

Handling with AI-enhanced Robotic
Technologies for flexible ManUfacturing

D2.1- HARTU Architectural specification and integration plan















Deliverable ID:	D2.1
Project Acronym:	HARTU
Grant:	101092100
Call:	HORIZON-CL4-2022-TWIN-TRANSITION-01
Project Coordinator:	TEKNIKER
Work Package	WP2
Deliverable Type	Document, report
Responsible Partner:	Engineering Ingegneria Informatica S.p.a. (ENG)
Contributors	TEK, AIMEN, DFKI, ITRI
Edition date:	22 December 2023
Version:	10
Status:	Final
Classification:	PU



This project has received funding from the European Union's Horizon Europe - Research and Innovation program under the grant agreement No 101092100. This report reflects only the author's view and the Commission is not responsible for any use that may be made of the information it contains.

HARTU Consortium

HARTU “Handling with AI-enhanced Robotic Technologies for flexible manufactUring” (Contract No. 101092100) is a collaborative project within the Horizon Europe – Research and Innovation program (HORIZON-CL4-2022-TWIN-TRANSITION-01-04). The consortium members are:

1		FUNDACION TEKNIKER (TEK)	Contact: Iñaki Maurtua inaki.maurtua@tekniker.es
2		DEUTSCHES FORSCHUNGSZENTRUM FUER KUENSTLICHE INTELLIGENZ GMBH (DFKI)	Contact: Dennis Mronga dennis.mronga@dfki.de
3		ASOCIACIÓN DE INVESTIGACIÓN METALÚRGICA DEL NOROESTE (AIMEN)	Contact: Jawad Masood jawad.masood@aimen.es
4		ENGINEERING INGEGNERIA INFORMATICA S.P.A. (ENG)	Contact: Riccardo Zanetti riccardo.zanetti@eng.it
5		TOFAS TURK OTOMOBIL FABRIKASI ANONIM SIRKETI (TOFAS)	Contact: Nuri Ertekin nuri.ertekin@tofas.com.tr
6		PHILIPS CONSUMER LIFESTYLE BV (PCL)	Contact: Erik Koehorst erik.koehorst@philips.com
7		ULMA MANUTENCION S. COOP. (ULMA)	Contact: Leire Zubia lzubia@ulmahandling.com
8		DEEP BLUE Srl (DBL)	Contact: Erica Vannucci erica.vannucci@dblue.it
9		FMI HTS DRACHTEN B.V. (FMI)	Contact: Floris goet floris.goet@fmi-improvia.com
10		TECNOALIMENTI S.C.p.A (TCA)	Contact: Marianna Faraldi m.faraldi@tecnoalimenti.com
11		POLITECNICO DI BARI (POLIBA)	Contact: Giuseppe Carbone giuseppe.carbone@poliba.it
12		OMNIGRASP S.r.l. (OMNI)	Contact: Vito Cacucciolo vito.cacucciolo@omnigrasp.com
13		INDUSTRIAL TECHNOLOGY RESEARCH INSTITUTE INCORPORATED (ITRI)	Contact: Curtis Kuan curtis.kuan@itri.org.tw
14		INFAR INDUSTRIAL Co., Ltd (INFAR)	Contact: Simon Chen simon@infar.com.tw

Document history

Date	Version	Status	Author(s)	Description
31/07/2023	01	Draft	ENG	Table of Content definition
30/09/2023	02	Draft	ALL	Table of content finalization
09/10/2023	03	Draft	ENG	Added further descriptions in the sections; Added draft content in Sections 4 and 5; Added Section 6.2
23/10/2023	04	Draft	ENG	Started filling Abbreviations and Glossary of terms sections. Main responsibilities assignment
25/10/2023	05	Draft	AIMEN	Contribution to section 3.1
14/11/2023	06	Draft	ENG	Contribution to sections 2 and 5
11/12/2023	07	Draft	TEK, AIMEN, DFKI, ITRI	Contribution to 4.6 sub-sections, section 5 (Integration plan) and 4.7 (Requirements Traceability Matrix)
15/12/2023	08	Draft	ENG	Reviewing the entire document and addressing comments
18/12/2023	09	Draft	ENG	Quality check and content review
22/12/2023	10	Final	ENG	Version submitted

Executive Summary

The current deliverable (i.e., D2.1 - HARTU Architectural specification and integration plan) aims to report the main results of tasks T2.1 and provides relevant information about Reference Architecture or high-level specification of the HARTU software solutions that will be provided by technical partners (TEK, AIMEN, ITRI and DFKI) to address pilot needs and requirements. Tasks T2.1 covers the overall project approach on Reference Architecture model, reporting the results on technical drawings and software components specifications identifying key technical challenges to be addressed in HARTU, in terms of implementation and architecture definition.

Considering this, the main inputs that contributed to the definition of the architecture and the specifications of the software are:

- “D1.1 - Real world scenarios and metrics for validation definition”: an exploratory investigation of ‘requirements’ for the HARTU solution and its results, capturing both user level requirements and high-level requirements (i.e., functional, and non-functional) important to understand and extract the basic functionalities of the system.
- “D1.2 Initial setup of real-world scenarios”: helps to extract ‘technical requirements’ considering the different prototypes corresponding to the 8 use cases defined by the project.

The approach followed to complete deliverable D2.1 included:

- The summary (ToC) and scope of the document agreed between the partners, and a detailed template including a description of each section and what was required to detail it, were disseminated to guide section leaders in collecting the required information.
- A general overview of the manufacturing context and the reference architectures in Industry 4.0 is presented.
- Specifications, definition of the reference model (Reference Architecture) in terms of technical drawings (HARTU Blueprint), definition of the architectural layers and FBB specification: starting from the requirements (D1.1) and the use case descriptions as well as prototypes (D1.2), the HARTU approach is represented using UML diagrams.
- Definition of the integration plan that clearly shows “when” all functional blocks within each layer of the HARTU architecture will be delivered for testing and validation within industrial environments or test labs.
- Support for the software development plan through a CI/CD enabled tool that allows rapid updating of the source codes of HARTU components, as well as code quality control and providing a valid repository for technical documentation and problem management.
- Follow-up activities to monitor the progress of the work through different types of meetings: (1) General project follow-up web meetings (every three weeks) involving all partner representatives; (2) WP2 conference specific meetings; (3) Overall project F2F meeting.

1 Table of contents

1	Introduction	9
1.1	Scope of this deliverable	11
1.2	Relationship with other tasks.....	11
1.3	Structure of the document.....	12
2	Methodology.....	13
2.1	Layered Architecture	13
2.2	Functional Building Blocks Specification	14
3	The Context.....	15
3.1	Industry 4.0 reference architectures.....	15
3.2	Background on ROS2 (Robot Operating System).....	17
3.2.1	ROS2 Nodes.....	18
3.2.2	ROS2 Messages	18
3.2.3	ROS2 Services.....	19
3.2.4	ROS2 Actions	19
3.2.5	ROS2 Command Line Interface	20
4	HARTU Adaptive Platform for Robotic Orchestration (APRO).....	21
4.1	Frontend Layer	23
4.2	Application Layer	24
4.3	Information Layer.....	25
4.4	Middleware Layer.....	25
4.5	Physical Layer	25
4.6	FBB Specification	26
4.6.1	Frontend Layer FBBs Specification.....	27
4.6.2	Application Layer FBBs specification	29
4.6.3	Information Layer FBBs Specification	34
4.6.4	Middleware Layer FBBs Specification	37
4.7	Requirements Traceability Matrix.....	41
5	HARTU Integration Plan	43
5.1	General Integration Status	43
5.2	Component Detailed Plan	43
5.3	Software Configuration Management	45

5.3.1	Source code repository	45
5.3.2	CI/CD setup and automation	47
5.3.3	Quality control	50
5.3.4	Guidelines to ease collaboration	54
6	Conclusion.....	55
7	References.....	57

