middleware designed for scalable and real-time systems [4]. This middleware enables more reliable communication across distributed robotic systems and provides improved support for multi-robot deployments and heterogeneous hardware platforms.

Node execution in ROS 2 is managed by an *executor* responsible for scheduling callback functions associated with subscriptions, timers, and services. Executors can operate in either single-threaded or multi-threaded modes, enabling both deterministic synchronous processing and higher-throughput asynchronous execution strategies. The design and behavior of ROS 2 executors and the underlying middleware architecture are discussed in detail by Macenski et al. [5] and Maruyama et al. [1].

In addition to its execution model, ROS organizes functionality into *packages*, which serve as modular units of software distribution. A package typically contains source code, configuration files, interface definitions, and dependency specifications, enabling structured development and reuse across projects. Application execution is commonly orchestrated through *launch files*, which define node instantiation, topic remapping, and parameter configuration. This mechanism supports reproducible system deployment and centralized parameter management in complex robotic applications.

By defining a standardized communication protocol and execution paradigm, ROS promotes interoperability between heterogeneous hardware drivers, perception algorithms, and control modules. Its package management system and clearly defined interface specifications further enhance software reusability across platforms. The architectural principles of ROS—distributed execution, message-based communication, and abstraction of hardware interfaces—directly reflect the design requirements of complex autonomous robotic systems.

2.1.3 Perception–Planning–Control Interaction in Robotic Systems

Autonomous robotic systems are commonly organized as a perception-planning-control pipeline [6]. Sensor data is first processed by perception components that extract structured information about the environment, such as object detections, semantic labels, or spatial relationships. These outputs form an intermediate representation of the robot's surroundings that can be consumed by higher-level decision-making modules.

Planning components use this information to determine appropriate actions in order to achieve task objectives while respecting kinematic, dynamic, and environmental constraints. Depending on the application, planning may involve trajectory generation, motion planning, or task-level decision making. The resulting plans are translated into control commands that are executed by the robot's actuators through dedicated control interfaces.

Within ROS-based systems, these subsystems are typically implemented as distributed nodes that communicate through message-based interfaces. Perception nodes publish structured environmental information, while planning and control nodes subscribe to these messages and generate corresponding motion commands. This modular architecture enables clear separation between sensing, decision making, and actuation while maintaining flexibility in system composition.

During development and validation, the execution stage may be performed either on physical robotic hardware or within physics-based simulation environments. Simulation frameworks are widely used in robotics research because they allow perception, planning, and control algorithms to be tested in controlled and repeatable conditions without the risks or costs associated with physical hardware. In such environments, robot models, sensors, and actuators are represented by physics engines that approximate the dynamics of real

robotic systems while exposing the same software interfaces used by the control stack. This allows developers to evaluate perception-driven behaviors, motion planning strategies, and control policies before deployment on physical robots. While simulation cannot perfectly reproduce real-world dynamics or sensor noise, it provides an effective environment for rapid experimentation and system-level validation.

2.1.4 Perception Pipelines in ROS-Based Systems

Within ROS-based robotic systems, perception components typically operate under a frame-based, event-driven execution model. Sensor drivers publish image messages to dedicated topics, and perception nodes process these messages through callback functions that are triggered whenever a new frame arrives. Each callback invocation initiates a processing cycle in which the received frame is preprocessed, passed through inference or other perception algorithms, and converted into structured outputs such as detections or classifications. These outputs are subsequently consumed by planning and control nodes, forming part of the perception–planning–control loop.

Visual perception plays a central role in enabling robots to interpret their environment and support higher-level decision making [7]. This callback-driven and distributed execution model promotes modularity and extensibility, as perception components can be replaced without modifying downstream modules, provided message interfaces remain consistent. However, it also introduces constraints related to timing, synchronization, and data consistency. Since each frame is processed independently upon arrival, blocking operations or variable execution times directly affect system responsiveness and may lead to latency accumulation.

Consequently, computational components integrated into a ROS-based perception pipeline must operate within this message-driven, frame-oriented execution model and produce results within bounded processing time for each

frame. Components that violate these timing constraints may introduce latency accumulation or message backlog, negatively affecting the responsiveness of downstream planning and control modules.

Modern robotic perception pipelines increasingly rely on deep learning models for tasks such as object detection and scene understanding. Among these models, the YOLO family of object detectors has become widely adopted in real-time robotic perception systems due to its favorable trade-off between detection accuracy and computational efficiency.

2.2 Embedded Deep Learning Inference

Deep learning models now form the foundation of modern robotic perception systems, enabling tasks such as object detection, classification, and semantic segmentation. While training these models is computationally intensive and typically performed offline on high-performance hardware, inference represents the operational phase deployed on robotic platforms. During deployment, inference must satisfy strict timing and resource constraints in order to ensure stable and responsive system behavior. Embedded platforms typically operate under limited computational resources, restricted power budgets, and thermal constraints, which require efficient execution of inference workloads.

An inference pipeline generally consists of three stages: (i) preprocessing of input data into the required tensor representation, (ii) execution of the neural network on a computational backend, and (iii) postprocessing of outputs into interpretable detections or classifications.

The deployment process begins with exporting the trained neural network to a graph-based intermediate representation that can be compiled for a target hardware backend [8]. This abstraction decouples model development from hardware execution and enables portability across heterogeneous platforms. The deployment process can therefore be summarized as model export, hardware-specific compilation and optimization, and runtime integration

through a dedicated software development kit (SDK).

Hardware-aware optimization commonly includes quantization, which reduces numerical precision (e.g., from 32-bit floating point to 8-bit integer representations) to decrease memory footprint and computational cost while preserving inference accuracy within acceptable bounds [9]. While quantization is often applied as an optional optimization to improve inference efficiency, certain edge AI accelerators require quantized models as part of their supported execution format.

Inference execution semantics differ across platforms. Many edge AI accelerators are designed primarily for continuous streaming workloads, where data is processed as part of an uninterrupted pipeline. Others support batch-oriented or explicitly synchronous invocation. In middleware-driven robotic systems, however, perception components typically operate under frame-based execution semantics, where each input frame is processed independently and results are produced deterministically. Consequently, deployed inference systems must align with the execution requirements imposed by robotic middleware while maintaining bounded latency and predictable processing behavior.

2.2.1 YOLO Object Detection Architectures

Object detection is a fundamental component of robotic perception systems, enabling robots to identify and localize objects within their environment. From the various deep learning approaches proposed for object detection, the *You Only Look Once* (YOLO) family of models has emerged as a prominent solution for real-time applications due to its computational efficiency and strong detection performance [10].

Traditional object detection pipelines often relied on multi-stage architectures that first generated candidate object regions and subsequently classified them. In contrast, YOLO-based models formulate object detection as a single-stage prediction problem in which bounding box coordinates and class

probabilities are directly predicted from image features within a single neural network forward pass. This unified design significantly reduces inference latency, making YOLO models well suited for real-time systems such as robotic perception pipelines.

Modern YOLO architectures typically consist of three main components: a backbone, a neck, and a detection head. The backbone network extracts hierarchical feature representations from the input image, capturing both low-level spatial features and high-level semantic information. The neck aggregates features across multiple spatial scales, enabling the detection of objects of different sizes. Finally, the detection head predicts bounding box coordinates, objectness scores, and class probabilities for each candidate object.

In this work, object detection models from the YOLOv5 and YOLOv8 families are used to evaluate the perception pipeline. YOLOv5 [11] is a widely adopted architecture designed to provide efficient real-time detection across a range of hardware platforms. It incorporates Cross Stage Partial (CSP) modules in the backbone and employs a Path Aggregation Network (PANet) structure to combine multi-scale feature information.

YOLOv8 [12] represents a more recent evolution of the YOLO architecture and introduces several design refinements, including an anchor-free detection head and improved training strategies. The anchor-free formulation simplifies bounding box prediction by directly regressing object locations without relying on predefined anchor boxes, which can improve both training stability and detection performance.

Both YOLOv5 and YOLOv8 are available in multiple model sizes that trade detection accuracy for computational cost. Lightweight variants, such as the *nano (n)* and *small (s)* configurations used in this work, are specifically designed for low-latency inference on embedded platforms. These characteristics make YOLO-based models well suited for deployment on edge AI accelerators within real-time robotic perception pipelines.

2.3 Hardware Acceleration for Embedded Robotics

Embedded robotic platforms operate under strict resource constraints, including limited computational capacity, restricted power budgets, and thermal limitations [13]. General-purpose GPUs have traditionally been used to accelerate deep learning inference due to their high parallel processing capability and mature software ecosystems [14]. However, GPUs are designed as flexible, general-purpose parallel processors and may exhibit relatively high energy consumption and memory overhead when deployed in embedded systems. In compact robotic platforms—where physical space, heat dissipation capacity, and energy availability are inherently constrained—such characteristics can introduce architectural challenges and restrict sustainable long-term operation.

In response to these constraints, dedicated AI accelerators have emerged as specialized hardware solutions tailored specifically for neural network inference [15]. By exploiting the structured computation patterns characteristic of deep learning models—particularly convolutions and matrix multiplications—these architectures allocate computational and memory resources more directly to inference-critical operations. This specialization enables improved performance-per-watt efficiency and makes such accelerators particularly suitable for embedded robotic deployment [16].

In practice, embedded AI deployments rely on several classes of hardware platforms, including integrated GPU-based systems, lightweight inference accelerators, and domain-specific neural network processors. For example, NVIDIA Jetson platforms provide integrated GPUs that support deep learning inference through frameworks such as CUDA and TensorRT, offering a flexible environment for developing robotic perception systems [17]. Alternatively, specialized inference devices such as the Google Coral Edge TPU are designed for energy-efficient execution of quantized neural networks in edge environments [18]. More recent domain-specific accelerators, including

architectures such as the Axelera Metis used in this work, aim to provide high inference throughput while maintaining power consumption compatible with embedded robotic platforms [19].

Table 2.1: Comparison of representative embedded AI hardware platforms.

Platform	Example	INT8 TOPS	Power (W)
Embedded GPU	Jetson Orin family	$\sim$40–275	$\sim$7–60
Edge TPU	Coral Edge TPU	$\sim$4	$\sim$2
AI accelerator	Axelera Metis M.2	214	3.5–9

These representative specifications illustrate the different design trade-offs targeted by embedded AI hardware platforms. Integrated GPU systems provide flexibility and strong software support but typically operate within higher power envelopes. Lightweight inference accelerators such as the Edge TPU prioritize energy-efficient execution of quantized neural network workloads but provide more limited computational capacity. Dedicated AI accelerators aim to combine high inference throughput with improved performance-per-watt efficiency by specializing hardware execution units for neural network operations.

From a system-level perspective, however, hardware acceleration in robotics must be evaluated not only in terms of raw computational throughput, but also with respect to latency characteristics, integration complexity, and software ecosystem maturity. In practical deployments, the interaction between the host processor and the accelerator—including memory transfers, execution synchronization, and runtime management—plays a decisive role in determining effective end-to-end performance within a robotic perception pipeline. Consequently, evaluating accelerator platforms in isolation may not accurately reflect their impact on the overall system performance of embedded robotic applications.

Chapter 3

Axelera AI M.2 Accelerator

This chapter introduces the Axelera Metis M.2 AI accelerator and the Voyager software development kit (SDK) used throughout this work. Understanding the architectural characteristics and execution model of this platform is essential for analyzing the integration challenges discussed later in the thesis.

The chapter first presents an overview of the Metis hardware architecture and its relevance for embedded inference workloads. It then describes the structure of the Voyager SDK, including the model compilation workflow and runtime interfaces used to execute inference tasks. Particular attention is given to the execution semantics of the SDK, as these design choices directly influence the integration of the accelerator within ROS-based perception pipelines.

The architectural diagrams and software stack representations presented in this chapter are derived from official documentation and publicly available technical materials released by Axelera AI [19, 20].

3.1 Architecture Overview

The Axelera AI Metis M.2 accelerator is an inference-focused hardware module designed for embedded and edge computing applications. It is implemented in the M.2 2280 M-key form factor and connects to the host system via PCIe Gen 3.0 x4, enabling compact integration into industrial PCs and robotic

platforms while maintaining high-bandwidth communication.

The device integrates a quad-core Metis® AI Processing Unit (AIPU) optimized specifically for neural network inference. In contrast to general-purpose GPU architectures, which commonly use a Single-Instruction-Multiple-Thread (SIMT) execution model and separate memory and compute hierarchies, the Metis AIPU adopts a domain-specific design tailored to structured deep learning operations such as convolutions and matrix multiplications.

According to vendor technical documentation [19], the Metis architecture incorporates in-memory computing principles, in which computation is performed closer to memory arrays in order to reduce data movement overhead. Since data transfer between memory and compute units constitutes a significant portion of energy consumption in deep learning workloads, minimizing such transfers can improve performance-per-watt efficiency. These architectural characteristics were presented in the Industry Invited Session of the 2024 IEEE International Solid-State Circuits Conference (ISSCC) [21], situating the design within contemporary research on energy-efficient AI accelerators. The specific performance claims associated with this architecture are based on vendor-reported materials and are not independently validated within the scope of this thesis.

Key hardware specifications are summarized in Table 3.1.

The moderate power envelope (3.5-9 W) and compact M.2 form factor align with the spatial and thermal constraints typical of embedded robotic platforms. However, the limited on-board memory (1 GB DRAM) imposes constraints on deployable model size and intermediate tensor buffering, reinforcing the need for quantized and memory-efficient architectures.

A functional overview of the development and runtime separation is illustrated in Fig. 3.2. The diagram emphasizes the distinction between offline model preparation and runtime execution, highlighting the API layers mediating interaction between the host system and the Metis hardware.

Table 3.1: Axelera Metis M.2 hardware specifications.

Feature	Specification
Form factor	M.2 2280 M-key
AIPU	Quad-core Metis architecture
Peak INT8 throughput	214 TOPS
On-board memory	1 GB DRAM
Host interface	PCIe Gen3 x4 (4 GB/s)
Power envelope	3.5-9 W
Operating temperature	−20°C to 70°C
Security	Secure Boot, Root of Trust

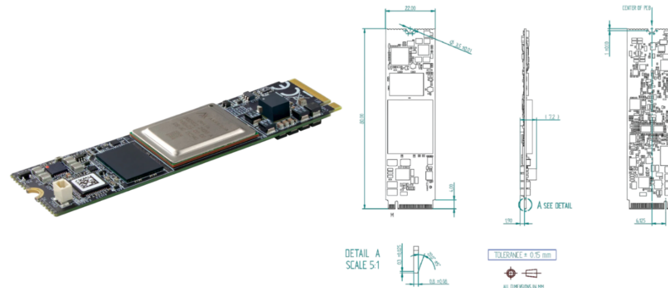


Figure 3.1: Axelera Metis M.2 AI Inference Acceleration card.

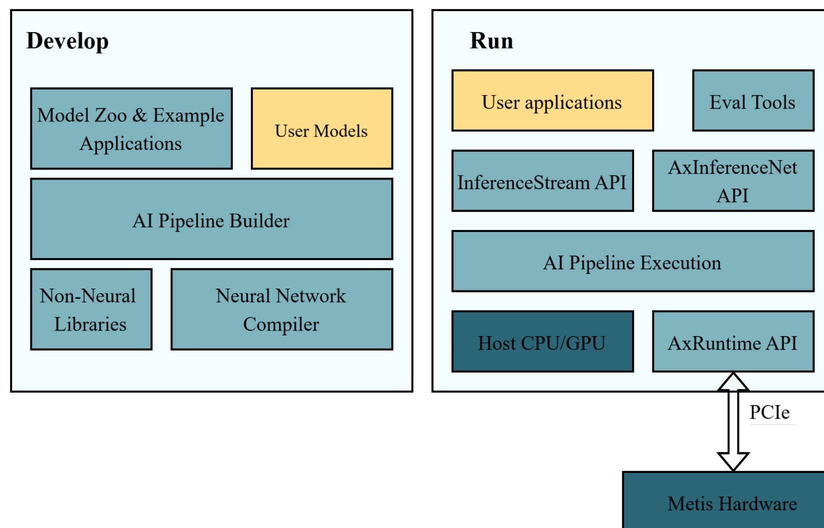


Figure 3.2: Voyager SDK development and runtime architecture.

The architecture separates development-time components (model zoo, user models, AI Pipeline Builder, neural network compiler, and non-neural libraries) from runtime components (Inference APIs, AI Pipeline Execution, and AxRuntime API). The runtime layer manages host–accelerator communication over PCIe to the Metis hardware, enabling structured deployment of compiled inference pipelines.

3.2 Voyager SDK Architecture

The Axelera Metis accelerator is supported by the Voyager® SDK, which provides a structured toolchain for model compilation, deployment, and runtime execution [20]. As discussed in Section 2.2, deployment of deep learning models on embedded accelerators typically involves exporting a trained model to an intermediate representation, applying hardware-specific optimization and quantization, and integrating the resulting executable into a runtime environment. The Voyager SDK encapsulates these steps within a unified software stack.

The overall architecture of the Voyager SDK is illustrated in Figure 3.3. The stack separates model preparation, pipeline construction, and runtime execution layers, enabling structured deployment of neural network inference workloads on the Metis hardware.

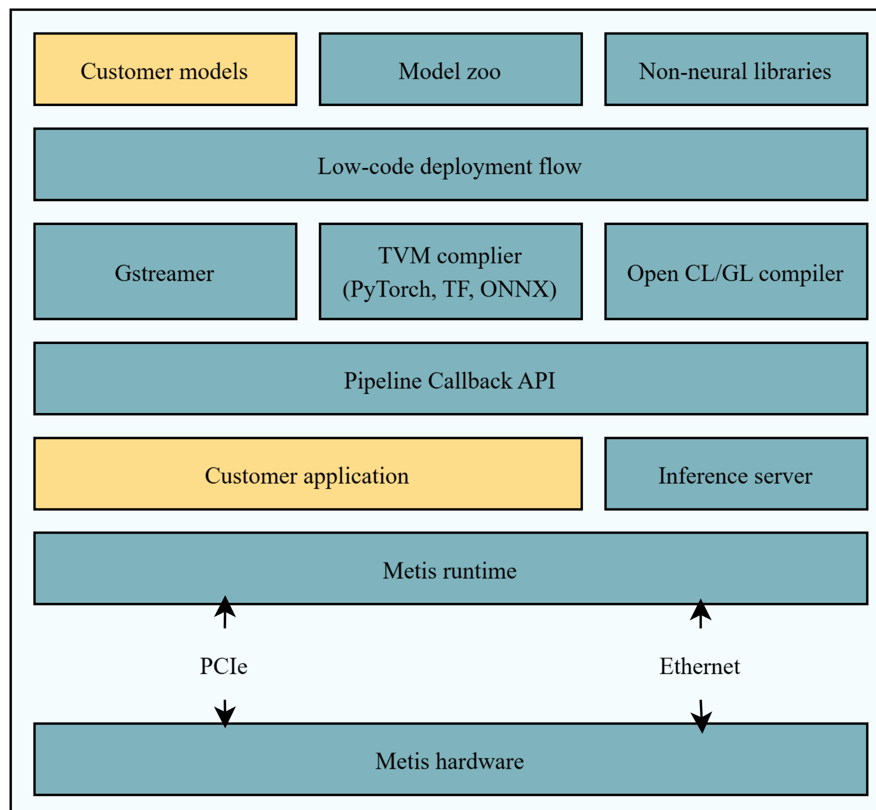


Figure 3.3: Overview of the layered architecture of the Voyager SDK.

3.2.1 Compilation and Deployment Flow

The deployment process begins with exporting a trained neural network to the *Open Neural Network Exchange (ONNX)* format, a framework-independent computational graph representation. This graph is translated by the Voyager compiler into a hardware-optimized executable tailored to the Metis AIPU.

During compilation, the model is quantized to INT8 precision in order to match the execution format supported by the Metis AIPU. Unlike some inference platforms where quantization is optional, the Metis accelerator executes neural networks exclusively in INT8 representation, making quantization a mandatory step of the deployment workflow. This conversion reduces memory usage and computational cost while enabling efficient execution on the accelerator architecture.

Although reducing numerical precision may introduce small accuracy differences compared to floating-point models, quantization procedures typically preserve most of the original model performance when appropriate calibration datasets are used. As a result, INT8 quantized models can achieve accuracy levels close to their floating-point counterparts while benefiting from improved inference efficiency on specialized hardware.

The resulting binary is then integrated into either a YAML-defined inference pipeline or directly loaded via the runtime APIs. This workflow abstracts low-level hardware details while constraining model execution to the supported precision formats and operator set of the accelerator.

3.2.2 SDK Components

The SDK comprises four primary components:

- **Neural Network Compiler:** Converts ONNX models into Metis-executable binaries. During compilation, graph-level optimizations are applied and models are quantized (typically to INT8 precision) to meet hardware constraints.

- **AI Pipeline Builder:** Constructs end-to-end inference pipelines from declarative YAML configurations, integrating preprocessing, inference, and postprocessing stages.
- **Runtime Libraries:** Including `AxRuntime`, `InferenceStream`, and `AxInferenceNet`, which provide different abstraction levels for hardware interaction and inference execution.
- **Model Zoo:** Provides pre-optimized network architectures and evaluation utilities for rapid deployment.

This modular structure separates model preparation, pipeline configuration, and runtime execution into distinct layers.

3.3 Execution Model and Abstraction Levels

The Voyager SDK supports multiple levels of integration, enabling applications to trade development simplicity for execution control.

3.3.1 Levels of Abstraction

Three primary abstraction levels are provided:

- **High-Level (Pipeline-Based Execution):** Inference pipelines are defined declaratively in YAML and executed through `InferenceStream`. Preprocessing, inference, and postprocessing are orchestrated internally by the SDK runtime.
- **Mid-Level:** `AxInferenceNet` enables applications to load compiled models and perform inference with explicit control over input submission and output retrieval.
- **Low-Level:** `AxRuntime` exposes lower-level primitives for memory management, scheduling, and device interaction, allowing fine-grained control over execution behavior.

This layered abstraction supports both rapid deployment in video-centric applications and tighter integration in embedded systems requiring deterministic control.

3.3.2 Execution Semantics

The default execution paradigm of the SDK is stream-oriented. Input data is processed within a continuous pipeline flow, and scheduling, buffering, and synchronization are managed internally by the runtime. This design reflects the SDK's focus on multi-camera and video-based inference workloads.

However, robotic middleware platforms such as ROS typically operates under a message-driven, frame-based execution model (see Section 2.1.4). In such systems, each incoming sensor frame triggers a callback-based processing routine, and results are expected to be produced deterministically per frame.

The difference between continuous pipeline-driven execution and discrete frame-triggered processing introduces integration challenges. While the SDK abstraction simplifies inference deployment, it centralizes scheduling and execution control within the runtime environment. Reconciling these execution semantics is therefore essential when integrating the accelerator into latency-sensitive robotic perception pipelines.

3.4 Architectural and SDK Implications for Robotic Integration

The Metis accelerator is architecturally suitable for embedded robotic perception workloads. Its inference-optimized design, moderate power consumption, and stable thermal behavior align with the constraints of autonomous platforms operating under limited computational and energy budgets. The M.2 form factor further supports integration into compact robotic systems.

Support for ONNX-based deployment ensures compatibility with common deep learning training frameworks, enabling a straightforward transition from model development to embedded inference.

From a software perspective, the Voyager SDK provides an automated workflow for compiling and deploying neural networks, including integrated preprocessing, inference, postprocessing, and quantization support. This enables efficient model optimization and reproducible deployment on the target hardware. However, the SDK follows a stream-based execution paradigm in which data is processed as a continuous pipeline. This design differs from the message-driven, frame-oriented architecture commonly used in robotic middleware such as ROS. Consequently, adaptation is required to reconcile the SDK's streaming execution model with the frame-by-frame semantics of ROS-based perception pipelines.

Chapter 4

Adaptation of the Axelera SDK for ROS Perception Pipeline

This chapter presents the integration of the Axelera Metis M.2 accelerator into a ROS-based perception pipeline and evaluates the resulting system-level behavior on an embedded Jetson Orin platform [22]. The proposed adaptation enables accelerator-backed inference to be invoked within the frame-triggered execution model of ROS perception nodes. To analyze the impact of host-side software architecture on the overall perception pipeline, two inference node implementations are developed in Python and C++. The evaluation investigates end-to-end latency, latency stability, pipeline stage behavior, and throughput under sustained input streams in order to characterize the performance of the accelerator-backed perception pipeline.

For the middleware implementation, this work uses ROS 2 (Humble distribution on Ubuntu 22.04). Unless stated otherwise, the term *ROS* in the remainder of this thesis refers to ROS 2.

The target workload is *object detection* using pretrained YOLO models described in Section 2.2.1. The integration goal is to invoke accelerator-backed inference in a frame-triggered manner consistent with the ROS perception execution model described in Section 2.1.4. The complete workspace and reference implementation are available in the accompanying code repository [23].

4.1 Problem Statement and Scope

As discussed in Section 3.2, the Voyager SDK employs a stream-oriented execution model, whereas ROS perception pipelines operate under frame-triggered processing semantics (Section 2.1.4). This difference introduces integration challenges when deploying accelerator-backed inference within ROS-based robotic systems.

The objective of this work is therefore not to modify the internal execution model of the accelerator runtime, but to adapt how the SDK runtime APIs are invoked within a ROS perception pipeline. In particular, the accelerator firmware, compiler toolchain, and vendor-provided runtime implementation remain unchanged.

Instead, the integration reorganizes inference invocation at the application level by transferring control-flow ownership from the runtime-managed streaming pipeline to a ROS-controlled execution flow. In this design, inference is triggered explicitly for each incoming sensor frame, aligning accelerator execution with the frame-based processing model of ROS perception pipelines.

4.2 Baseline SDK Execution Model

As discussed in Section 3.2, the Voyager SDK provides a software stack for compiling and executing neural network inference on the Metis accelerator. In typical usage, the SDK examples construct a continuous inference pipeline around a specified media source using a command-line application ("inference.py"):

```
./inference.py <network-or-model> <media>
```

In this mode, the pipeline owns the execution loop and runs continuously until the media source ends or execution is stopped. Scheduling, buffering,

and synchronization are handled internally by the SDK runtime rather than by an external application framework. This stream-oriented design is well suited for video analytics workloads, where frames are processed as part of a continuous execution graph.

The SDK examples implement this pipeline using a multimedia streaming framework (e.g., GStreamer) as described in the official Voyager SDK documentation [20]. Figure 4.1 summarizes the typical processing stages in this default stream-based pipeline.

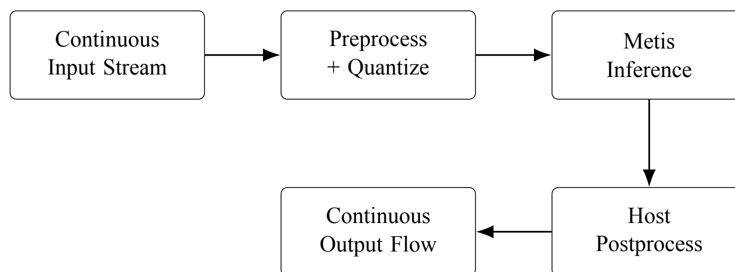


Figure 4.1: Simplified representation of the stream-oriented inference pipeline used by the Voyager SDK.

Figure 4.1 illustrates the default execution model provided by the SDK. In this configuration, the runtime owns the execution loop and processes input data continuously within a streaming pipeline. The ROS-based integration developed in this work replaces this runtime-managed execution with externally controlled, frame-triggered invocation, described later in Section 4.4.

4.3 Adaptation Goals

To integrate the accelerator coherently into a ROS perception pipeline, the following design constraints were defined:

- **Frame-triggered invocation:** inference must be triggered for each incoming `sensor_msgs/Image` message.
- **Bounded processing:** execution time per frame must remain bounded to prevent message backlog within the ROS pipeline.

- **Deterministic outputs:** each processed frame must produce a corresponding output message to preserve the semantics of the perception pipeline.

- **Predictable latency:** the integration must maintain stable per-frame processing latency to support responsive downstream planning and control modules.

- **Operational stability:** the node must support long-running execution without resource exhaustion or runtime failures.

4.4 Adaptation Strategy

The central adaptation consists of transferring control-flow ownership from the SDK-managed streaming pipeline to the ROS execution context. Instead of executing the vendor-provided CLI pipeline, inference is invoked explicitly for each received image frame within the ROS callback workflow.

The adapted integration follows these principles:

- Load the compiled accelerator model once during node initialization using the SDK runtime APIs.

- Reuse preallocated input and output buffers to avoid per-frame memory allocation overhead.

- Perform preprocessing of each incoming `sensor_msgs/Image` message to convert image data into the tensor format expected by the accelerator.