List of figures

Figure 1: Relationships with other WPs and Tasks	12
Figure 2. Component representing a Robot-asset and its Asset	17
Figure 3: ROS 2 Architecture Overview.....	18
Figure 4: Service communication for nodes in the ROS graph	19
Figure 5: ROS2 Action.....	20
Figure 6: HARTU Reference Architecture (Latest version).....	22
Figure 7: HARTU FBB – Frontend Layer – AppManager (builder) Component Diagram	27
Figure 8: HARTU FBB – Frontend Layer – AppManager (Control) Component Diagram	28
Figure 9: HARTU FBB – Frontend Layer – SimEnv Component Diagram	29
Figure 10: HARTU FBB - Application Layer – ImageAcquisition Component Diagram	30
Figure 11: HARTU FBB - Application Layer – ImageSegmentation Component Diagram.....	30
Figure 12: HARTU FBB - Application Layer – GraspPlanner Component Diagram	31
Figure 13: HARTU FBB - Application Layer – PoseEstimation Component Diagram	32
Figure 14; HARTU FBB - Application Layer – ReleasePlanner Component Diagram	33
Figure 15: HARTU FBB - Application Layer – Adaptive MPC Component Diagram	33
Figure 16: HARTU FBB - Application Layer – Imitation Learning Component Diagram.....	34
Figure 17: HARTU FBB – Information Layer – Components Diagram	35
Figure 18: HARTU FBB - Information Layer – HARTU_SegmentModel.....	35
Figure 19: HARTU FBB - Information Layer – LocalGraspModel.....	35
Figure 20: HARTU FBB - Information Layer – GlobalGraspModel.....	36
Figure 21: HARTU FBB - Information Layer – HARTU_PoseEstimationModel	36
Figure 22: HARTU FBB - Information Layer – DMPModel.....	36
Figure 23: HARTU FBB - Information Layer – ART Model	37
Figure 24: HARTU_GMMMModel	37
Figure 25: HARTU FBB - Middleware Layer – ART Classifier Component Diagram	37
Figure 26: HARTU FBB - Middleware Layer – Dynamic Movement Primitives Component Diagram.....	38
Figure 27: HARTU FBB - Middleware Layer – Model Predictive Control Component Diagram	39
Figure 28: HARTU FBB - Middleware Layer – BehaviorTree Component Diagram	40
Figure 29: HARTU FBB - Middleware Layer – DeepReinforcementLearning Component Diagram ..	41
Figure 30: GitLab "Issues" feature example.....	46

Figure 31: Docker based build process automated with GitLab in HARTU	48
Figure 32: GitLab Runner setup for HARTU repositories	49
Figure 33: GitLab pipeline basic representation	49
Figure 34: .gitlab-ci.yml -> pipeline definition to automate Docker build process within GitLab.....	50
Figure 35: excerpt of Google Style Guides	51
Figure 36: excerpt of Common Weakness Enumeration (CWE)	51
Figure 37: extract from gitlab-ci.yml showing code-review setup in to a pipeline	53
Figure 38: extract from codeclimate.yml.....	54
Figure 39: HARTU repositories naming convention.....	55

List of tables

Table 1. ROS2 Commands	20
Table 2. Main Functional Building Blocks (FBB) for HARTU Reference Architecture	26
Table 3. Requirements – Functional Building Blocks (FBB) traceability matrix	41
Table 4 HARTU FBB General Integration Status.....	43
Table 5. HARTU FBB Component, Detailed Integration Plan	44
Table 6. APRO SW component - programming language map	52

Acronyms

List of the acronyms	
CAD	Computer Aided Design
CI/CD	Continuous Integration/Continuous Delivery
DDS	Data Distribution Service
DoA	Description of Action
FBB	Functional Building Blocks
GUI	Graphical User Interface
HARTU	Handling with AI-enhanced Robotic Technologies for flexible manufactUring
H2M	Human to Machine
ONNX	Open Neural Network Exchange
LMM	Learning Movement Model
GMM	Gaussian Mixture Model
WP	Work Package
PM	Project Month
SAM	Segment Anything Model
RA	Reference Architecture
ROS	Robot Operating System
SCM	Software Control Management
XML	Extensible Markup Language
YAML	Yet Another Markup Language
ToC	Table of Contents
UI	User Interface

1 Introduction

Grasp, assembly, and release planning in manufacturing environments involves optimizing the sequence in which a robot or automated system picks up and place objects or assembles them. It aims to enhance efficiency by minimizing configuration time, improving process control and maximizing utilisation of resources, ultimately improving the overall production process.

In grasp and release planning, considerations include the geometry, material and weight of objects, their pose, as well as the robot's capabilities. The goal is to develop a strategy that allows the robot to efficiently grasp items, move them to desired locations, assemble them and release them appropriately. This planning often integrates with larger production planning systems to streamline operations and enhance productivity in manufacturing settings.

Some common characteristics to build a reliable and efficient system to support these features, also from a hardware optimisation point of view, involve the use of some components such as the ones presented below:

- **Object Recognition:** Implement a robust system for recognizing and identifying objects in the manufacturing environment. This can involve computer vision techniques or other sensing techniques, e.g., barcode readers.
- **Grasping Algorithms:** Develop artificial intelligence algorithms that determine the optimal way for a robot to grasp an object based on its shape, size, and weight. This may involve pre-defined grasping strategies or machine learning approaches.
- **Path Planning:** Integrate path planning algorithms to determine the most efficient route for the robot to move between different locations while carrying out manipulation tasks.
- **Robot Control Interface:** Create an interface that allows seamless communication between the grasp and release planning system and the robot's control system. This ensures the planned actions are executed accurately.
- **Sensors and Feedback:** Implement sensors that provide real-time feedback on the success of grasping and releasing actions. This information can be used to adjust the plan if unexpected events arise.
- **Integration with manufacturing Workflow:** Ensure that the manipulation planning system integrate smoothly with the broader manufacturing workflow, coordinating with other systems and processes.
- **Adaptability:** Design the system to be adaptable to changes in the manufacturing environment, such as variations in object types, production volumes, or spatial configurations.
- **Test and Validation:** Rigorously test the system under various conditions to validate its reliability, accuracy, and efficiency in different manufacturing scenarios.

HARTU solution proposes to combine these elements, to create a comprehensive system that optimizes manipulation tasks in a manufacturing context, providing an architecture model (objective of this deliverable and of WP2) that will provide a structured framework for system design and integration.

HARTU implements a Reference Architecture Model which bases its foundations on the following principles:

- **Modularity:** An architectural model allows to break down the system into modular components. This modularity enhances maintainability, scalability, and facilitates easier updates or replacements of specific functionalities, such as object recognition, grasping algorithms or path planning.
- **Interoperability:** Define clear interfaces between different modules to ensure seamless communication enables easy integration with existing manufacturing systems and promotes interoperability with various hardware and software components.
- **Scalability:** with a well-defined architecture, it becomes easier to scale the system to handle increased production demands or adapt to changes in the manufacturing environment. New modules or functionalities can be added without disrupting the entire system.
- **Flexibility and Adaptability:** An architecture model provides a foundation for building flexible and adaptable systems. This is crucial in manufacturing, where the types of objects to be handled and production requirements may evolve over time. The system can be adjusted or extended without extensive redesign.
- **Testing and Validation:** Architectural models support systematic testing at different levels, from individual modules to the overall system. This helps to ensure that each component functions correctly and that the integrated system meets performance and reliability requirements.
- **Data flow optimization:** The architecture model aids in optimizing the flow of data between different components. Efficient data exchange is crucial for real-time decision-making in grasp, assembly, and release planning, especially when coordinating with other manufacturing processes.
- **Documentation:** An architecture model serves as a comprehensive documentation tool, providing a clear understanding of the system's structure and functionality. This documentation is invaluable for maintenance, troubleshooting, and future enhancements.

In summary, an architecture model provides a strategic framework for designing, implementing, and maintaining a robust grasp, assembly, and release planning system in manufacturing, offering benefits respecting the above principles. These principles are satisfied through the approach and methodology used for the definition and specification of the HARTU reference architecture, from its implementation by observing this methodology, while WP2 will gather structured and suitable tool for continuous deployment and testing of the software produced, that will improve system solution in the immediate response to changes due to changed conditions in the acceptance of the system's functionality by end users. The HARTU framework will be able to respond quickly to these changes, as it is supported by an architectural model that foresees and plans the management of change requests, adapts continuously, and provides valid support for the active understanding of the system.

Furthermore, HARTU Reference Architecture efficiently integrates artificial intelligence components, and Robotic Operating System (i.e., ROS and ROS2) that significantly enhance grasp, assembly and release planning systems in manufacturing (in addition to the characteristics to build a reliable and efficient system), (1) enhancing adaptive decisions making based on the real-time

feedback from sensors, (2) continuous learning from data generated during operations allowing system improvement over time, enhancing the collaboration between robots and human workers, ensuring safe and efficient interaction, capturing suitable requirements from them (T1.3), (3) integrating artificial intelligence, manufacturers can achieve a higher level of automation, efficiency, and adaptability, ultimately improving the overall productivity and competitiveness of their manufacturing processes, (4) adopting middleware provided by ROS2 can improve communication between different components of a robotic system, to facilitate the object recognition, manipulation planning and control, (5) integrating simulation tools, like UNITY and MuJoCo, allow developers to simulate and test their grasp, assembly and release planning algorithms in a virtual environment, using production data, enhancing testing, refining, and software validation before deploying it to a physical robot.

1.1 Scope of this deliverable

The current deliverable (i.e., D2.1: HARTU Architectural specification and integration plan) aims to report on the results of tasks T2.1 about the definition of HARTU Reference Architecture (RA). Task T2.1 also provides an integration plan for software identifying for each block of the architecture alpha-beta-final release; it's to show the progress of the software solution during the implementation phase, in order to provide a clear picture about delivery date of each software solution to the project end-users.

The results of D2.1 will serve as input for technical results providers in WP2, WP3 and WP4 (i.e. AIMEN, TEK, DFKI, ITRI) with the supervision of ENG, during the implementation phase of HARTU solution.