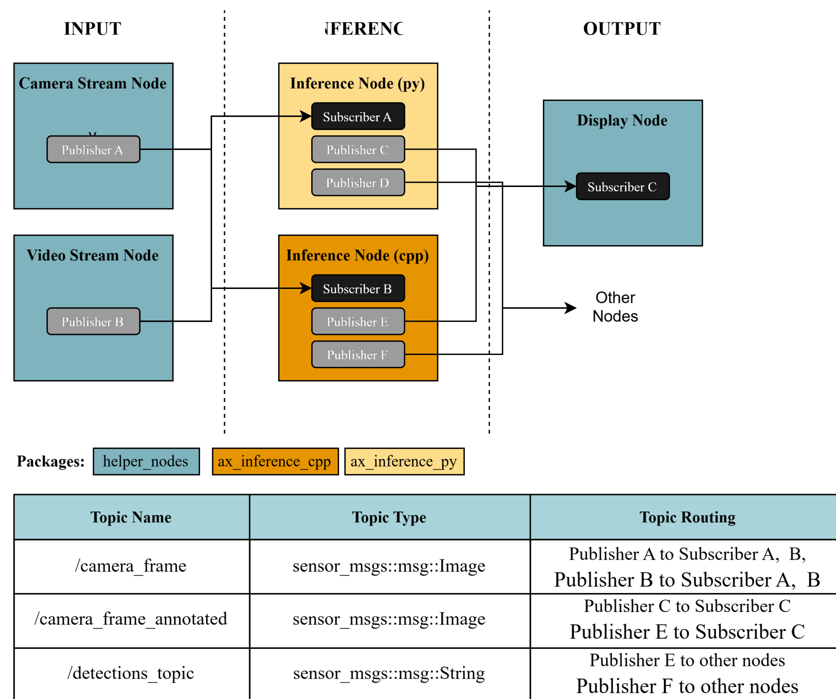
- Invoke accelerator inference explicitly for each frame and perform host-side postprocessing of the resulting model outputs.

- Maintain bounded processing per frame to preserve responsiveness under sustained input streams.

This restructuring modifies how the SDK runtime is invoked at the application level while leaving the vendor runtime, compiler toolchain, and hardware execution path unchanged.

4.5 ROS Workspace and Node Topology

A dedicated ROS workspace was created to evaluate the inference nodes in a controlled setting, independent of higher-level robot behavior. Figure 4.2 shows the evaluation topology. A camera/video publisher node emits image messages, the inference node performs accelerator-backed inference, and output nodes provide visualization and structured result consumption.



Topic Name	Topic Type	Topic Routing
/camera_frame	sensor_msgs::msg::Image	Publisher A to Subscriber A, B, Publisher B to Subscriber A, B
/camera_frame_annotated	sensor_msgs::msg::Image	Publisher C to Subscriber C Publisher E to Subscriber C
/detections_topic	sensor_msgs::msg::String	Publisher E to other nodes Publisher F to other nodes

Figure 4.2: ROS-based inference architecture. Input nodes publish image messages, which are consumed by either the Python or C++ inference node. Annotated frames and detection results are published to separate topics for visualization and downstream processing.

All configurable parameters—including model selection, accelerator core

count, input and output topics, preprocessing statistics, and detection thresholds—are exposed through ROS launch files. This enables reproducible experiments across different YOLO variants without modifying the source code.

4.6 Auxiliary Nodes and Workspace Modularity

The evaluation workspace includes auxiliary ROS nodes responsible for input generation and visualization. Input nodes publish image messages (`sensor_msgs/Image`) from prerecorded or live sources, while output nodes subscribe to annotated frames and structured detection messages.

This separation isolates accelerator execution from rendering and I/O overhead, ensuring that performance measurements reflect inference behavior rather than visualization costs. The modular structure follows ROS design principles (Section 2.1.4) and allows components to be replaced without modifying the inference node implementation.

4.7 Inference Interface and Processing Stages

Both inference nodes subscribe to an image topic publishing messages of type `sensor_msgs/Image` and publish (i) structured detection results (e.g., `std_msgs/String`) and (ii) an annotated image (of type `sensor_msgs/Image`). Preprocessing and postprocessing are kept consistent across implementations so that performance differences reflect runtime integration rather than algorithmic variation.

Per frame, the processing stages are:

- ROS image conversion and tensor formatting
- aspect-ratio preserving resize (letterbox) and normalization
- model-specific quantization and input preparation

- accelerator inference invocation
- output decoding and postprocessing (confidence filtering and NMS)

4.8 Inference Node Architecture

The inference node implements the interface between the ROS perception pipeline and the Axelera Metis accelerator. The node subscribes to an image topic publishing messages of type `sensor_msgs/Image` and publishes both annotated images and structured detection results.

Incoming image messages trigger the execution of a frame-based inference pipeline that performs preprocessing, accelerator invocation, postprocessing, and result publication. The same processing pipeline is implemented across both node variants used in this work in order to ensure consistent functionality and comparable performance behavior.

4.8.1 Common Inference Pipeline

Both inference nodes implement the same frame-triggered inference pipeline executed within a ROS 2 image callback. The pipeline defines the complete data path from image reception to detection publication and establishes the timing boundaries used for performance measurements.

When an image message arrives, the ROS 2 callback is invoked and processing of the frame begins. The incoming ROS image message is first converted into an OpenCV-compatible frame using the `cv_bridge` interface. The resulting frame is then passed to the preprocessing stage.

During preprocessing, the image is resized using aspect-ratio preserving scaling and letterbox padding to match the input resolution required by the neural network model. Pixel values are normalized according to the model parameters and quantized to the INT8 representation expected by the accelerator.

The quantized tensor is then submitted to the Axelera Metis accelerator using the Voyager runtime. Model execution occurs on the accelerator's AI Processing Unit (AIPU) and is invoked synchronously within the callback context, meaning that the callback blocks until inference results are returned. This design ensures that each frame is processed independently and that the measured latency corresponds to the complete processing time of a single frame. While asynchronous execution could potentially increase throughput, synchronous invocation simplifies timing analysis and preserves the deterministic frame-by-frame execution semantics expected in ROS perception pipelines.

After inference, the raw accelerator outputs are converted back to floating-point representation using the scale and zero-point parameters associated with each output tensor. The resulting tensors are then processed using an ONNX Runtime postprocessing graph that performs bounding box decoding and Non-Maximum Suppression (NMS).

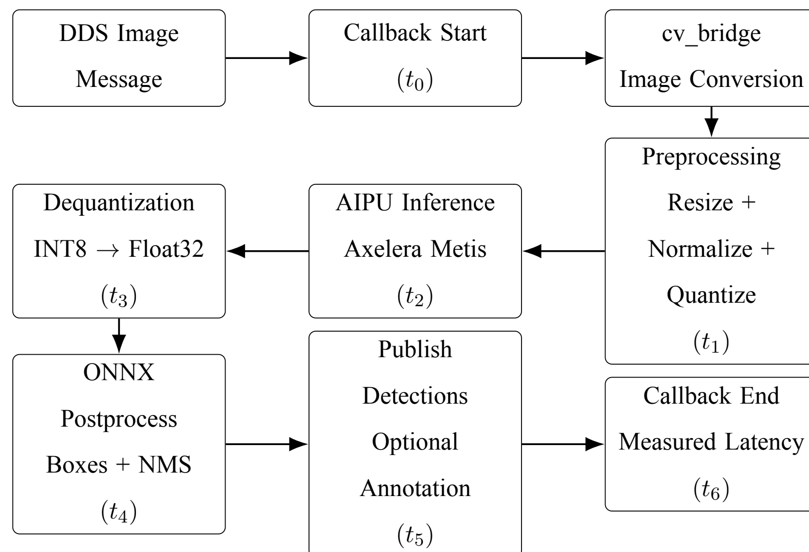


Figure 4.3: Frame-triggered inference pipeline executed within the ROS 2 image callback. Timing markers t_0-t_6 define the latency measurements used for performance evaluation.

Finally, the detected objects are converted into ROS-compatible messages and published to the output topic. Optionally, annotated images can also be generated for visualization. End-to-end inference latency is defined as the

interval between callback start and callback completion.

Whereas Figure 4.1 presents the baseline SDK pipeline, Figure 4.3 illustrates the adapted execution flow used in this work for frame-triggered ROS 2 inference and latency measurement.

4.8.2 Python Node Implementation

The Python inference node is implemented using the `rclpy` client library and interacts with the Axelera runtime through the Python bindings provided by the Voyager SDK. Numerical operations required by the pipeline are implemented using the NumPy library.

In this implementation, preprocessing and tensor manipulation are performed using vectorized NumPy operations applied to preallocated arrays. This approach simplifies implementation while leveraging optimized numerical routines from the Python scientific computing ecosystem.

Accelerator inference is invoked through the Python runtime interface, while postprocessing is executed using ONNX Runtime together with OpenCV utilities for Non-Maximum Suppression and detection filtering.

4.8.3 C++ Node Implementation

The C++ inference node is implemented using the `rclcpp` client library and interfaces with the Axelera runtime through its native C API. Compared with the Python implementation, the C++ version provides more explicit control over memory allocation, data layout, and runtime execution.

Preprocessing and tensor manipulation are implemented using explicit buffer operations, with input and output tensors allocated once during initialization and reused for subsequent frames. This design minimizes dynamic memory allocation and reduces host-side overhead during runtime execution.

Inference is executed by invoking the accelerator runtime with pointers to the prepared input buffers. Postprocessing is performed using ONNX Runtime

through its C++ API, followed by extraction of bounding boxes, class labels, and confidence scores before applying Non-Maximum Suppression.

4.8.4 Implementation Comparison

Both implementations execute the identical inference pipeline described in Section 4.7. The primary differences arise from the host-side programming environment and runtime interaction mechanisms.

Aspect	**Python Node**	**C++ Node**
ROS client library	`rclpy`	`rclcpp`
Runtime interface	Python SDK bindings	Native C API
Tensor operations	NumPy arrays	Explicit buffer operations
Memory management	Managed arrays	Preallocated buffers
Postprocessing	ONNX Runtime + OpenCV	ONNX Runtime C++ API
Implementation focus	Development flexibility	Runtime efficiency

Table 4.1: Comparison of the Python and C++ inference node implementations. Both nodes execute the same accelerator-driven inference pipeline while differing in host-side runtime interaction and memory management strategies.

These differences influence host-side computation overhead, memory behavior, and runtime efficiency. Evaluating both implementations under identical experimental conditions enables analysis of how host-side software architecture affects the performance of accelerator-based perception pipelines.

4.9 Experimental Setup

This section describes the hardware platform, experimental conditions, and performance metrics used to evaluate the ROS-based perception pipeline introduced in Chapter 4. The experiments focus on analyzing the latency behavior of the frame-triggered inference pipeline defined in Section 4.8 and evaluating how different host-side implementations influence its runtime performance.

4.9.1 Hardware Platform

All experiments were conducted on an embedded edge-AI platform based on the NVIDIA Jetson Orin system-on-module equipped with an external Axelera Metis M.2 AI accelerator. The accelerator performs neural network inference, while the Jetson host system executes preprocessing, postprocessing, and ROS 2 communication tasks described in Section 4.8. The accelerator is accessed through the Axelera Voyager SDK runtime.

Table 4.2 summarizes the main hardware components used in the experimental setup.

Table 4.2: Embedded inference platform hardware configuration.

Component	**Specification**
Host platform	NVIDIA Jetson Orin
CPU architecture	ARM64 (aarch64)
Operating system	Ubuntu 22.04 LTS
ROS distribution	ROS 2 Humble
AI accelerator	Axelera Metis M.2
Accelerator interface	M.2 (PCIe Gen3)
SDK	Axelera Voyager SDK v1.4
Postprocessing runtime	ONNX Runtime

4.9.2 Test Conditions

The experiments evaluate the two inference node implementations described in Section 4.8: a Python-based node and a C++ node implementation. Both nodes execute the identical accelerator-driven inference pipeline, enabling a controlled comparison of host-side implementation strategies under identical runtime conditions.

Four pretrained object detection models were used in the evaluation (Section 2.2.1): *YOLOv5n*, *YOLOv5s*, *YOLOv8n*, and *YOLOv8s*. These models were selected to represent different architectural families and model sizes. The *n* variants correspond to lightweight configurations optimized for low-latency

inference, while the s variants provide higher representational capacity at increased computational cost. This selection enables analysis of how model complexity influences the performance of the accelerator-backed perception pipeline.

Each node-model configuration was executed for five consecutive minutes while processing a continuous video stream containing multiple detectable objects per frame. The input stream was provided at 30 frames per second, ensuring a steady flow of incoming messages to the inference nodes. This duration allows the system to reach steady-state operation and provides a sufficiently large number of samples for meaningful latency analysis.

In frame-triggered perception pipelines, each incoming image message initiates a new processing cycle. To prevent message backlog in the ROS callback queue, the processing of each frame must complete before the arrival of the next frame. For an input stream operating at $f = 30$ FPS, the corresponding frame period is

$$T_{\text{frame}} = \frac{1}{f} = \frac{1}{30} \approx 33.3\,\text{ms}. \tag{4.1}$$

Consequently, to sustain real-time processing under the given input rate, the end-to-end perception latency must remain below approximately $33\,\text{ms}$.

All experiments were performed using identical runtime parameters across both implementations, including detection confidence thresholds, NMS thresholds, normalization parameters, and ROS topic configurations.

In total, eight experimental configurations were evaluated (two implementations $\times$ four models).

4.9.3 Metrics and Measurement Methodology

To evaluate the performance of the perception pipeline, detailed timing measurements were recorded for each processed frame. Timing points were inserted at key stages of the inference pipeline described in Section 4.8, allowing

stage-level latency analysis as well as measurement of the overall processing time.

The following latency metrics were collected:

- **Callback queue delay** – time between the image message timestamp and the start of the ROS callback.
- **Preprocessing time** – duration of image resizing, normalization, quantization, and preparation of the accelerator input tensor.
- **AIPU inference time** – execution time of the neural network model on the Axelera Metis accelerator.
- **Dequantization time** – conversion of accelerator output tensors from INT8 to floating-point representation.
- **ONNX postprocess time** – execution of the postprocessing graph and Non-Maximum Suppression.
- **End-to-end latency** – total processing time from callback start to detection publication.

In addition to timing measurements, the number of **detections per frame** was recorded to characterize the workload processed by the postprocessing stage.

During runtime, rolling statistics were maintained to estimate mean latency, standard deviation, and effective frame rate.

4.10 Performance Evaluation

This section analyzes the runtime behavior of the ROS-based perception pipeline under the experimental conditions described in Section 4.9. The evaluation examines the performance of the accelerator-backed inference pipeline

across multiple model configurations and compares the behavior of the two inference node implementations.

The analysis focuses on four aspects:

- End-to-end latency comparison between implementations
- Latency stability and variance
- Stage-level latency breakdown within the inference pipeline
- Throughput and queue behavior under continuous camera input

4.10.1 End-to-End Latency Comparison

Table 4.3 summarizes the measured end-to-end latency for both implementations across the evaluated YOLO models. End-to-end latency corresponds to the time interval between the start of the ROS callback and the publication of the detection results. Figure 4.4 visualizes the same measurements to illustrate the relative performance differences across model configurations.