1.2 Relationship with other tasks

The requirements collection activities in WP1 considered inputs coming from (1) the DoA; (2) user, functional and non-functional requirements collected during the interview with end-user in T1.1 (D1.1), (3) initial setup scenarios definitions in T1.2 (D1.2), (4) technical requirements coming from software components analysis in T2.1, T2.3 and T2.4. By processing the main inputs, this task will provide WP1, WP2, WP3 and WP4 with the design of the architecture, the specification of the functional blocks that compose it, the integration plan and the tool for integration and continuous delivery, as well as providing a technical integration plan for the software components and main system interfaces as output for the definition of timelines and software releases needed for organizing the integration activities of the results in the experimental scenarios and evaluation in the Tasks T1.5 and T1.6. These specifications will be formalized in this deliverable, which will be used by the technical WPs as a basis for development, as shown in Figure 1:

Task 2.1 – Relationships with other WPs and Tasks

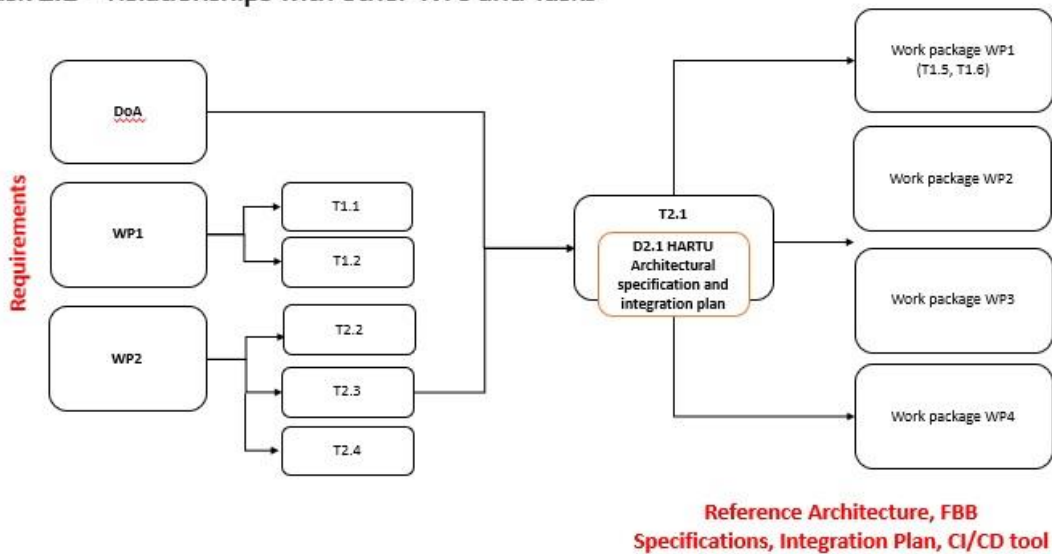


Figure 1: Relationships with other WPs and Tasks

1.3 Structure of the document

D2.1 is divided in six main parts involving:

- **Introduction:** This section identifies the tasks of the project related to the deliverable including information on objectives as well as a short description of the relationship of the current deliverable with the results of other tasks and work-packages.
- **Methodology:** This section describes the approach followed in tasks T2.1 to complete the deliverable D2.1.
- **Context:** A State of The Art Analysis of other initiatives and existing solutions relevant to the HARTU architecture design, complemented by a description on how HARTU can be aligned or supported by them.
- **HARTU RA Specifications:** This is the core part of the document including relevant information from each module composing the HARTU overall solution. This description includes the technical drawing (holistic view) and the specifications of the blocks (modules) representing an external view of the system derived from the requirements defined in WP1; the identification of the HARTU Architecture Model providing the solution to be adopted in HARTU in terms of Functional Building Blocks; the description of main business interfaces to be put in place in the final solution to realize a coherent system from the individual modules; the detailed specification of FBB in terms of system interfaces and the overall Functional and Modular architecture of the HARTU project.
- **HARTU Integration Plan:** that is, the description of the integration plan of the software components that form the system, the date on which these will be available in the different versions, and finally the CI/CD solution that will be implemented for Source Code Management and for deployment and continuous testing of the software as it is developed by the partners involved in the process.
- **Conclusion:** This section provides summarised information on the HARTU Reference Architecture to pave the way to the technical developments in WP2, WP3 and WP4.

2 Methodology

This section presents the architecture approaches used in the definition and specification of HARTU Reference Architecture (RA). These methodological approaches, allow to address separately the concerns of the various stakeholders of the HARTU project, mainly technical partners, and business partners (Pilots), and to handle separately the functional and non-functional requirements. The HARTU RA is designed using an abstraction of the business logic of the services that make up the solution, using a system decomposition based on functional blocks.

The HARTU RA deals with the design and implementation of the high-level structure of new manufacturing systems developing the best technical solution and methodologies to increase flexibility, reconfigurability, and production line efficiency with innovative robotic components.

This document relies on the assembly of a certain number of architectural components in some well-chosen forms to satisfy the major functionality and non-functional requirements of the system.

Following this approach this document provides an abstraction, decomposition, and composition of several architecture layers that provide a holistic view of the system, how the solution is built, the main actors and technologies involved.

Also, this deliverable considers WP1 requirements and the use cases described in D1.1 to develop the HARTU architecture. In particular, the architecture of the project satisfies several of the functional and non-functional requirements of the project. Some of the key requirements driving the development of the architecture are those relating to standards compliance (e.g., Robot Operating System version 2¹), implementation of new manipulation planning and control, and the use of simulation environments. Note however that the HARTU architecture has considered high-level requirements of D1.1, rather than the low-level technical ones that will be further analysed in the scope of WP2, WP3 and WP4. This is because the presented HARTU architecture focuses on high-level decisions with system wide impact on HARTU based systems, rather than on low-level technical details that will be elaborated as part of implementation.

In addition, the objectives of the HARTU RA can be decomposed as in the following:

- To develop a reference architecture for the implementation of HARTU software solution
- To define the logical structure of the infrastructure components in the HARTU stack.
- To define the functional components implementing each infrastructure component to support the evolving adaptive assembly system concept.
- To address functional and non-functional requirements, clearly presenting to the HARTU end user and through a technical drawing the functional block that meets the requirements.
- In order to achieve the above objectives, it should be noted that the different levels of description of an architecture could include both the design and the detailed description of the implementation levels.

2.1 Layered Architecture

The layered architecture approach relies on organizing a system into different hierarchical layers, each responsible for specific functions. This helps to enhance modularity, maintainability, and

¹ <https://docs.ros.org/en/rolling/index.html>

scalability by isolating different aspect of the system itself. Each layer in the architecture shows how the system answers to the users that interact with it in the frontend layer, business logic or component that satisfy the frontend request, data storage or where the data exchanged in the system is stored and maintained, and each handling specific responsibilities within the overall architecture.

The topmost layer that interacts directly with the end users or external systems is responsible for user interfaces, input validation, and frontend logic. Its primary goal is to present information to users in a comprehensible format. The frontend layer, interact with the Application (or Business Layer) and focuses on core system functionalities. It processes and manages data, implementing business rules and orchestrates communication between different parts of the system. This layer is independent of the user interface, making it easier to modify or update business rules without affecting the frontend layer. The data, however, is managed in the information layer, where it is possible to find services and/or supports to store data, or simply placeholders to the data model that describes data and metadata models, i.e., the container.

The main advantages of using this approach, are:

- **Modularity:** each layer is modular, making it easier to understand, maintain and update specific components without affecting the entire system.
- **Scalability:** layers can be scaled independently, scaling for instance the business logic without touching the frontend or information layers.
- **Interchangeability:** layers can be replaced or updated without affecting the other layers, as long as the interfaces between layers remain consistent.

The layered approach enhances the separation of interests, making it easier to manage and evolve complex systems over time, and for this reason it remains a valid support and definition and specification mechanism for complex software systems such as those in the robotic manufacturing context.

2.2 Functional Building Blocks Specification

To address the intricacies of complex problems, the Functional Building Blocks (FBB)² approach emerges as a structured and systematic methodology, facilitating the decomposition of problems into manageable components. Based on the principles of modularity and abstraction, this approach provides a robust framework for the design and construction of systems that possess attributes of flexibility, scalability, and maintainability.

The core principle of the FBB approach revolves around the division of a problem or system into discrete modules, commonly referred to as building blocks. These building blocks are meticulously crafted to encapsulate specific functions or operations that contribute to the overall functionality of the system. Through the decomposition of a problem into these coherent and manageable units, the FBB approach facilitates a comprehensive understanding of the system's requirements, while fostering effective collaboration among team members.

² <https://pubs.opengroup.org/architecture/togaf8-doc/arch/chap32.html>

One notable advantage of the FBB approach lies in its ability to facilitate reusability and extensibility. Each building block is carefully designed to be modular and independent, characterized by well-defined interfaces that facilitate seamless interaction and communication with other blocks. This modular nature streamlines the development process and enables the reuse of existing building blocks in diverse systems or contexts, thereby conserving time and effort.

Furthermore, the FBB approach fosters abstraction as a fundamental principle. Abstraction involves concealing the internal complexities of a building block and exposing only the pertinent details to the external environment. By presenting a high-level view of the building block's functionality, this abstraction layer allows other system components to interact with it without needing an understanding of its intricate internal mechanisms. Consequently, modifications made to a specific building block are less likely to have an overarching impact on the entire system, enhancing maintainability and mitigating the risk of unintended consequences.