Table 4.3: End-to-end latency comparison between Python and C++ inference nodes.

Model	Python (ms)	C++ (ms)	Speedup
YOLOv5n	73.50 ± 7.98	29.09 ± 1.15	$2.53\times$
YOLOv5s	73.70 ± 8.63	45.02 ± 2.62	$1.64\times$
YOLOv8n	57.90 ± 8.11	31.88 ± 1.21	$1.81\times$
YOLOv8s	60.61 ± 9.17	33.61 ± 1.46	$1.80\times$

The results show that the C++ implementation consistently achieves significantly lower end-to-end latency than the Python implementation. As illustrated in Figure 4.4, this behavior is observed across all evaluated models. Depending on the evaluated model, the observed speedup ranges from approximately $1.6\times$ to $2.5\times$.

The largest performance difference is observed for YOLOv5n, where the C++ implementation reduces the average processing latency from 73.50 ms to 29.09 ms. Similar trends are observed across the remaining models.

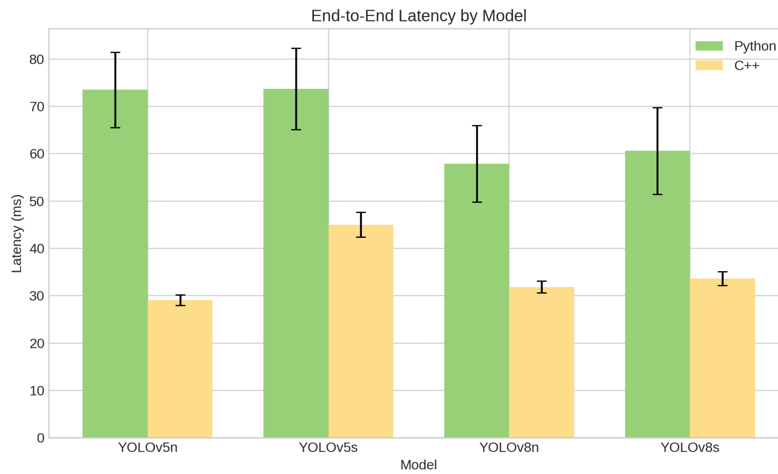


Figure 4.4: End-to-end latency comparison between Python and C++ implementations for each evaluated model. Error bars indicate one standard deviation.

Importantly, the execution time of the accelerator inference stage itself remains nearly identical across implementations. For example, YOLOv5n inference requires approximately 5.9 ms in C++ and 6.2 ms in Python. This indicates that the hardware execution path is identical in both cases. The observed performance difference therefore originates primarily from host-side processing overhead, including preprocessing, tensor handling, and postprocessing operations.

4.10.2 Latency Stability

In robotic perception pipelines, stable execution timing is as important as low average latency, since variability in perception latency can introduce jitter in downstream planning and control components.

Across all evaluated models, the C++ implementation exhibits significantly lower latency variance than the Python implementation. As illustrated in Figure 4.5, this difference is particularly visible for the YOLOv5n configuration. Similar stability improvements are observed for the remaining YOLO models, where the C++ implementation consistently maintains substantially lower standard deviation in end-to-end latency.

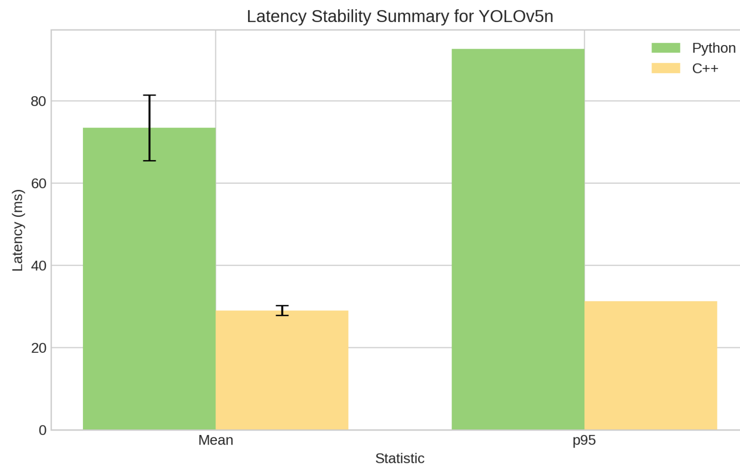


Figure 4.5: Latency stability summary for YOLOv5n. The mean and 95th percentile are shown for both implementations; error bars indicate one standard deviation.

YOLOv5n is shown as a representative configuration due to its low inference latency, which makes host-side variability more visible.

For this model, the standard deviation of the end-to-end latency is 1.15 ms in C++ compared to 7.98 ms in Python.

This difference reflects the higher runtime overhead associated with the Python execution environment. Python-based processing relies on higher-level abstractions such as NumPy arrays and dynamic memory allocation, which introduce additional variability in execution timing.

In contrast, the C++ implementation uses preallocated buffers and explicit memory management, reducing both runtime overhead and variance.

A notable observation is that the accelerator inference stage itself remains highly stable across all experiments. The standard deviation of the inference stage remains below approximately 0.5 ms for all evaluated models, indicating that the Metis AIPU provides highly predictable execution behavior. Most latency variability therefore arises from host-side processing and middleware scheduling rather than from the accelerator runtime.

4.10.3 Pipeline Stage Latency Breakdown

To better understand the origin of the observed latency differences, the execution time of the individual pipeline stages was analyzed.

Tables 4.4 and 4.5 summarize the stage-level latency for the YOLOv5n configuration in both implementations.

Table 4.4: Pipeline stage latency breakdown for the C++ inference node (YOLOv5n).

Stage	Latency (ms)
Preprocessing	5.99
AIPU inference	5.89
Dequantization	2.86
ONNX postprocessing (NMS)	13.45
Total	29.09

Table 4.5: Pipeline stage latency breakdown for the Python inference node (YOLOv5n).

Stage	Latency (ms)
Preprocessing	20.85
AIPU inference	6.21
Dequantization	4.87
ONNX postprocessing (NMS)	39.42
Total	73.50

The comparison reveals that the majority of the performance difference originates from host-side preprocessing and postprocessing operations rather than from the accelerator inference stage.

Two particularly important observations emerge.

First, preprocessing overhead is significantly larger in the Python implementation. Image preprocessing requires approximately 20-21 ms in Python compared to roughly 6 ms in C++. This difference is primarily due to the use of high-level Python numerical operations and additional memory copying within the Python runtime environment.

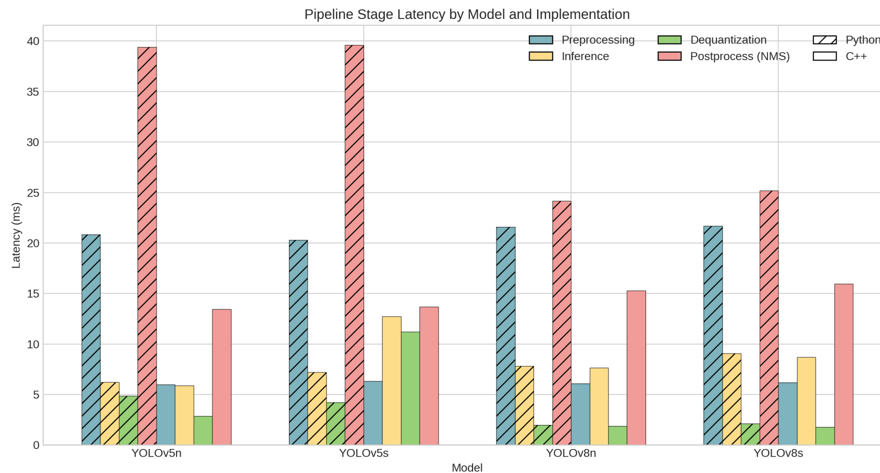


Figure 4.6: Pipeline stage latency comparison for each evaluated model and implementation. Colors represent pipeline stages, while hatched bars indicate the Python implementation.

Second, the most significant latency difference occurs during the ONNX-based postprocessing stage. For YOLOv5n, postprocessing requires approximately 39.4 ms in Python but only 13.5 ms in C++. This stage includes bounding box decoding and Non-Maximum Suppression (NMS), which are computationally intensive operations that benefit from the lower-level memory and loop control available in the C++ implementation.

These results demonstrate that even when inference is executed on dedicated AI hardware, the host-side processing pipeline remains a major determinant of overall system performance.

4.10.4 Throughput and Queue Behavior

In addition to latency, throughput measurements were collected to determine whether the perception nodes can sustain the incoming camera frame rate.

Because the inference nodes execute the perception pipeline synchronously within the ROS callback, each frame must complete the full preprocessing–inference–postprocessing sequence before the next frame can be processed. Consequently, the achievable throughput is directly determined by the end-to-end processing latency of the pipeline.

Table 4.6: Effective processing throughput of the inference nodes.

Model	Python FPS	C++ FPS
YOLOv5n	13.01	29.99
YOLOv5s	12.68	21.96
YOLOv8n	15.73	29.96
YOLOv8s	15.63	28.24

The C++ implementation achieves near camera frame rate for the lightweight models, reaching approximately 30 FPS for YOLOv5n and YOLOv8n. Even for the larger models, throughput remains above 20 FPS.

In contrast, the Python implementation processes frames at approximately 13-16 FPS, which is significantly below the input stream rate.

This throughput mismatch leads to noticeable differences in the ROS callback queue delay. When the node cannot process frames as quickly as they arrive, incoming messages accumulate in the ROS middleware queue before entering the callback. This behavior is clearly visible in the Python experiments, where queue delays exceed 300 ms, while the C++ implementation maintains queue delays below approximately 10 ms for lightweight models.

Despite this backlog, no frames were dropped in any of the experiments. This indicates that the ROS middleware successfully buffered incoming messages even when the processing node could not sustain the full source rate. However, large queue delays imply that the perception results correspond to stale sensor data, which can negatively affect downstream robotic control loops.

4.10.5 Discussion

The experimental results reveal several important characteristics of accelerator-backed perception pipelines operating within a ROS-based middleware architecture.

The perception pipeline was evaluated using an input stream operating at 30 frames per second, corresponding to a frame period of approximately

33.3 ms. To sustain real-time operation under these conditions, the end-to-end processing time of each frame must remain below this bound to prevent accumulation of messages in the ROS callback queue.

The experimental results show that the C++ implementation is capable of sustaining near real-time processing for the lightweight model configurations. In particular, the YOLOv5n and YOLOv8n models achieve average end-to-end latencies of 29.09 ms and 31.88 ms respectively, allowing the perception node to process frames at approximately the full input rate of 30 FPS. For the larger models, latency approaches or slightly exceeds the frame-period bound, reducing the available timing margin and increasing the risk that transient latency spikes may lead to queue accumulation and reduced effective throughput.

In contrast, the Python implementation exhibits substantially higher latency values, exceeding 57–73 ms depending on the evaluated model. Because these processing times are significantly longer than the frame period of the input stream, the perception node cannot process frames as quickly as they arrive. Consequently, incoming messages accumulate in the ROS middleware queue, leading to the large callback queue delays observed during the Python experiments.

Beyond this constraint analysis, the experiments highlight several important system-level observations.

First, the Axelera Metis accelerator provides stable and predictable inference performance. Across all experiments, inference latency remained within a narrow range and exhibited very low variance, confirming that the hardware runtime is well suited for embedded perception workloads.

Second, the overall performance of the perception pipeline is determined not only by accelerator execution but also by the efficiency of host-side processing. Preprocessing, tensor conversion, and postprocessing represent significant portions of the total latency budget and therefore strongly influence end-to-end system performance.

Third, the comparison between Python and C++ implementations demonstrates that host-side software architecture has a major impact on perception pipeline behavior. While Python provides development flexibility and rapid prototyping capabilities, the C++ implementation achieves substantially lower latency, reduced variance, and higher throughput.

Overall, the results confirm that the proposed frame-triggered integration strategy enables stable operation of the accelerator within a ROS-based perception pipeline. At the same time, the experiments show that once neural network inference is accelerated by dedicated hardware, the traditional inference bottleneck is largely removed and the dominant factors determining real-time perception performance shift to host-side preprocessing, data handling, and postprocessing stages within the perception pipeline.

Chapter 5

System-Level Validation

While Chapter 4 evaluated the performance of the accelerator-backed inference node in isolation, this chapter investigates its behavior when integrated into a complete robotic pipeline. The objective is to examine how the Axelera Metis M.2 accelerator operates within a ROS-based perception system when deployed as part of an end-to-end perception-driven manipulation workflow.

In the implemented system, perception outputs act as triggers for downstream components responsible for motion planning and execution. The implementation of the accelerator-backed inference node and its ROS 2 integration is publicly available in the accompanying project repository [23]. Consequently, the latency and reliability of the perception subsystem directly influence the responsiveness of the overall robotic pipeline. The validation scenario therefore embeds the accelerator-backed inference node within a closed-loop perception–planning–execution workflow, allowing detection events to propagate through the system and trigger subsequent actions.

The remainder of this chapter describes the validation environment and its individual components. First, the validation scenario and overall processing pipeline are introduced. The logical system architecture and hardware deployment are then presented, outlining how perception, planning, and simulation components are distributed across the platform.The remainder of this chapter describes the validation environment and its individual components. First, the

validation scenario and overall processing pipeline are introduced. The logical system architecture and hardware deployment are then presented, outlining how perception, planning, and simulation components are distributed across the platform. The perception subsystem is subsequently described through the task-specific object detection model used in the pipeline, which was fine-tuned for the manipulation scenario in order to generate reliable visual trigger events. The robotic simulation environment is then introduced as the execution platform for the planning and control components, enabling controlled evaluation of the perception–planning–execution loop. Finally, the experimental setup and system-level results are presented in order to analyze the behavior of the complete robotic pipeline.

5.1 Validation Scenario Overview

The validation scenario implements a perception-triggered robotic sorting pipeline in which visual object detections initiate a pick-and-place task performed by a simulated robotic arm.

The task is inspired by automated recycling systems in which objects must be sorted according to material category. Four object classes are considered: plastic, paper, glass, and metal. For the purposes of the sorting task, glass and metal share a common destination bin, resulting in three sorting bins in total. This simplified configuration loosely mimics real-world recycling processes, where certain material categories are collected together during initial sorting stages before further downstream separation. The perception system therefore identifies the material class of the observed object and triggers a corresponding pick-and-place action toward the appropriate bin.

The overall processing chain follows the structure illustrated in Figure 5.1.

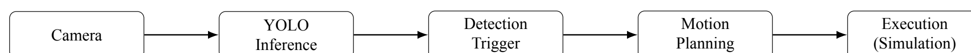


Figure 5.1: High-level perception-to-action pipeline used in the simulation experiments.

Visual input is captured by a fixed camera observing a predefined manipulation workspace. The spatial layout of this workspace, including the pickup region and the drop-off locations associated with the different material classes, is illustrated in Figure 5.2.

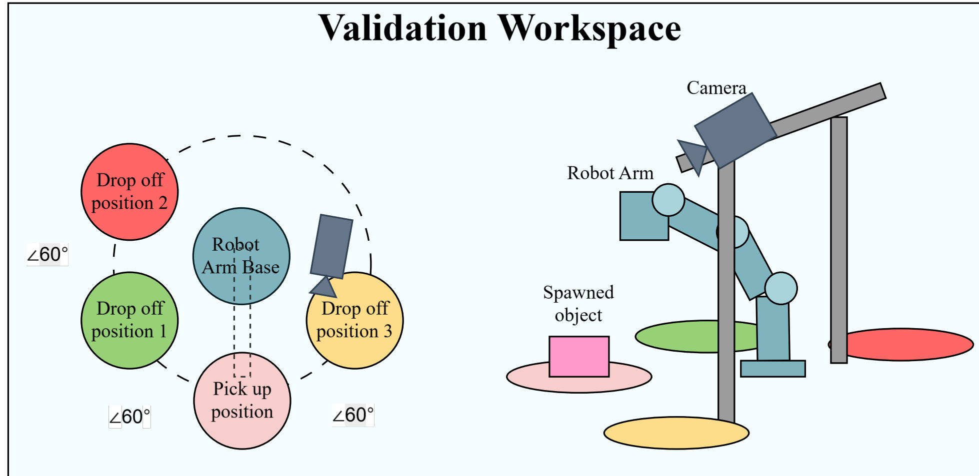


Figure 5.2: Layout of the simulated sorting workspace used during system validation. Objects appear in the pickup region and are transported by the robot arm to one of several drop-off locations depending on the detected material class.

The perception node processes the camera stream using YOLO inference executed on the Axelera Metis AI accelerator through the ROS inference node described in Sections 4.7 and 4.8. The node publishes detection results identifying the class of the observed object, which are interpreted as triggers for the manipulation pipeline.

Upon reception of a valid trigger, a pick-and-place routine is executed in which a simulated robotic arm retrieves an object from a predefined pickup location and moves it to a drop location corresponding to the detected material class. Motion planning is performed using the MoveIt framework [24], while execution takes place in a Gazebo-based robotic simulation environment.

The scenario intentionally simplifies several aspects of the perception problem in order to isolate the system-level integration behavior of the accelerator-backed inference pipeline. The camera remains fixed relative to the workspace,

and objects appear at a predefined pickup location within the field of view. Furthermore, the system assumes the presence of a single dominant object in the region of interest at any given time.

These constraints eliminate the need for full pose estimation or multi-object tracking and allow the evaluation to focus on reliable detection triggering and stable interaction between perception and downstream planning components.

5.2 Logical System Architecture and Data Flow

This section describes the logical organization of the validation scenario and the data flow through the robotic pipeline. The system architecture follows the perception–planning–control structure introduced in Section 2.1, with perception results acting as triggers for downstream manipulation behavior.

The validation scenario instantiates this architecture through three main functional components: a perception component responsible for generating detection triggers, a manipulation component that performs motion planning, and a simulation environment that executes the resulting trajectories. In addition to the perception node, a lightweight auxiliary node is used to handle image format conversion and visualization tasks. This separation ensures that rendering and image processing overhead do not affect the performance measurements of the accelerator-backed perception pipeline.

The resulting data flow forms a predominantly one-directional perception-to-action pipeline. Visual observations are first processed by the perception subsystem, which publishes detection results that act as triggers for the manipulation stage. The planning and execution components react to these perception outputs but do not provide feedback to the perception node itself. Consequently, the perception subsystem operates independently of the manipulation stage and continuously processes the incoming sensor stream, while downstream components respond asynchronously to detection events.

The system is distributed across two machines: (i) an embedded platform hosting camera acquisition and accelerator-backed inference; and (ii) a separate simulation host running the physics-based robotic environment, including the motion planning and manipulation components. Communication between both machines is realized through ROS 2 message interfaces over a wired Ethernet connection.

Figure 5.3 summarizes the logical system architecture and the data flow between the perception, planning, and control components deployed across the embedded platform and the simulation host.

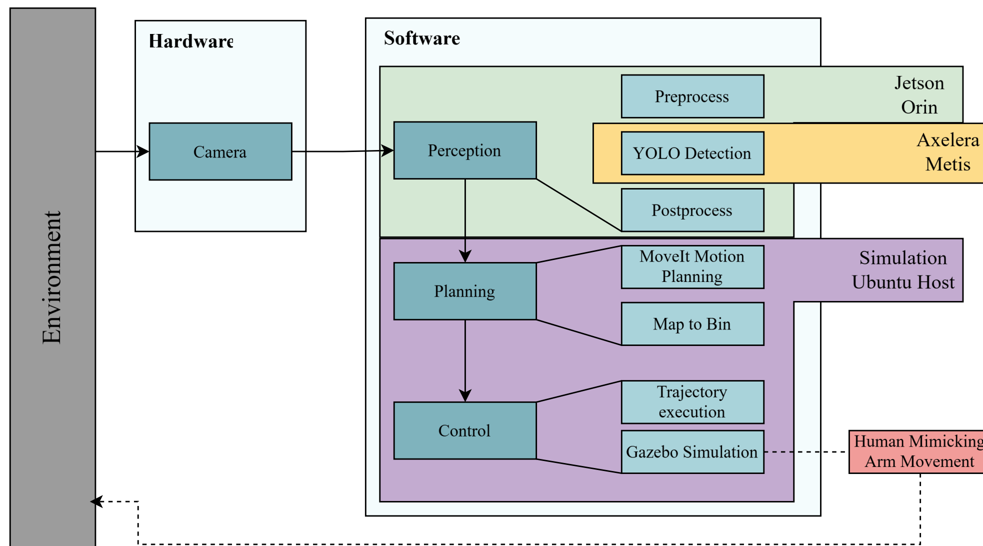


Figure 5.3: Logical system architecture of the validation scenario. The camera provides image frames to the perception pipeline, which runs on the embedded Jetson Orin platform. Preprocessing and postprocessing are executed on the host, while YOLO-based neural network inference is offloaded to the Axelera Metis M.2 accelerator. Stable detections trigger the downstream planning and control modules, and trajectory execution is carried out in the Gazebo-based simulation environment on a separate Ubuntu host.

5.2.1 Perception Pipeline and Trigger Generation

Visual input is provided by a USB camera connected to the embedded platform. An auxiliary node handles image conversion and visualization tasks

before forwarding the camera frames to the perception pipeline. The camera publishes image frames on a ROS topic of type `sensor_msgs/Image`. A perception node subscribes to this topic and performs accelerator-backed inference using the YOLO-based detector described later in Section 5.4.

The perception pipeline follows a sensor-driven execution model in which inference is triggered directly by the arrival of new camera frames. Inference is therefore executed for each image message received from the camera stream, allowing the perception subsystem to continuously analyze the incoming visual data. This design reflects the typical architecture of ROS-based perception pipelines, where perception modules operate as subscribers to sensor topics and process incoming data streams independently of downstream planning components.

Each processed frame is cropped to a predefined region of interest corresponding to the manipulation workspace in order to reduce unnecessary background processing and improve inference efficiency.

The detector internally produces bounding boxes and classification scores; however, the system-level interface reduces these outputs to a discrete class identifier. The detector distinguishes between four recyclable material categories (Paper, Plastic, Glass, and Metal), which correspond to predefined sorting targets within the manipulation pipeline.

A temporal stabilization mechanism is used to prevent transient false positives from triggering manipulation. Detection results are tracked across a short temporal window consisting of multiple consecutive frames and only transition to a *stable* state once sufficient consistency has been observed across these frames. This introduces a deliberate debounce delay prior to generating a manipulation trigger.

Once a detection becomes stable, the perception node publishes the corresponding class identifier on a ROS topic of type `std_msgs/msg/Int32`. To improve robustness against occasional message loss, the class identifier may be periodically re-published while the object remains stable in view.

5.2.2 Pick-and-Place Control and Planning

The manipulation component subscribes to the detection topic and initiates a sorting cycle whenever a valid class identifier is received. Motion planning is performed using MoveIt, an open-source software framework for robotic manipulation that provides planning scene management, kinematic solvers, and trajectory generation for robotic arms.

Upon reception of a detection trigger, the manipulation node executes a complete pick-and-place cycle consisting of the following stages:

1. spawning a box object of predefined size and weight in the simulation environment at a predefined pickup location,

2. planning a trajectory from the current robot state to a pickup configuration,

3. activating the simulated vacuum gripper,

4. planning a trajectory to a predefined drop configuration associated with the detected object class,

5. releasing the object and removing it from the simulation scene.

The drop location is determined by a predefined mapping between object classes and joint-space target configurations corresponding to different sorting bins. Although the goal configurations are predefined, motion planning is executed dynamically for each cycle, allowing MoveIt to generate a collision-aware trajectory from the robot's current state rather than replaying precomputed motions.

The resulting trajectory is forwarded to the ROS control interface responsible for translating motion plans into actuator commands. Within the validation environment, these commands are executed by the simulated robot through the control interfaces provided by the Gazebo simulation backend.

5.2.3 Simulation Execution

Execution of planned trajectories takes place within Gazebo, a physics-based robotic simulator that provides rigid-body dynamics, collision handling, and integration with ROS-based control interfaces [25]. The simulator runs on a separate host machine and acts as the execution backend of the manipulation pipeline.

The robotic arm exists only in simulation; no physical hardware is used in the validation scenario. Gazebo maintains the simulated environment and executes the planned trajectories through the configured ROS control interfaces.

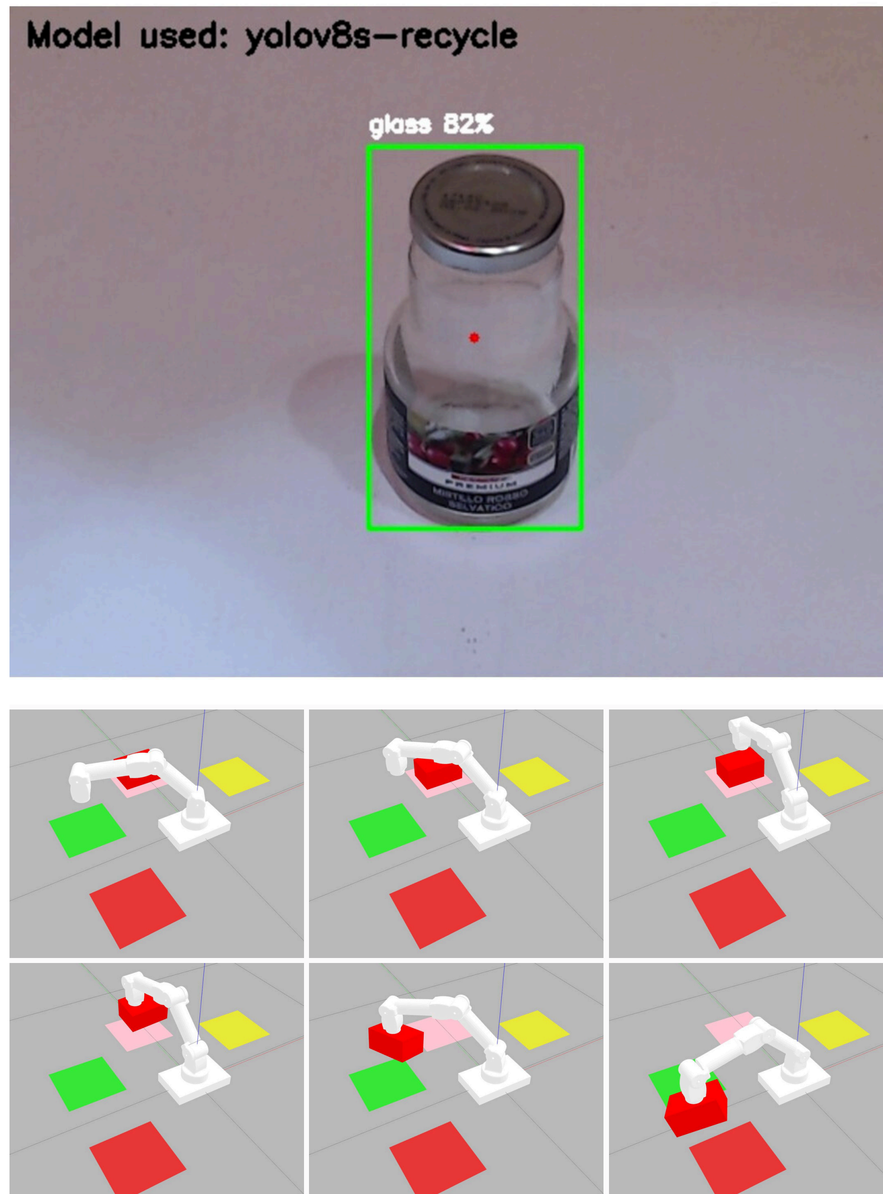


Figure 5.4: Example of the perception-driven manipulation pipeline during system validation. Top: detection output produced by the YOLOv8-based perception module identifying a recyclable glass object within the workspace. Bottom: sequence of robot motion in the Gazebo simulation environment demonstrating the pick-and-place procedure triggered by the detection result (read left to right, top row first).

5.3 Hardware Architecture

This section describes the physical deployment of the validation system and the hardware components used to realize the perception-driven robotic pipeline. The setup consists of an embedded platform responsible for camera acquisition and accelerator-backed perception, an external AI accelerator executing neural network inference, a USB vision sensor providing visual input, and a separate workstation hosting the robotic simulation and manipulation environment.

This distributed configuration reflects a typical embedded robotics setup in which perception is executed on resource-constrained edge hardware, while planning, simulation, and visualization tasks are offloaded to a separate host system.

5.3.1 Embedded Host Platform

The perception components of the validation pipeline execute on an NVIDIA Jetson Orin (4011 variant) running Ubuntu 22.04 LTS with ROS 2 Humble. The same platform was used previously for the inference evaluation described in Section 4.9. Within the validation scenario, the Jetson is responsible for camera acquisition, ROS message transport, preprocessing and postprocessing of image data, and coordination of inference execution on the external accelerator.

The host CPU manages image handling, tensor preparation, and interaction with the accelerator runtime, while neural network inference itself is executed on the external Metis accelerator.