The FBB approach also has inherent scalability, an attribute derived from its modular design. By conceptualizing a system through the lens of building blocks, the addition or removal of a functionality becomes effortless. As system requirements evolve or expand, new building blocks can be created and seamlessly integrated, or existing blocks can be modified or replaced. This intrinsic flexibility ensures that the system can adapt and expand without major disruptions or extensive overhauls.

3 The Context

HARTU project's developments are guided by the requirements identified with the support of the 5 industrial companies and the expertise of the technology partners. Due to the wide variety of applications to be implemented and the aim to impact the development process of future applications, HARTU proposes an open reference architecture that can be shared and adopted for third parties.

3.1 Industry 4.0 reference architectures

Several government institutions and business organizations have considered the Industry 4.0 (I4.0) paradigm as a key factor in their industrial development strategies. This has given rise to initiatives such as Plattform Industrie 4.0³ in Germany and Industry IoT Consortium⁴ (IIC) in the USA, to name the most cited ones. Their common denominator is a coordinated effort between government, industry and academia to support innovation in manufacturing processes, based on the total interconnection between different manufacturing assets in a technological context of big data capturing and processing. In this sense, all initiatives are working on the standardization of their reference architectures (e.g., RAMI 4.0 in the case of Plattform Industrie 4.0 and IIRA in the case of IIC) and on the analysis of their confluence. However, standardization and correlation of reference architectures is often a slow process.

³ <https://www.plattform-i40.de/IP/Navigation/EN/Home/home.html>

⁴ <https://www.iiconsortium.org/>

RAMI [1] presents a cubic model that provides a framework for a common understanding among the entities of an I4.0 System with respect to three axes:

Layers axis: It describes the six functional levels into which manufacturing systems for Industry 4.0 can be divided. In each of these layers, service definitions abstractly describe the functionality provided to a layer N by a layer N-1.

- Business layer: It orchestrates the high-level services to determine the status of the processes at a factory level.
- Functional layer: It provides a definition of the high-level services offered by an asset and manages their access remotely.
- Information layer: It acquires, processes and adapts the data from assets while ensuring its integrity and persistence.
- Communication layer: It establishes architectural styles, message patterns and data formats to ensure interoperability.
- Integration layer: It offers low-level services that enable access to the data and functionalities of the asset.
- Asset layer: It represents the physical or logical entities with value for a company, including human beings.

Life Cycle Value Stream axis: Supported by IEC 62890, it describes the operational status of the product, differentiating between product type and product instance:

- Product type: A product in development stage constitutes a product type.
- Product instance: A manufactured product constitutes an instance of a product type.

Hierarchy Levels axis: It adopts some of the factory hierarchy levels of ISA 95 and ISA 88 (such as Enterprise, Work centres, Stations and Control device), while adding additional levels to make the factory hierarchy consistent with Industry 4.0:

- Connected world: It represents a group of companies collaborating above the Enterprise level.
- Field device: It represents devices that are directly involved in the manufacturing process below the Control device level.
- Product: It represents the product in the factory hierarchy.

Together with this cubic model, RAMI 4.0 also introduces the I4.0 Component as a participant of a manufacturing system [2]. An I4.0 Component consists of an asset (physical part) and an Asset Administration Shell or AAS (virtual part). I4.0 Components are service-oriented: the AAS provides the asset an interface through which service requests from other I4.0 Components are channelled. Within the AAS, service requests are handled by a Component Manager that manages the AAS submodels (also called Manifest), made up by the set of properties that describe the data and functionalities of the asset.

Services provided by a I4.0 component can be grouped into two categories as seen Figure 2:

- Submodel Services: They provide access to the information submodels of an I4.0 component without interacting with the asset.
- Asset Related Services: They involve an interaction with the asset to execute some functionality or operation, or to manage its state.

Services provided by I4.0 Components can be combined to compose manufacturing applications that are offered to other components, resulting in Application Relevant Services [2].

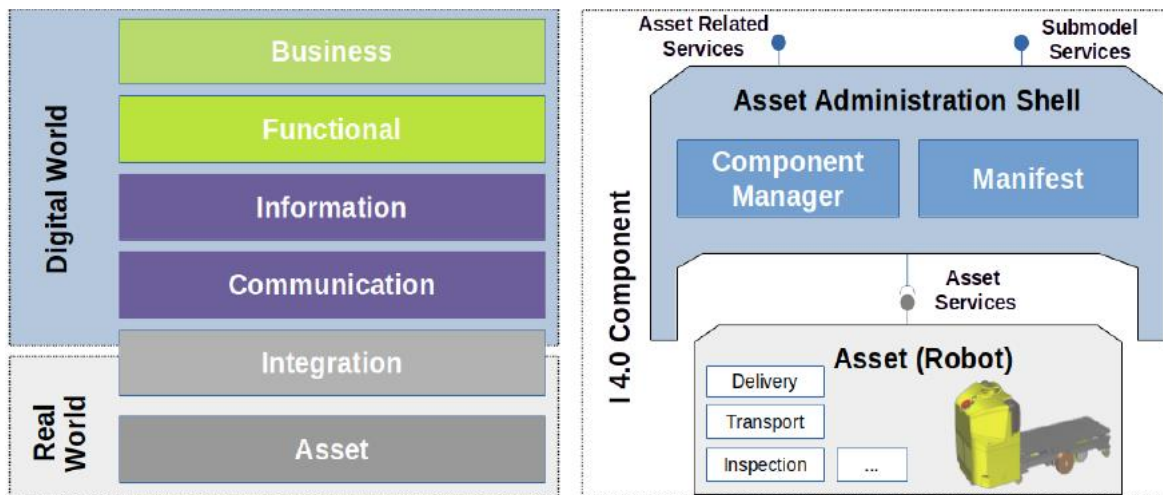


Figure 2. Component representing a Robot-asset and its Asset

3.2 Background on ROS2 (Robot Operating System)

ROS2 is a middleware, a software system dedicated to handling connections and data exchange between users and applications. It is based on an anonymous publish/subscribe mechanism that allows for message passing between different processes. ROS2 includes a set of software libraries and tools to build robot applications. It contains a great number of drivers, state-of-the-art algorithms, and developer tools.

Among these key tools, the ROS core includes tools that can be helpful to visualize data, navigate package structures, and create scripts that automate complex configurations. Examples include “rviz”, a 3D visualizer used to present robots, sensor data, and their environment. Furthermore, additional packages are available for SLAM (Simultaneous Localization and Mapping), navigation, perception, and simulation, to name a few.

ROS2 handles a network of nodes in a system and the connections that allow them to communicate, this is known as the ROS2 graph. A general overview of the architecture behind ROS2 is shown in the following image:

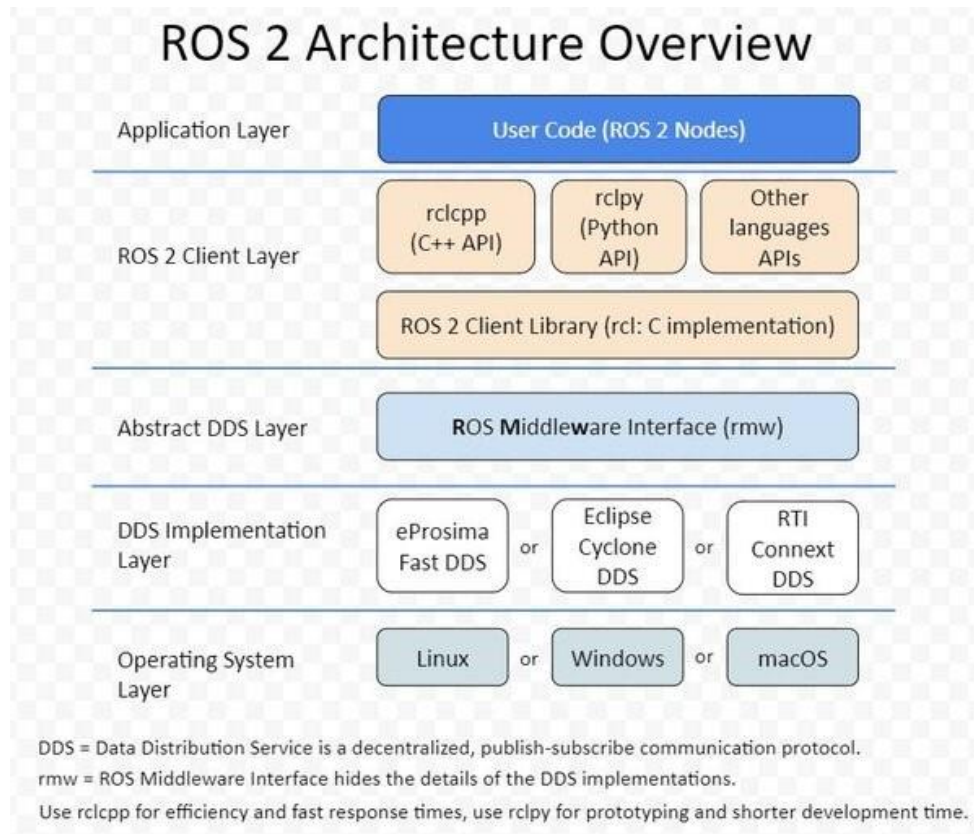


Figure 3: ROS 2 Architecture Overview

Overall, the main improvements from ROS to ROS2 include an improved communication stack with real-time data distribution service (DDS) protocol, logging improvements, the ability to configure Quality of Service in an easier fashion, provide improvements to the command-line interface, improved support for continuous integration and deployment, among others. ROS2 provides a more robust, flexible, and powerful solution for robotic applications.

A brief introduction of nodes, messages, services, actions, command line interface, and launch tools in ROS2 follow in this section of the document.

3.2.1 ROS2 Nodes

ROS2 nodes are the basic computation unit of the network. Each node represents a single process running in the system. They use the ROS2 Client Library to communicate with other nodes. ROS2 applications typically communicate through interfaces of one of three types: messages, services, or actions. Nodes can send and receive messages via buses known as topics, which can be user-defined and include anything from sensor data to actuator commands. On the other hand, services and actions provide a single result for each call.

These nodes also possess a shared database that allows for communal access to static and dynamic information. This is known as a parameter server.

3.2.2 ROS2 Messages

Nodes can publish messages to named topics to transmit data to other nodes or subscribe to topics to receive messages from other nodes. For example, a node can be focused on reading and handling

information from a camera and publish an image message on a specific topic so the image can be accessed by the whole system. Messages are written in the ROS Message Description Language (.msg) and are used to describe how the data being sent is structured. Messages can contain many different types of data, from simple types like integers to more complicated ones such as images and custom messages.

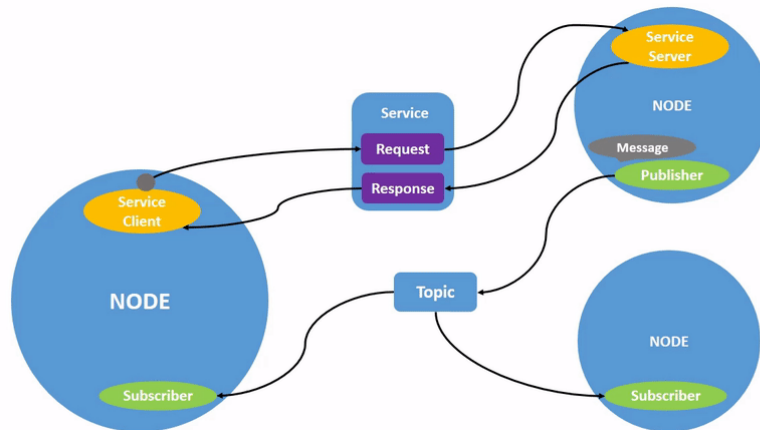


Figure 4: Service communication for nodes in the ROS graph

3.2.3 ROS2 Services

Nodes can also **act** as services. Services handle a specific computation or data on request from another node. These are optimized for handling quick computations since the node requesting is usually on hold until it receives a response from the service request. A node that wants to request a computation or data (*client*) sends a request message to another node (*server*), which fulfills the request and sends a response to the client.

A service is defined by a service server and a service client. The service server's external behavior is defined by ".srv" files in the "srv/" directory, where the input and output of the service is defined. Service clients are the nodes that send the input information and request an output.

3.2.4 ROS2 Actions

If there is a need for handling longer computation requests, ROS2 provides a solution: actions. Actions can send feedback on the status of the long-running computation and can be interrupted. Actions are defined by ".action" files in the "action/" directory.

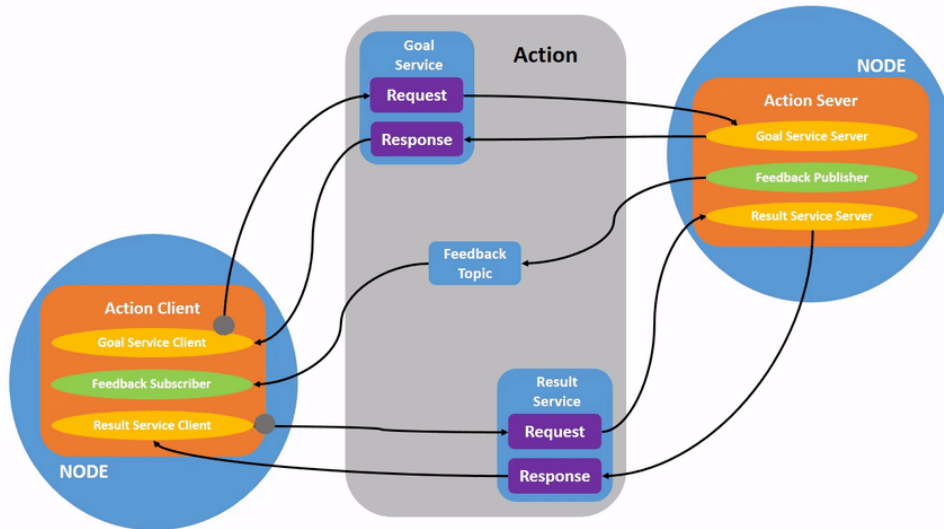


Figure 5: ROS2 Action

3.2.5 ROS2 Command Line Interface

ROS2 also provides a set of commands for introspecting and working with nodes, topics, services, and more. These commands are accessible through the main command “ros2”. A list of some commands is shown in the following table:

Table 1. ROS2 Commands

Command	Description
action	Introspect/interact with ROS2 actions.
bag	Record/play a rosbag.
launch	Run/introspect a launch file.
node	Introspect ROS2 nodes.
param	Introspect/configure parameters on a node.
run	Run ROS2 nodes.
service	Introspect/call ROS2 services.
topic	Introspect/publish ROS2 topics.

The use of these commands should be done in the following way:

```
ros2 <command> <verb> <flags>
```

Where “<command>” can be one of those listed in the previous table, “<verb>” further specifies the action of the command (*list, echo, pub*), and “<flags>” help define the behavior of the command (e.g., list of topics or loop flag for rosbags). For example, the *topic* command can be used to display all the existing topics:

```
ros2 topic list
```

For more information on the command line interface, the “--help” flag can be used in different scenarios:

```
ros2 --help
```

```
ros2 <command> --help
```

```
ros2 <command> <verb> --help
```

ROS2 Launch

Since a ROS2 system typically consists of many nodes running across different processes, the launch system provides a way to automate the running of many nodes with a single command. This system utilizes a “.launch” file, which can be written in Python, XML, or YAML, to describe the configuration of the system. This configuration may include instructions to use the ROS2 commands, run programs, and define arguments. These instructions can then be run using the “ros2 launch” command.

4 HARTU Adaptive Platform for Robotic Orchestration (APRO)

HARTU Reference Architecture (RA) is designed as multi-layered architecture and built on top of functional building blocks methodology. Inspired by the requirements collected in WP1, the technical design of the architecture was also started in WP2 with the collaboration of the main technological partners involved (i.e., ENG, TEK, AIMEN, DFKI, ITRI). Figure 6 shows the final version of the HARTU RA, obtained after several iterations with the partners involved, through brainstorming meetings and resolution sessions which allowed the technical group to release substantial parts of the architecture:

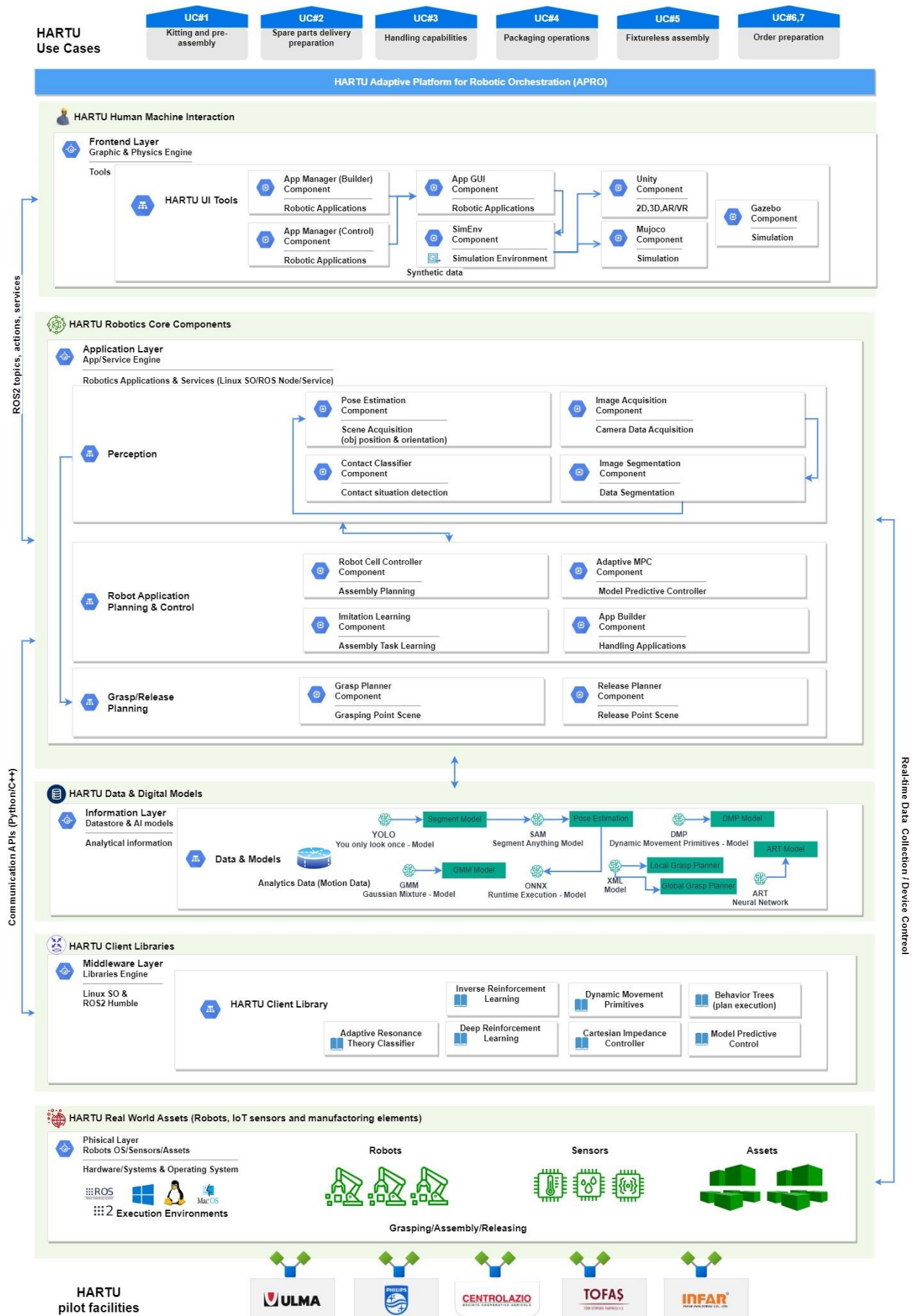


Figure 6: HARTU Reference Architecture (Latest version)

Each layer in the architecture communicates with the adjacent through well-defined interfaces or APIs, enabling loose coupling and facilitating independent development and testing of each layer. This separation of concerns allows for easier maintenance, scalability, and the ability to replace or modify individual layers without affecting the entire system. Interoperability between the layers will be guaranteed by the interfaces offered by the ROS2 in HARTU, its architecture being based on this technology; however, any other interaction mechanisms (standard protocols, third-party libraries) are not excluded should there be a need to solve specific integration problems using custom solutions. Furthermore, to address complex problems in software solution design, the Functional Building Blocks (FBB) approach was applied to better specify the technical structure of HARTU RA layers (see section 2.2).

Valid graphics and physic engine tools (e.g., Gazebo⁵, Mujoco⁶, Unity⁷, etc.) will be provided to test machine learning or deep learning models to build software applications without compromising real data (e.g., production data). Functional building blocks will implement added value services able to cover all HARTU use cases and meet the needs of the end user.

The HARTU RA is composed of the following layers:

- Frontend Layer
- Application Layer
- Information Layer
- Middleware Layer
- Physical Layer

Some common features between the layers can be summarized as follow:

- Each layer can include one or more functional (e.g., Perception in the Application Layer) or technological block (HARTU UI Block in the Frontend Layer).
- Each main block can include sub-blocks (and so on), which extend the main one (not in the current version of the HARTU RA).
- Each block (and sub-block) can contain components/modules/services/software artifacts and technologies. Components (generic term to express the list of solutions) expose a set of interfaces (APIs) that allows them to communicate with other blocks (in the same layer) or externally with other layers of the architecture. Internally, each component can implement interfaces that perform operations useful for carrying out activities (e.g., tasks, operations, etc.) common to the functional or technological area to which it belongs.

4.1 Frontend Layer

The Frontend Layer is about Human-to-Machine (H2M) interaction with software systems exposing high-level functionality or interfaces (UI) for HARTU operators and system integrators. GUI and simulation tools are the main components of this layer, which provide visual representations of information about the process to be monitored/controlled and business functions that will provide

⁵ <https://gazebosim.org/home>

⁶ <https://mujoco.org/>

⁷ <https://unity.com/>

outputs to the end user. Simulation functionality to design new physical solutions in realistic environments with synthetic or real-time data flows will be also provided. HARTU end users will interact with this layer using dashboards (graphical user interfaces) to perform operations such as to learn new assembly skills or start the execution of an application.

The main interaction based on ROS 2 communication protocols (mostly) will be with the Application layer, which will provide the highest-level business functions of the architecture (detail below).

4.2 Application Layer

The Application Layer contains interfaces (APIs) and provides services for frontend application processes; transmission forward the requests to the level of presentation (Frontend Layer), while receiving them. Communication with the ROS2 layer typically occurs via interfaces of the following three types:

- messages (topics),
- services
- and actions.

Topics are used for data streams (sensor data, robot state), services to execute remote procedures fast, while actions are used for any discrete behaviour that moves a robot or runs for a longer time but provides feedback during execution. Communication between the Frontend and Application layers in HARTU will be provided by ROS2, although other communication protocols can be used if there is a real need.

The core components for robotic interaction will be exposed in this layer; heterogeneity in the technologies that express the functional blocks in this layer will represent one of the main characteristics of this layer that will host ROS 2 compliant applications, but also components developed ad-hoc in any programming language to meet a specific end user need. However, this level supports the following functional application areas of robotics applications:

- Perception,
- Robot Application Planning & Control
- and Grasp/Release Planning.

The Perception block will contain components and technologies capable of providing functions to support operators in perception (through the Frontend layer tools) of static and dynamic objects to build a reliable representation and detailed robot environment using computer vision and machine learning techniques. The perception block in a robot is therefore responsible for object detection, segmentation, and tracking. The component APIs of this block will interact at a lower level mainly with the Grasp/Release Planning block components, and internally with the components of the same block (e.g., the Image Segmentation component will identify and separate parts to be handled by the robotic system within an image provided by the Image Acquisition component).

The Robot Application Planning & Control block will provide components for the execution of tasks, such as controlling the trajectory of the robot (Planning), and position movement (Control). In all robotic applications, completing a generic task requires execution of a specific action prescribed to

the robot. The correct execution of this action is entrusted to the Robot Application Planning & Control Block, that should take care of it with commands consistent with the desired action.

The Grasp/Release Planning block contains components and services for grasp detection and planning. The technologies of this block are particularly important because allow a robot to support humans in daily work.

4.3 Information Layer

The Information Layer encompasses dedicated data stores to persist information related to various aspects of the system. This support will be of different nature, and will manage heterogeneous information, serving the applications and services in the Application layer that will need to consume data (e.g., analytics data, time series etc), or instantiate a specific data model to train AI algorithms, or model information for portability of applications between different execution environments. Furthermore, The HARTU reference models are also included in this layer (e.g., ONNX, SAM, DMP, GMM, etc.) like “placeholders” to make understandable the plethora of modelling solutions that the project will make available for system integrators.

4.4 Middleware Layer

The Middleware Layer is an abstraction software between an operating system and the applications running on it. It essentially functions as a hidden translation layer, allowing communication between the Application layer and Operating System libraries regardless of their implementation. It effectively abstracts the high-level functions of the HARTU architecture from the dependency of the main OS like Linux, Windows, or Mac. For ROS 2 the decision was made to build it on top of an existing middleware solution (i.e., DDS or Data Distribution Service). The main advantage of this approach is that ROS 2 can leverage an existing, well-developed implementation of that standard. There are many different implementations available, and each has advantages and disadvantages in terms of supported platforms, programming languages, performance characteristics, memory space, dependencies, and licenses. To abstract from the specifics of these APIs, an abstract interface has been introduced that can be implemented for different DDS implementations. This middleware interface defines the API between ROS 2 client library and any specific implementation. Each interface implementation will usually be a thin adapter that maps the generic middleware interface to the middleware implementation-specific API.

The HARTU Client Library block contains a series of AI and non-AI libraries, ROS 2 compliant or simply third-party libraries, which will provide functionality and algorithms to the Application layer blocks. AI algorithms use machine learning, deep learning and so on techniques to train training models, therefore, to make accurate predictions.

4.5 Physical Layer

The Physical Layer represents the lowest layer of the architecture where the partners/end-user facilities (laboratories, shop floor, etc.) are located. The layer also defines the physical components of the system, including production equipment, product parts, sensors, and machinery (e.g., robots). Here, the data relating to the assets will be produced in real-time which will be used to feed the decision-making process, the AI algorithms, and the business functions in the Application layer.

Finally, given the high flexibility of the architectural design the software deployment strategy in the HARTU pilots will be based on a modular basis, using docker containerization technology, while YAML file⁸ will be distributed to provision each installation using a configuration file. Furthermore, pilots will be free to choose additional technologies for container orchestration and management (e.g., Kubernetes) if they deem it necessary to automate or simply better manage workloads.