5.3.2 AI Acceleration Module

Neural network inference is executed on an Axelera Metis M.2 AI accelerator connected to the Jetson through an M.2 HAT interface providing PCIe connectivity. The device operates using Voyager SDK version 1.4.

The Metis module serves as the dedicated inference backend of the perception pipeline, executing the YOLO-based detector used for object classification in the sorting scenario. Offloading inference to the accelerator allows the embedded host CPU to handle image acquisition, preprocessing, and ROS communication while the neural network computation is performed on specialized hardware.

5.3.3 Vision Sensor

Visual input is provided by a Logitech C922 Pro HD webcam connected to the Jetson via USB 3.0. The camera captures images at a resolution of 1920×1080 pixels at 30 frames per second. Captured frames are published to the perception pipeline described in Section 5.2, where they are processed by the accelerator-backed detection node.

5.3.4 Simulation Host

Robotic planning and simulation are executed on a separate workstation running Ubuntu 22.04. This machine hosts the Gazebo Classic simulation environment together with the MoveIt motion planning framework. Offloading simulation and planning tasks to a dedicated host prevents computational interference with perception processing on the embedded Jetson device.

Table 5.1: Hardware configuration of the simulation host.

Component	Specification
CPU	Intel Core i7-10750H
Memory	16 GB
GPU	NVIDIA Quadro T1000 Max-Q
Operating System	Ubuntu 22.04 LTS
Simulation Software	Gazebo Classic, MoveIt

Within the validation scenario, perception triggers generated on the Jetson

are transmitted over the network to the simulation host, where the manipulation node performs motion planning and generates collision-aware trajectories for the robotic arm. These trajectories are executed within the Gazebo environment through the configured ROS control interfaces, which update the simulated robot state and environment accordingly.

5.3.5 Network Topology

The Jetson Orin and the simulation host are connected via wired Ethernet. Both systems operate within the same ROS 2 domain and communicate using DDS-based peer-to-peer middleware.

Perception results generated on the embedded platform are transmitted to the simulation host, where they trigger motion planning and execution within the robotic environment. This distributed setup allows observation of latency propagation across the perception-planning-simulation pipeline during system-level validation.

5.4 Detection Model

Reliable perception is a prerequisite for perception-triggered robotic manipulation. In the validation pipeline described in Section 5.2, object detections serve as triggers that initiate motion planning and execution. Missed detections prevent manipulation cycles from being initiated, while unstable predictions increase latency due to the temporal stabilization mechanism applied to perception outputs. The choice of the object detection model therefore directly affects the reliability of the closed-loop system. This section describes the datasets, model selection process, and training refinements used to obtain the detector used in the validation scenario. The resulting trained model is subsequently compiled and deployed on the Metis accelerator as described in Section 5.3.2.

5.4.1 Datasets

Two publicly available datasets were used during model development to train and evaluate the detector.

Dataset 1

Dataset 1 is the *Garbage Detection – 6 Waste Categories* dataset, available on Kaggle [26]. It provides object detection annotations in YOLO format together with predefined training, validation, and test splits.

The original dataset contains six categories: *Biodegradable*, *Cardboard*, *Glass*, *Metal*, *Paper*, and *Plastic*. The *Biodegradable* class was removed prior to training to better align the dataset with the deployment scenario described in Section 5.2, which assumes a single dominant object in a controlled pickup region.

Images in the dataset may contain objects belonging to multiple categories. Consequently, the class distribution reported in Table 5.2 reflects the total number of annotated objects per category rather than the number of unique images. An image containing objects from multiple classes contributes to each corresponding category count. Table 5.3 therefore reports the actual number of images in each split together with the total number of annotated objects.

Representative samples from the dataset are shown in Figure 5.5.

Table 5.2: Class distribution for Dataset 1 after filtering.

Category	Train	Validation	Test	Total
Cardboard	3386	1292	20	4698
Glass	5429	2380	0	7809
Metal	3948	1360	533	5841
Paper	2981	33	1376	4390
Plastic	4146	214	1585	5945

Table 5.3: Dataset 1 split statistics after filtering.

Split	Images with Objects	Total Objects
Train	5864	19890
Validation	1481	5279
Test	1042	3514

Figure 5.5: Example images from Dataset 1 with ground-truth bounding boxes.

Dataset 2

Dataset 2 is the *Garbage Dataset Plus* dataset, also available on Kaggle [27]. Like Dataset 1, it provides YOLO-format annotations and can therefore be integrated directly into the training workflow.

The dataset defines a broader waste taxonomy including *Organic*, *Electronics*, and *Miscellaneous* categories. To align with the material-sorting objective of the manipulation scenario, only the recyclable classes *Paper*, *Plastic*, *Glass*, and *Metal* were retained.

As in Dataset 1, individual images may contain objects from multiple categories. The class distribution reported in Table 5.4 therefore corresponds to the number of annotated objects per category rather than the number of unique images. Images containing objects from multiple classes contribute to each relevant category count. Table 5.5 reports the actual number of images in each dataset split together with the total number of annotated objects.

Representative samples are shown in Figure 5.6.

Table 5.4: Class distribution for Dataset 2.

Category	Train	Validation	Test	Total
Glass	1139	136	67	1342
Metal	1954	531	262	2747
Paper	2567	306	146	3019
Plastic	2922	2177	1013	6112

Table 5.5: Dataset 2 split statistics.

Split	Images with Objects	Total Objects
Train	6237	8582
Validation	1895	3150
Test	981	1488

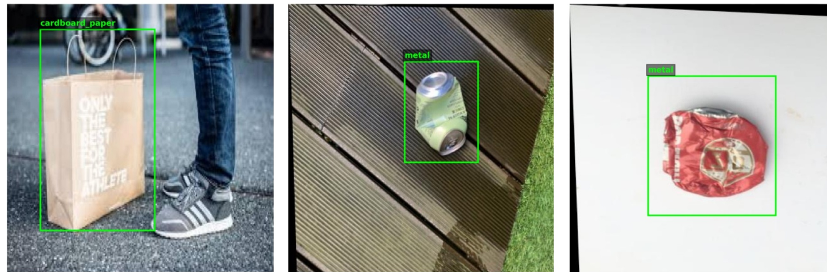


Figure 5.6: Example images from Dataset 2 with ground-truth bounding boxes.

5.4.2 Model Selection

The perception component requires spatial object detection rather than image classification, as the system must verify object presence and track detections across frames within the workspace region described in Section 5.2. For this reason, the detector must produce bounding boxes together with class predictions.

The detection backbone used in this work is based on the YOLO family of real-time object detection models introduced in Section 2.2.1 and evaluated in Chapter 4. YOLO architectures were selected due to their suitability for embedded perception and accelerator-backed deployment.

A total of four lightweight models were evaluated: YOLOv5n, YOLOv5s, YOLOv8n, and YOLOv8s. These models represent two generations of the

YOLO family and include nano (n) and small (s) variants suitable for deployment on resource-constrained systems. The selected models enable comparison across different model generations and network sizes while remaining compatible with accelerator deployment constraints.

All models were implemented using the Ultralytics framework and initialized with pretrained weights.

5.4.3 Training and Model Refinement

The detector development process consisted of two stages. First, multiple YOLO architectures were trained under identical conditions using Dataset 1 in order to compare baseline detection performance. Second, the best-performing architecture was refined through dataset integration and training adjustments aligned with the deployment scenario.

Training was performed in a cloud-based environment using Google Colab with an NVIDIA H100 GPU. Training hardware differs from the deployment platform, as inference performance is evaluated separately within the system-level validation experiments.

Table 5.6 summarizes the architectural comparison performed during the initial evaluation phase.

Table 5.6: Detector architecture comparison on Dataset 1

Model	Inference (ms/img)	mAP@0.5: 0.95	mAP@0.5	Precision	Recall
YOLOv8s	8.54	0.4268	0.5510	0.680	0.512
YOLOv8n	9.33	0.4058	0.5352	0.707	0.471
YOLOv5n	8.56	0.3998	0.5427	0.651	0.495
YOLOv5s	9.01	0.3982	0.5351	0.677	0.505

YOLOv8 variants consistently outperformed YOLOv5 models, and the small variants achieved higher accuracy than their nano counterparts. The best overall detection performance was achieved by YOLOv8s, which maintained real-time feasibility and was therefore selected for further refinement.

To improve detection robustness, Dataset 1 and Dataset 2 were merged after harmonizing label structures. The *Paper* and *Cardboard* classes were consolidated into a single *Paper* category to increase sample density and reduce class ambiguity.

In addition, several training adjustments were introduced to better reflect the deployment environment. Data augmentation strategies that create dense multi-object scenes were disabled, and training was extended to allow the model to converge on the expanded dataset.

5.4.4 Selected Detector Performance

Table 5.7 summarizes the aggregate performance improvements obtained after dataset integration and training refinement.

The final YOLOv8s detector used in the validation scenario is exported to the ONNX format and compiled for execution on the Axelera Metis accelerator using the Voyager SDK toolchain described in Section 3.2.1. During compilation, the network is quantized to INT8 precision in order to match the execution format supported by the Metis AIPU and improve inference efficiency. Although reducing numerical precision may introduce small differences compared to floating-point inference, quantization procedures typically preserve most of the original model accuracy, as discussed in Section 3.2.1. The resulting compiled model is deployed as the inference backend of the perception node described in Section 5.3.2.

Table 5.7: Detection performance before and after dataset integration

Metric	Baseline	Refined Model	Δ
Precision	0.680	0.780	+0.100
Recall	0.512	0.640	+0.128
mAP@0.5	0.5510	0.700	+0.149
mAP@0.5:0.95	0.4268	0.500	+0.073

The refined model shows consistent improvements across all evaluation metrics, with the largest relative gain observed in recall. From a system-level

perspective, improved recall directly reduces the probability of missed trigger events in the perception–planning pipeline and therefore increases the reliability of manipulation cycles within the validation scenario.

Confusion matrix comparison (Figure 5.7) further indicates reduced background misclassification and improved inter-class stability.

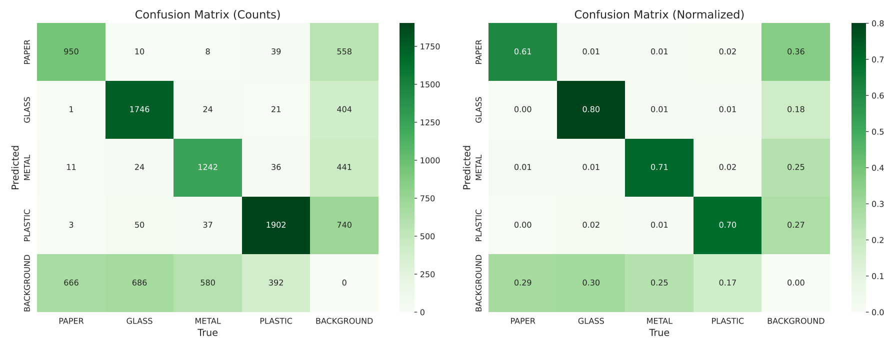


Figure 5.7: Confusion matrix for the refined detector model.

5.5 Simulation Environment

This section describes the robotic simulation environment used as the execution backend of the validation system. The environment is implemented in Gazebo Classic and integrated with ROS 2 and MoveIt 2. The simulation setup builds upon the open-source pick-and-place framework by Balaji [28], which was adapted and extended to match the system-level validation requirements of this work.

5.5.1 Robot Model and Kinematic Representation

The manipulator is a custom-designed four-degree-of-freedom robotic arm defined using URDF (Unified Robot Description Format), a standard XML-based representation used in ROS to describe robot kinematics, geometry, and physical properties. The model is generated through a Xacro macro file that parameterizes the robot structure and incorporates custom STL meshes for each link.

The arm consists of four revolute joints—*arm_base_joint*, *shoulder_joint*, *bottom_wrist_joint*, and *top_wrist_joint*—and a fixed intermediate connection between links. Inertial properties for the links were derived from CAD models and included in the URDF description to support physically consistent simulation.

For motion planning, the robot is additionally described using an SRDF (Semantic Robot Description Format) file, which defines planning groups and collision relationships used by MoveIt. In this configuration, a single planning group contains the four active joints of the arm, while collision checks between adjacent links are disabled to improve planning efficiency.

The validation scenario specifies manipulation targets directly in joint space rather than through Cartesian pose goals. This simplifies the motion generation process and isolates evaluation of the perception-triggered pipeline described in Section 5.2 without introducing additional pose-estimation complexity.

5.5.2 Simulation World and Object Modeling

The robot operates within a simplified Gazebo Classic simulation world designed to provide a controlled and reproducible environment for manipulation experiments. The conceptual layout of the workspace used in the experiments is illustrated in Figure 5.2, while the corresponding simulation environment in Gazebo Classic is shown in Figure 5.8. An example of the resulting manipulation execution during validation is presented in Figure 5.4.

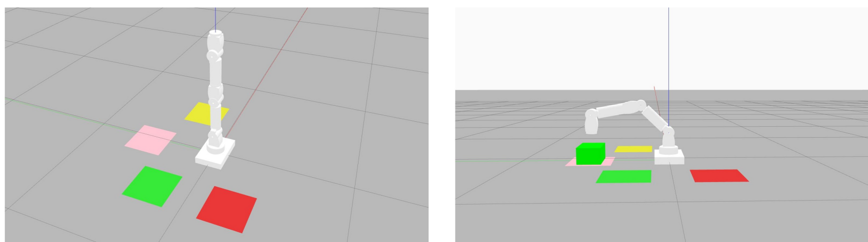


Figure 5.8: Gazebo Classic simulation environment used during system validation, showing the robotic arm, pickup region, and sorting bins.

The simulation environment consists of a fixed table structure, three drop bins corresponding to the material sorting categories, and box-shaped objects representing recyclable items. Objects are spawned at a predefined pickup location positioned within the robot's reachable workspace.

Each object is modeled as a simple box geometry with a color assigned according to the corresponding material class. This visual encoding allows the simulation environment to reflect the classification result of the perception pipeline while maintaining a lightweight object representation suitable for real-time simulation.

The pickup location remains fixed across experiments, while the drop location is selected from a set of predefined joint-space configurations corresponding to the available sorting bins.

5.5.3 End-Effector and Grasping Mechanism

Grasping is implemented using a vacuum-style gripper attached to the final link of the manipulator. The gripper is modeled using a Gazebo plugin that provides a binary activation interface allowing objects within a specified proximity to be attached to the end-effector.

When the gripper is activated, nearby objects are attached to the robot link through Gazebo's internal physics-based attachment mechanism. Deactivation releases the object, allowing it to be deposited into the appropriate drop location.

For simplicity, external objects are not added to the MoveIt planning scene. Consequently, motion planning remains collision-aware with respect to the robot itself but assumes free space relative to the environment. This design choice intentionally reduces environmental complexity while preserving the core perception–planning interaction required for system-level validation.

5.5.4 Trajectory Execution Interface

Trajectory execution is handled through the ROS 2 control framework, which connects the simulated robot model in Gazebo Classic to the motion planning interface used by MoveIt. Planned trajectories generated by the motion planner are forwarded to the robot control interface, where they are executed as joint-space motion commands by the simulated robot.

For each manipulation cycle, MoveIt generates a trajectory from the current robot state to the desired joint-space target configuration. The generated trajectory is then executed by the controller within the simulation environment, enabling physics-based motion consistent with the robot's kinematic model.

5.6 Experimental Setup

The system was evaluated under continuous operation in order to measure the behaviour of the complete robotic pipeline.

The experiments were conducted using the C++ perception node introduced in Chapter 4, which integrates the accelerator-backed inference pipeline into the ROS-based robotic architecture. This node forms the core component of the perception subsystem evaluated in the following experiments.

The perception pipeline was configured to operate at an input rate of 20 FPS, with each incoming frame processed by the inference node. This frame rate provides sufficient temporal resolution for the manipulation task while avoiding unnecessary processing of redundant frames during robot motion. In contrast to the accelerator-focused evaluation presented in Chapter 4, which examined high-frequency inference behaviour, the system-level validation processes every frame of the perception stream in order to observe how the perception pipeline interacts with the motion planning and manipulation components of the system.

The experiment was conducted over a runtime period of approximately

30 minutes with the full validation setup active. During this interval, the perception pipeline, motion planning subsystem, and Gazebo simulation environment operated continuously.

Performance metrics were collected and written to external log files during execution. These logs recorded processing counts and timing measurements throughout the experiment. Latency values reported in the results section were computed using both rolling measurement windows and cumulative statistics over the entire runtime. The rolling statistics allow observation of short-term variations in system behaviour, while the cumulative averages provide an overall summary of performance across the full duration of the experiment.

To maintain a continuous workload, objects were repeatedly presented within the camera field of view. During each manipulation cycle, the current object was manually removed from the scene as the robot approached the pickup stage, and a new object was placed in the workspace. This ensured that the perception pipeline remained continuously active while the manipulation subsystem executed repeated pick-and-place actions.

It should be noted that the experiment focuses on evaluating system-level behaviour and latency propagation within the perception–planning–execution pipeline. Detection accuracy of the trained model was evaluated separately during the detector development phase described in Section 5.4.

5.7 System-Level Metrics

To evaluate the behaviour of the integrated perception-manipulation pipeline, a set of system-level metrics was defined. These metrics extend the stage-level timing measurements introduced in Section 4.8, where detailed latency measurements were used to characterize the internal performance of the perception pipeline.

While the earlier measurements focused on the execution time of individual processing stages within the inference node, the system-level metrics defined in this section capture the behaviour of the complete robotic workflow. In particular, they characterize the throughput, latency, and temporal stability of the perception subsystem as well as its interaction with downstream manipulation execution.

For the system-level validation scenario, the perception pipeline receives images at an input rate of $f = 20$ frames per second. The corresponding frame period is

$$T_{\text{frame}} = \frac{1}{f} = \frac{1}{20} = 50\,\text{ms}. \tag{5.1}$$

As discussed in Section 1.2, stable frame-triggered execution requires that the end-to-end perception latency remains below the frame period so that each frame can be processed before the next one arrives. This constraint defines the upper bound for perception latency in the system-level validation scenario.

5.7.1 Perception Throughput

Perception throughput characterizes the rate at which the integrated perception pipeline processes visual input.

The following metrics were recorded:

- **Frames received** – The total number of image frames delivered by the camera during the experiment.
- **Frames processed** – The number of frames actually processed by the perception pipeline.
- **Processed fraction** – The ratio between processed frames and received frames, used to identify potential frame loss or backlog within the perception pipeline.

- **Inference throughput** – The effective processing rate of the perception pipeline, expressed in frames per second.

These metrics quantify the input load applied to the perception pipeline and the rate at which the integrated inference stage is executed during sustained operation.

5.7.2 Perception Pipeline Latency

Perception pipeline latency characterizes the time required to transform an input frame into a detection result. The latency measurements extend the stage-level metrics introduced in Section 4.8 by evaluating their behaviour under full system operation.

The following latency metrics were considered:

- **Callback queue delay**
- **Preprocessing time**
- **Inference stage latency**
- **Dequantization time**
- **ONNX postprocessing time**
- **Tracking state machine time**
- **End-to-end latency**

Together, these measurements describe both the internal structure of the perception pipeline and the overall time required to process an input frame.

5.7.3 Temporal Stability

The temporal stability of the perception pipeline was evaluated using rolling statistics recorded during runtime.

Two time-series metrics were observed:

- **Inference stage latency over time**
- **End-to-end perception latency over time**

These measurements are used to assess whether the integrated accelerator-backed pipeline maintains stable execution characteristics during sustained operation.

5.7.4 Detection Stability

To prevent transient detections from triggering manipulation actions, the perception pipeline includes a lightweight tracking state machine that filters unstable detections.

The following metrics quantify the behaviour of this stabilization stage:

- **Tracks created** – total number of candidate object tracks.
- **Tracks stabilised** – tracks that remained stable long enough to generate a detection event.
- **Stability rate** – ratio between stabilised tracks and created tracks.

These metrics characterize the robustness of the perception output before it is forwarded to the manipulation stage.

5.7.5 System Throughput and Manipulation Performance

The behaviour of the complete robotic workflow was evaluated using metrics related to manipulation execution and overall system throughput.

The following metrics were recorded:

- **Completed cycles** – number of successful pick-and-place operations executed during the experiment.
- **Manipulation cycle duration** – total time required to complete a single manipulation cycle.

- **Detections executed** – detections that resulted in a manipulation action.
- **Detections dropped** – detections ignored because the robot was already executing a manipulation cycle.

These measurements describe the behaviour of the manipulation stage and its influence on the effective throughput of the integrated system.

5.7.6 Overall System Timing

To identify the dominant performance constraint of the perception-manipulation pipeline, the timing contributions of the major system stages were compared.

The following aggregate metrics were considered:

- **Perception processing time** – end-to-end latency of the perception pipeline.
- **Communication latency** – delay between detection publication and manipulation start.
- **Manipulation cycle duration** – duration of the robotic execution stage.

These metrics enable identification of the subsystem that dominates the total execution time of the integrated robotic pipeline.

5.8 Results

This section presents the quantitative results obtained during the 30-minute system-level evaluation using the metrics defined in Section 5.7.

5.8.1 Perception Throughput

Table 5.8 summarizes the overall throughput of the perception pipeline. During the experiment, the pipeline received 36,041 input frames and processed

all of them, resulting in a processed fraction of 100%. The sustained inference throughput was approximately 19.92 FPS.

Table 5.8: Perception pipeline performance.

Metric	**Value**
Frames received	36,041
Frames processed	36,041
Processed fraction	100.0%
Inference throughput	19.92 FPS

These results show that the inference node processed every frame it received without frame dropping. Although the camera stream operates at 20 FPS, images are forwarded to the inference node through an intermediate ROS node that performs frame-to-frame forwarding. Minor scheduling and message-handling overhead in this stage slightly reduces the effective rate of frames delivered to the inference node. Consequently, the measured inference throughput of approximately 19.92 FPS closely matches the effective upstream frame rate observed during the experiment.

5.8.2 Perception Pipeline Latency

Table 5.9 reports the cumulative average latency of each stage of the perception pipeline. Figure 5.9 visualizes the same breakdown and highlights the relative contribution of each stage to the overall processing time.

Table 5.9: Perception pipeline latency breakdown.

Stage	**Latency (ms)**
Callback queue delay	3.67
Preprocessing	16.97
Inference stage	8.21
Postprocessing	17.52
End-to-end latency	**42.69**

The callback queue delay represents the time a frame spends waiting in the ROS callback queue before the perception callback begins execution. The measured average queue delay of approximately 3.67 ms is small relative to

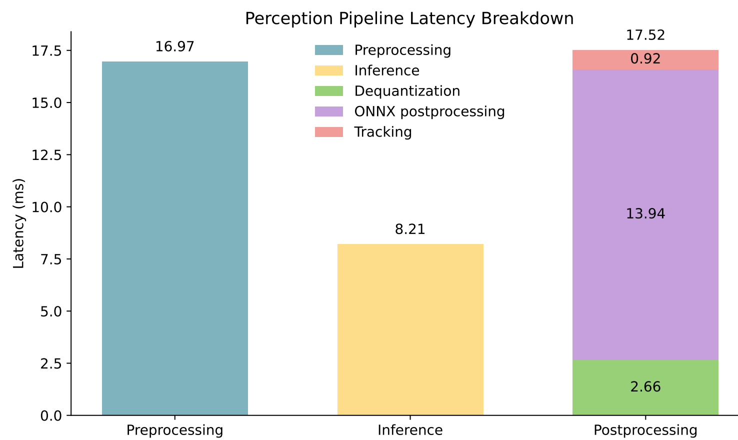


Figure 5.9: Latency breakdown of the perception pipeline. Host-side preprocessing and postprocessing dominate the total latency, while the inference stage contributes only a small fraction of the overall execution time.

the overall processing time and indicates that frames do not accumulate in the callback queue during sustained operation. This confirms that the perception node keeps up with the incoming frame stream without introducing backlog in the ROS execution pipeline. Since the queue delay occurs before the callback begins execution, it is not included in the end-to-end callback latency reported for the perception pipeline itself.

The inference stage contributes only a small fraction of the total pipeline latency. With an average execution time of 8.21 ms, it accounts for approximately 19% of the overall end-to-end callback latency. In contrast, host-side preprocessing and postprocessing together account for more than 80% of the total processing time. This behaviour is consistent with the observations reported in Chapter 4, where host-side processing stages were likewise identified as the main contributors to perception pipeline latency. The small differences in absolute latency values relative to the results reported in Chapter 4 are expected, since the system-level experiment includes task-specific preprocessing and postprocessing operations that differ from those used in the earlier evaluation. The latency of the inference stage itself, however, remains consistent across both experimental setups.

Finally, the measured end-to-end perception latency of 42.69 ms remains

below the frame period of 50 ms defined in Section 5.7. This confirms that the perception pipeline satisfies the real-time constraint required for frame-triggered processing, ensuring that each frame can be processed before the next one arrives.

5.8.3 Temporal Stability of the Perception Pipeline

To evaluate the stability of the perception pipeline during sustained execution, rolling latency statistics were recorded throughout the 30-minute experiment, with cumulative averages used as reference values. Figure 5.10 shows the evolution of the inference stage latency over time, while Figure 5.11 reports the corresponding end-to-end callback latency of the complete perception pipeline.

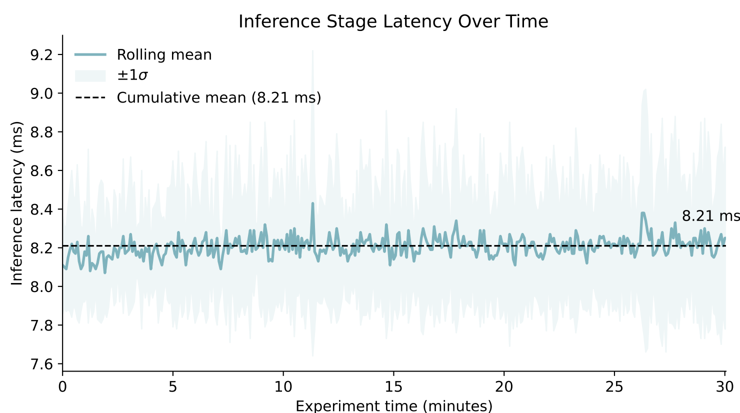


Figure 5.10: Inference stage latency during the 30-minute evaluation. The rolling mean remains stable throughout the experiment, with only minor short-term variation and no observable latency drift.

Both figures show stable behaviour over time. In Figure 5.10, the inference stage remains close to the cumulative mean value of approximately 8.21 ms throughout the evaluation, with only minor short-term fluctuations and no visible upward trend. Figure 5.11 shows the same behaviour at the level of the complete perception pipeline. A small deviation is visible during the initial portion of the experiment, where the latency is slightly lower before converging to its steady-state value. This behaviour can be attributed to

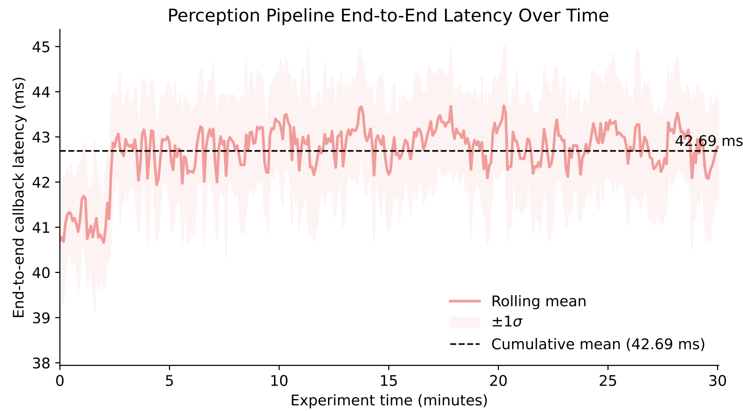


Figure 5.11: End-to-end callback latency of the perception pipeline during the 30-minute evaluation. After a brief startup transient, the latency remains stable for the remainder of the experiment.

a short initialization phase while the perception pipeline and its internal processing stages settle into steady-state operation. After this warm-up period, the end-to-end callback latency remains close to 42.69 ms for the remainder of the experiment. Overall, no latency drift or progressive slowdown is observed during the 30-minute run.

5.8.4 System Throughput and Manipulation Performance

The manipulation node executed 140 pick-and-place cycles during the experiment with no recorded failures, corresponding to a 100% task success rate. The average duration of a manipulation cycle was approximately 12.73 s. Since the robot can execute only one manipulation action at a time, detections that arrive while a cycle is in progress are ignored. As a result, the effective throughput of the complete system is determined primarily by the duration of the manipulation cycle rather than by the rate of perception output.

5.8.5 Overall System Bottleneck

A comparison of the dominant timing contributions across the perception–manipulation pipeline is provided in Table 5.10.

Table 5.10: End-to-end pipeline timing comparison.

Stage	Latency	Share of total
Perception processing	42.69 ms	0.34%
Manipulation cycle	12.73 s	99.66%

The results show that the perception pipeline contributes only a very small portion of the overall system cycle time. While the perception stage requires approximately 42.69 ms per frame, the manipulation phase dominates the execution time with an average duration of 12.73 s. Consequently, perception accounts for only about 0.34% of the total perception–manipulation cycle, whereas manipulation represents more than 99.6% of the execution time. This indicates that the proposed perception pipeline operates efficiently and does not constitute a performance bottleneck within the overall system.