4.6 FBB Specification

This section describes all the functional building blocks (FBBs) that compose the HARTU RA, going into detail about each block:

- Each block and sub-blocks are represented by a unique identifier. These identifiers are used in the traceability matrix to reference a block with a list of requirements as shown in **¡Error! No se encuentra el origen de la referencia.**, in **¡Error! No se encuentra el origen de la referencia.** where general integration status of each block is presented to show when different releases of the block and its components will be released (during the project lifetime).
- Each block is described in detail in the following sections, reporting a component diagram to depict the internal composition (list of components and main interaction among them), the general description of the block (the aim and purpose inside HARTU RA), and main interfaces or features to communicate with other components.
- Each block could be implemented by different partners that collaborate to deliver new added-value services for HARTU and users.

The table below contains the list of the main FBBs defined for the HARTU RA, with an associated unique code (ID), the name and a brief description of the module. Each identifier is formed as follows:

- **MOD:** it is common to all blocks and indicates MODULE, to denote the nature not of a single component but of a set of components and services.
- **FL, AL, IL, ML** is related to the layer to which the block belongs, i.e., where it's placed within the HARTU RA.
- The last letters are an abbreviation of the module name.

Table 2. Main Functional Building Blocks (FBB) for HARTU Reference Architecture

ID	FBB Name	Description
MOD.FL.UIT	HARTU UI Tools	Front-end components
MOD.AL.PER	Perception	Back-end perception components
MOD.AL.APC	Robot Application Planning & Control	Back-end robot control & planning
MOD.AL.GRP	Grasp/Release Planning	Back-end Grasp/Release planning components
MOD.IL.DEM	Data & Models	Data store & data models
MOD.ML.CLL	HARTU Client Library	Back-end Linux OS/ROS2 library components

⁸ <https://yaml.org/>

The HARTU RA primarily supports the functional requirements—what the system should provide in terms of services to its users. The system is decomposed into a set of key abstractions, taken (mostly) from the problem domain as stated in previous Section 2.2. This decomposition is not only for the sake of functional analysis, but also serves to identify common mechanisms and design elements across the various parts of the system to be further details in the sub-sections of section 4.6.

These sub-sections describe the HARTU functional blocks, by providing a description of each component of the HARTU solution in more technical detail.

4.6.1 Frontend Layer FBBs Specification

The Frontend Layer contains one main functional block or HARTU UI Tools (MOD.FL.UIT) and it is split into two main components (with a common interface):

- The Builder: used for the configuration of the application configuration.
- The Control: used at application runtime.

4.6.1.1 AppManager

The AppManager (Builder) offers the GUI and the functions to define an application:

- GUI. Based on QT, offers the GUI to create the application, control it and access to the simulation environment, which is used to create the segment and local grasp models.
- It takes as input various data specific for the application and the part to be handled: part CAD (if available), gripper CAD, robot model, part delivery (box, on a table), final position, etc.
- It generates a file with the configuration data and the models for segmentation and local grasping.

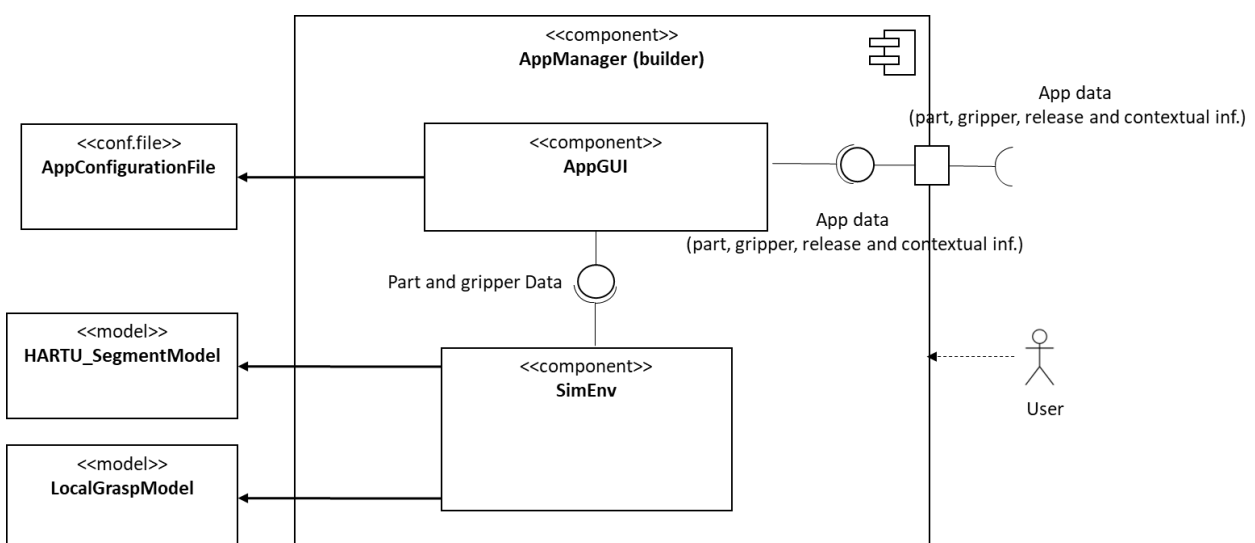


Figure 7: HARTU FBB – Frontend Layer – AppManager (builder) Component Diagram

The AppManager (Control), using the same GUI allows the user to launch the application and monitor and control the process:

- It takes as input the configuration file generated with the Builder and uses the data of other components that it calls at execution time.
- The result is a set of commands to the robot (movements, grasping tool related operations) as well as to other external devices (e.g., machines).

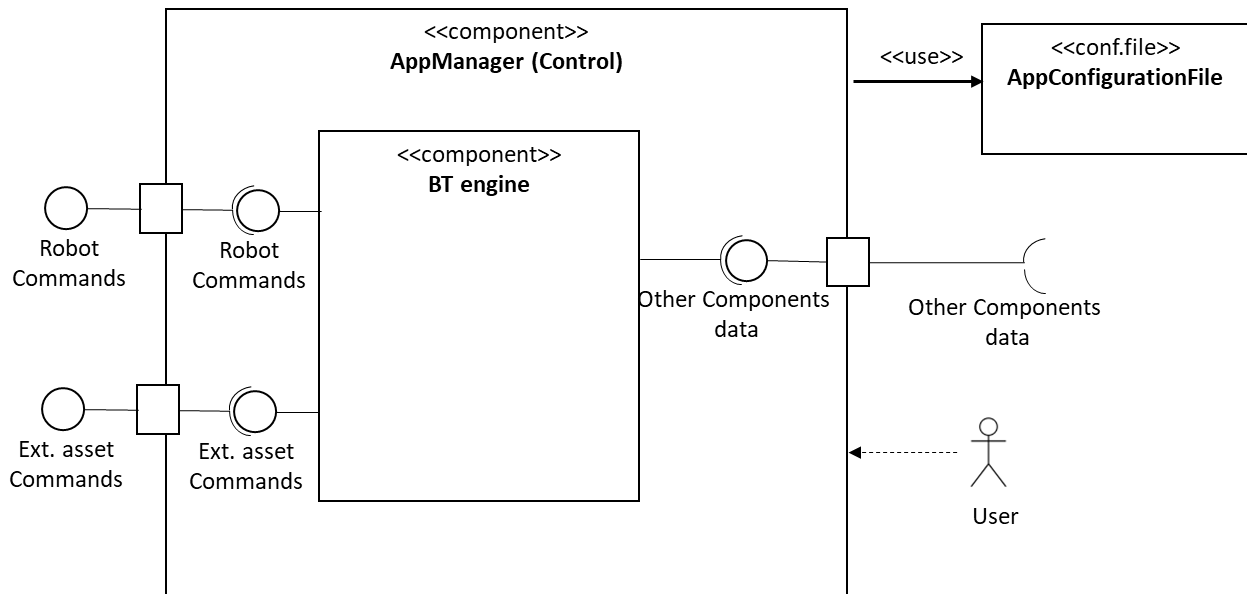


Figure 8: HARTU FBB – Frontend Layer – AppManager (Control) Component Diagram

4.6.1.2 SimEnv

The SimEnv is composed for a set of components used to create three of the models needed in the process. Internally it uses two third party simulation tools, UNITY (commercial) and MuJoCo (Open Source).

- It is accessed through the AppManager (Builder) and uses as input the information of the part, the gripper and the environment (boxes, robot, etc.)
- It supports three of the components in the HARTU RA:
 - SegmentModeller. It requests **SyntheticImageDatasetCreator** to create a database of synthetic images using the CAD of the part and various parameters to randomize the scene in Unity.

The SegmentModeller uses YOLOv5 to generate the Segment Model of the part.

- LocalGraspModeller. It estimates an initial list of possible grasping candidates based on several criteria and uses **LocalGraspPointTester** to validate the quality of these candidates. To do so, it requests MuJoCo to simulate each candidate's grasping operation and calculates the operation's metrics. The result is the final Model (list) of best grasping point candidates.