5.9 Interpretation of System-Level Results

The system-level evaluation provides two key insights regarding the behavior of the integrated perception–manipulation pipeline.

First, the results demonstrate that the accelerator-backed perception pipeline operates reliably within the ROS-based system architecture. The latency measurements show that inference execution remains stable throughout the duration of the experiments, without observable drift or degradation during sustained operation. Furthermore, the measured end-to-end perception latency remains below the frame period defined in Section 5.7, confirming that the perception pipeline satisfies the real-time processing requirements of the system. These observations indicate that integrating the accelerator into the ROS perception pipeline does not introduce instability and that the system can maintain predictable processing behavior over extended periods of operation.

Second, the results reveal a clear separation between the time scales of perception processing and robotic manipulation. While perception processing operates on the order of tens of milliseconds per frame, the execution of

a complete pick-and-place cycle requires several seconds. As a result, the throughput of the overall robotic system is primarily determined by the duration of the manipulation cycle rather than by perception latency or neural network inference throughput.

The system-level behavior observed in this chapter is also consistent with the pipeline-level analysis presented in Chapter 4. As discussed in Section 4.8, a substantial portion of the perception latency originates from host-side processing stages, including preprocessing, tensor conversion, and postprocessing, rather than from the accelerator inference itself. The system-level experiments confirm that these host-side stages do not prevent the perception subsystem from maintaining real-time operation within the robotic pipeline.

Taken together, these observations indicate that the proposed integration successfully enables accelerator-backed perception within the robotic system without becoming the limiting component of the perception–manipulation loop. The accelerator provides sufficiently fast and stable inference execution to support the perception workload, while the dominant constraint on system throughput arises from the physical execution time of manipulation tasks.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

This thesis investigated the integration of the Axelera Metis M.2 edge AI accelerator into a ROS-based perception pipeline for embedded robotic systems. Rather than evaluating neural network inference performance in isolation, the work focused on the behavior of the complete perception pipeline, including preprocessing, accelerator invocation, postprocessing, and ROS message handling. The experimental evaluation presented in this thesis was conducted on an embedded platform based on an NVIDIA Jetson Orin host equipped with the Axelera Metis accelerator. In addition to pipeline-level evaluation, the system was validated within a perception-driven robotic manipulation scenario in order to analyze how the perception subsystem behaves when integrated into a larger perception–planning–control architecture.

A central challenge addressed in this work is the mismatch between the execution model assumed by accelerator software development kits and the execution semantics commonly used in robotics middleware. The Axelera Voyager SDK is primarily designed for stream-oriented processing pipelines, whereas ROS perception systems operate on discrete frame-triggered callbacks driven by sensor messages. The integration strategy developed in this

thesis demonstrates that this mismatch can be resolved without modifying vendor runtime internals by invoking accelerator inference synchronously within the ROS callback context. This approach preserves the frame-based processing semantics expected by ROS while maintaining compatibility with the accelerator runtime.

The experimental results show that the accelerator can execute neural network inference with stable and predictable timing when invoked within this frame-triggered ROS callback structure. Predictable execution behavior is particularly important in robotic systems, where perception modules must operate with bounded latency to support reliable downstream decision-making and control. Across repeated experiments, the accelerator consistently processed incoming frames without introducing irregular timing behavior or message backlog within the ROS execution framework.

At the same time, the evaluation demonstrates that accelerator inference time alone does not determine the overall performance of the perception pipeline. The measured end-to-end latency includes contributions from several additional stages, including image preprocessing, tensor conversion, host-side buffer handling, and postprocessing operations such as bounding box decoding and non-maximum suppression. Once neural network inference is accelerated by dedicated hardware, these host-side stages constitute the dominant portion of the perception latency. This observation highlights an important system-level insight: successfully accelerating neural network execution removes the traditional inference bottleneck, shifting the primary performance constraints to the surrounding processing stages executed on the host processor.

The comparison between Python and C++ inference node implementations further illustrates the influence of host-side software architecture on perception pipeline behavior. Although both implementations execute the same inference pipeline and interact with the same accelerator runtime, the C++ node consistently achieves lower end-to-end latency and reduced execution variance. This result indicates that the choice of host-side runtime environment

can significantly affect the performance characteristics of accelerator-backed perception systems. While the present work does not isolate the precise causes of this difference, the results suggest that runtime overhead, memory management behavior, and execution characteristics of the host environment play a significant role in determining overall pipeline performance.

Finally, the system-level validation scenario demonstrates that the integrated perception pipeline can operate reliably within a complete robotic manipulation stack. In the evaluated pick-and-place scenario, the perception subsystem provides stable detection outputs that support downstream motion planning and manipulation tasks. The overall task execution time is primarily dominated by the physical execution of manipulation actions rather than perception latency, indicating that the perception subsystem satisfies the requirements of the evaluated application while maintaining predictable system behavior.

Taken together, the results of this work demonstrate that stream-oriented edge AI accelerators such as the Axelera Metis can be effectively integrated into frame-triggered robotic middleware through appropriate system-level adaptation. More broadly, the findings emphasize that the performance of AI accelerators in robotic systems must be evaluated within complete perception pipelines rather than through isolated neural network inference benchmarks, since host-side processing and middleware behavior ultimately determine the achievable end-to-end system performance.

6.2 Future Work

While the results presented in this thesis demonstrate the feasibility of integrating the Axelera Metis accelerator into a ROS-based perception pipeline, several directions remain for further investigation and improvement. In particular, future work may focus on optimizing the current execution pipeline,

improving software interfaces between accelerator runtimes and robotics middleware, and extending the evaluation to more realistic deployment scenarios.

6.2.1 Optimization of the Inference Pipeline

The integration presented in this thesis focuses on establishing a stable frame-triggered execution model for the Axelera accelerator within a ROS-based perception pipeline. While the current implementation demonstrates consistent runtime behavior, several stages of the pipeline may still introduce avoidable overhead. These include preprocessing, host-accelerator communication, buffer management, and postprocessing steps.

While this thesis already includes a stage-level latency analysis of the perception pipeline, more detailed profiling could further identify optimization opportunities within host-side preprocessing, tensor handling, and postprocessing operations. Future work could therefore investigate the current execution flow in greater detail in order to determine whether improvements are possible through more efficient buffer reuse, reduced memory transfers, or improved coordination between host-side processing and accelerator execution.

6.2.2 Frame-Oriented Execution Support within the SDK Workflow

The integration strategy developed in this thesis adapts a stream-oriented accelerator SDK to operate within the frame-triggered execution model commonly used in ROS-based perception pipelines. This is achieved without modifying the vendor runtime and instead relies on reorganizing how the existing SDK interfaces are invoked.

Future work could formalize this approach by developing a reusable frame-oriented inference interface built on top of the existing SDK components. Such an interface could simplify integration of the accelerator into robotics middleware that relies on event-driven execution models and could improve

the usability of accelerator platforms within robotics software stacks.

6.2.3 Evaluation on Physical Robotic Systems

The system-level validation presented in this thesis uses a real camera input but performs the manipulation task within a simulated robotic environment. While this setup enables controlled observation of perception-triggered behavior, further evaluation on a physical robotic platform would provide additional insight into real-world deployment conditions.

Future work could therefore extend the validation scenario to a system that includes real robotic actuators and physical manipulation tasks. Such experiments would allow analysis of the interaction between perception, planning, and execution under practical operating conditions, including the effects of sensor noise, environmental variability, and hardware-level timing constraints.

6.2.4 Energy Efficiency Analysis

While this thesis primarily focuses on system integration and latency behavior, edge AI accelerators are typically designed to improve performance per watt compared to general-purpose computing platforms. Future work could therefore extend the evaluation to include power consumption measurements and energy efficiency analysis.

Such experiments would provide additional insight into the trade-offs between computational performance, energy consumption, and system-level efficiency when deploying AI accelerators in embedded robotic systems. Evaluating these aspects would contribute to a more comprehensive understanding of how accelerator-based perception pipelines behave under realistic deployment constraints.

Bibliography

[1] Y. Maruyama, S. Kato, and T. Azumi, “Exploring the performance of ROS2,” 10 2016, pp. 1–10, doi:10.1145/2968478.2968502.

[2] P. Iñigo-Blasco, F. D. del Rio, F. Vicente-Diaz, A. C. Jimenez, and F. Candelas, “Robotics software frameworks for multi-agent robotic systems development,” *Robotics and Autonomous Systems*, vol. 60, no. 6, pp. 803–821, 2012, doi:10.1016/j.robot.2012.02.004.

[3] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, “ROS: An open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, 2009, p. 5.

[4] G. Pardo-Castellote, R. White, and S. Diego), “Leveraging DDS security in ros2,” 09 2018, doi:10.36288/ROSCon2018-900839.

[5] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022, doi:10.1126/scirobotics.abm6074.

[6] S. D. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. H. Eng, D. Rus, and M. H. Ang, “Perception, planning, control, and coordination for autonomous vehicles,” *Machines*, vol. 5, no. 1, p. 6, 2017, doi:10.3390/machines5010006.

[7] C. Steger, M. Ulrich, and C. Wiedemann, *Computer Vision in Robotics and Industrial Applications*. Springer, 2011.

[8] D. Sculley, G. Holt, D. Golovin *et al.*, "Machine learning systems: Development, deployment and operations," in *Machine Learning Engineering*. Wiley, 2023.

[9] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," pp. 2704–2713, 2018, doi:10.1109/CVPR.2018.00286.

[10] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[11] G. Jocher, "YOLOv5," GitHub repository, 2020, accessed: Mar. 26, 2026. [Online]. Available: https://github.com/ultralytics/yolov5

[12] G. Jocher, A. Chaurasia, and J. Qiu, "YOLOv8," GitHub repository, 2023, accessed: Mar. 26, 2026. [Online]. Available: https://github.com/ultralytics/ultralytics

[13] S. M. Neuman, B. Plancher, B. P. Duisterhof, S. Krishnan, C. Banbury, M. Mazumder, S. Prakash, J. Jabbour, A. Faust, G. C. de Croon *et al.*, "Tiny robot learning: Challenges and directions for machine learning in resource-constrained robots," in *2022 IEEE 4th international conference on artificial intelligence circuits and systems (AICAS)*. IEEE, 2022, pp. 296–299, doi:10.1109/AICAS54282.2022.9869967.

[14] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, "Efficient processing of deep neural networks: A tutorial and survey," *Proceedings of the IEEE*, vol. 105, no. 12, pp. 2295–2329, 2017, doi:10.1109/JPROC.2017.2761740.

[15] A. Reuther, P. Michaleas, M. Jones, V. Gadepally, S. Samsi, and J. Kepner, *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, 2021, doi:10.1109/HPEC49654.2021.9622867.

[16] S. M. Ahsan, T. Hoque, M. S. Hasan, M. Chowdhury, and A. Dhungel, "Hardware accelerators for artificial intelligence," in *AI-Enabled Electronic Circuit and System Design: From Ideation to Utilization*. Springer, 2025, pp. 497–535.

[17] NVIDIA, "Jetson orin platform documentation," Online documentation, 2023, accessed: Mar. 26, 2026. [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-orin/

[18] Google Coral, "Edge tpu documentation," Online documentation, 2023, accessed: Mar. 26, 2026. [Online]. Available: https://coral.ai/docs/edgetpu/

[19] Axelera AI, "Axelera metis platform documentation," Online documentation, 2025, accessed: Mar. 26, 2026. [Online]. Available: https://www.axelera.ai

[20] ——, "Voyager SDK," GitHub repository, 2025, accessed: Mar. 26, 2026. [Online]. Available: https://github.com/axelera-ai-hub/voyager-sdk

[21] P. A. Hager, B. Moons, S. Cosemans, I. A. Papistas, B. Rooseleer, J. Van Loon, R. Uytterhoeven, F. Zaruba, S. Koumousi, M. Stanisavljevic *et al.*, "11.3 Metis AIPU: A 12nm 15TOPS/w 209.6TOPS SoC for

cost- and energy-efficient inference at the edge," pp. 212–214, 2024, doi:10.1109/ISSCC49657.2024.10454395.

[22] NVIDIA, "Nvidia jetson orin series technical overview," Technical documentation, 2023. [Online]. Available: https://developer.nvidia.com

[23] J. Nikolovska, "ROS pick-and-place waste sorting," GitHub repository, 2026, accessed: Mar. 26, 2026. [Online]. Available: https://github.com/jananikolovska/ROS-pick-and-place-waste-sorting

[24] S. Chitta, I. Sucan, and S. Cousins, "Moveit![ROS topics]," vol. 19, 03 2012, pp. 18–19, doi:10.1109/MRA.2011.2181749.

[25] N. Koenig and A. Howard, "Gazebo: A 3d dynamic simulator for robot environments," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2004.

[26] "Garbage detection – 6 waste categories," Kaggle dataset, 2020, accessed: Mar. 26, 2026. [Online]. Available: https://www.kaggle.com/datasets/viswaprakash1990/garbage-detection

[27] "Garbage dataset plus: Yolo waste detection dataset," Kaggle dataset, 2024, accessed: Mar. 26, 2026. [Online]. Available: https://www.kaggle.com/datasets/engrbasit62/garbage-dataset-plusyaml

[28] S. Balaji, "Pick and place using ROS2, MoveIt2 and Gazebo," GitHub repository, 2023, accessed: Mar. 26, 2026. [Online]. Available: https://github.com/santoshbalaji/pick_and_place

Acknowledgements

I would first like to express my sincere gratitude to my thesis supervisor, Professor Davide Rossi, for his guidance and valuable insights throughout the development of this thesis. I also thank the teaching staff of the University of Bologna for inspiring my curiosity and deepening my passion for this field.

I extend my gratitude to Axelera AI for providing such an innovative and supportive environment. I am especially grateful to my manager, Cristian Garjitzky, for encouraging me to share my ideas and helping refine them through valuable feedback. I also thank the entire team, whose knowledge, enthusiasm, and commitment I greatly admire and hope to carry into my own work.

Finally, I would like to thank the people who stood beside me throughout this experience:

To my friend Megi, who I cherish as a sister, for being the perfect companion throughout this challenge.

To my parents, who are and will always be my role-models, my brothers, and my entire family for encouraging me to reach for the stars and always believing in me.

To my friends in Skopje, Bojana, Liki, Nikolina, who I hope to have by my side not only in this lifetime but in every other as well.

To my friends in Bologna, who made Bologna feel like home and filled a missing puzzle piece I did not know I had before coming here.

And finally, to Eno, for every single thing he does and for always inspiring me to become the best version of myself.

I would not have made it this far without your love and support.