- GlobalGraspModeller. It trains a neural network to determine which is the best candidate in a cluttered scene (GlobalGraspModel). The GlobalGraspModeller generates a scene with N randomly arranged objects, it segments the image and selects one of the possible grasping points. It then requests the **GlobalGraspPolicyTester** to execute the grasp operation (in MuJoCo), which returns the quality metric of the grasp operation and a vector with some features of the scene.

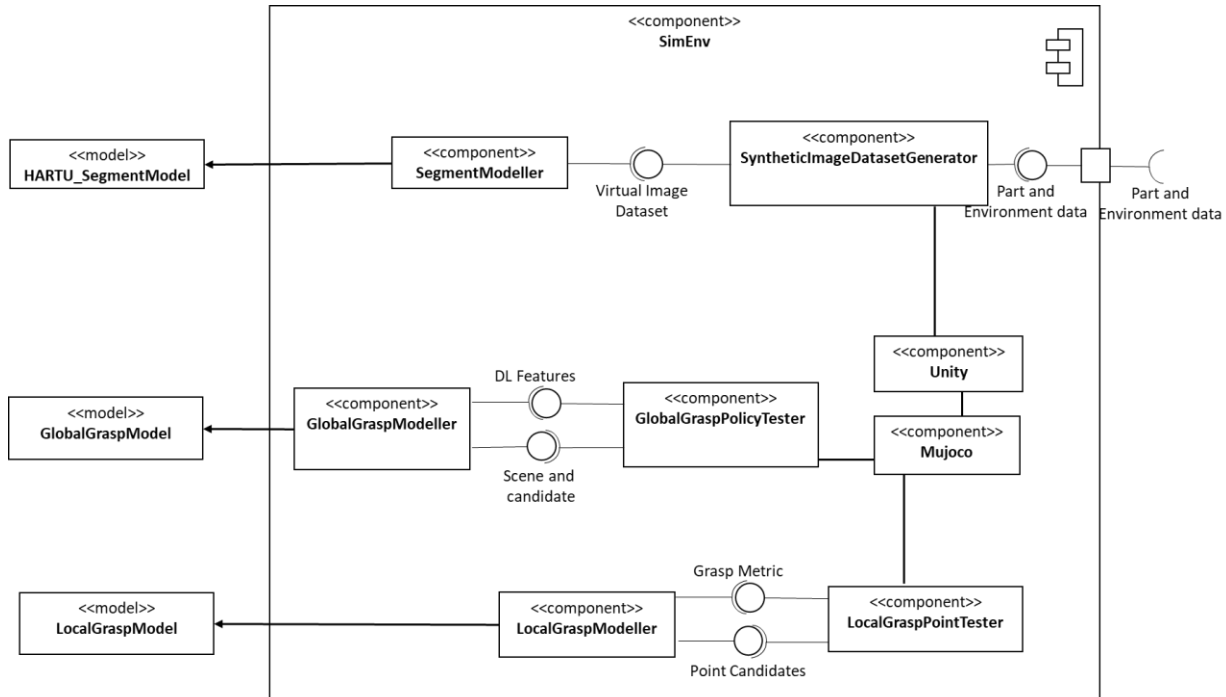


Figure 9: HARTU FBB – Frontend Layer – SimEnv Component Diagram

4.6.2 Application Layer FBBs specification

The Applications Layer contains three functional blocks:

- Perception (MOD.AL.PER)
- Robot Application, Planning & Controls (MOD.AL.APC)
- Grasp Release Planning (MOD.AL.GRP)

Each of these blocks contains several components which are described below, together with the main interfaces (communication internal to the component, i.e., with other internal components) both internal and external (communication between the blocks of the Application layer or with those of the Middleware).

4.6.2.1 ImageAcquisition

The ImageAcquisition component is used to generate the images of a scene in different formats. These images can be used for different components (e.g., the ImageSegmentation and the GraspPlanner).

- It takes as input the information provided by the camera. This is processed by the CameraAPI specific for each device (they have been implemented for those initially identified for the prototypes, i.e. Photoneo⁹ and ZED2i¹⁰).
- The CameraNode component uses the processed data to publish it in three different formats: RGB, PointCloud and Depth. This information is provided on subscription or on demand.

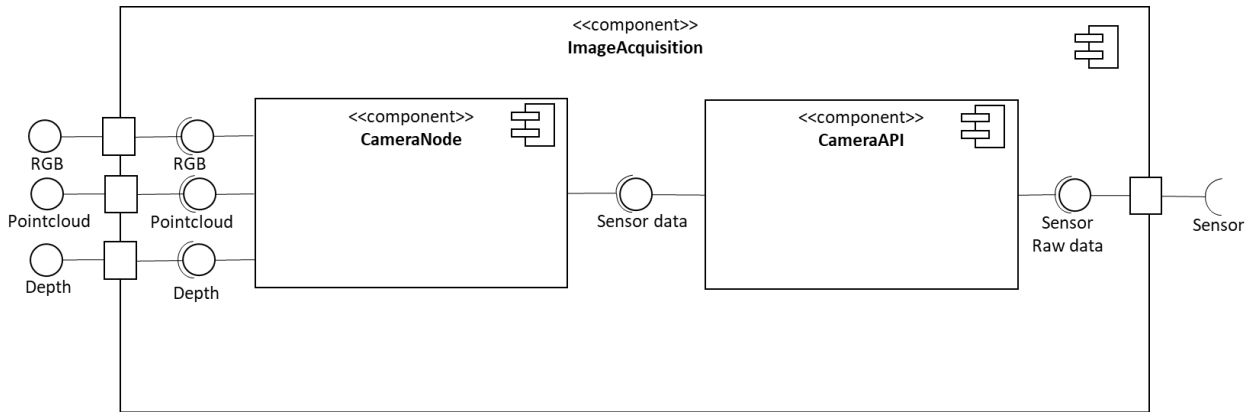


Figure 10: HARTU FBB - Application Layer – ImageAcquisition Component Diagram

4.6.2.2 ImageSegmentation

This component is in charge of creating a pixel level segmented image. It uses two sources of information:

- Information provided the ImageAcquisition component.
- The model generated by the SegmentModeller. It takes it as input for the SAM model that finally generates the segmented image.

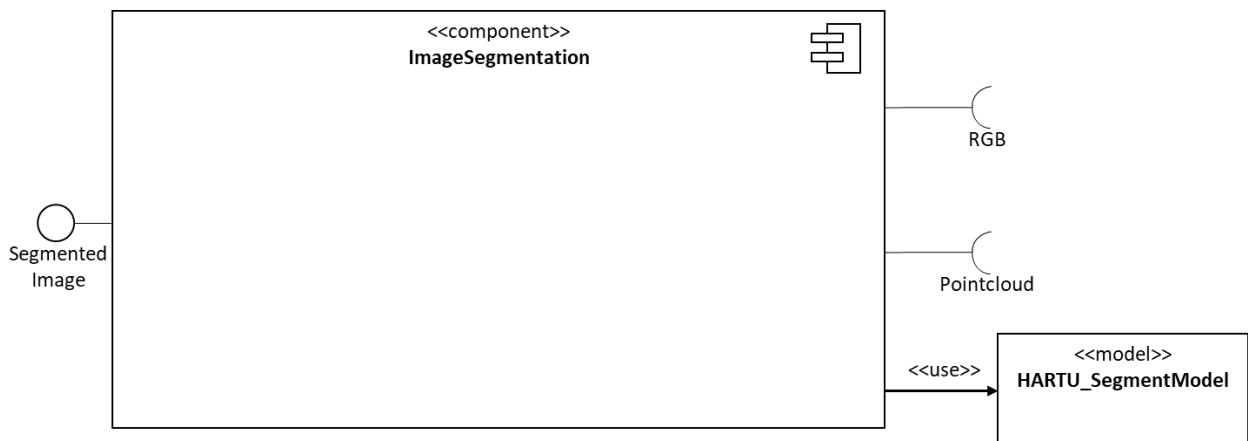


Figure 11: HARTU FBB - Application Layer – ImageSegmentation Component Diagram

⁹ <https://www.photoneo.com/>

¹⁰ <https://www.stereolabs.com/products/zed-2>

4.6.2.3 GraspPlanner

At configuration time, the LocalGraspModeller has generated the list of valid grasp candidates for a given part-gripper pair. The GraspPlanner is in charge of deciding which one to use in a given scene with multiple objects. The component offers two working modes:

- The heuristic mode is thought to be used in scenes where the picking sequence can be predefined (e.g., parts are orderly distributed on a tray)
- The AI-based mode is thought to be used in scenes where the picking order is not predefined, and a dynamic decision module is required to make a decision in each scene.

The component takes as input the RGB and PointCloud data of a scene provided by the ImageAcquisition component, the segmented image provided by the ImageSegmenter, the reference-specific LocalGraspModel, and the GlobalGraspModel obtained using Deep Reinforcement Learning. It also gets the pose of the part that will be finally picked up, provided by the PoseEstimation component.

Finally, the component generates the motion plan to be executed by the robot to pick the selected object from the selected grasping point.

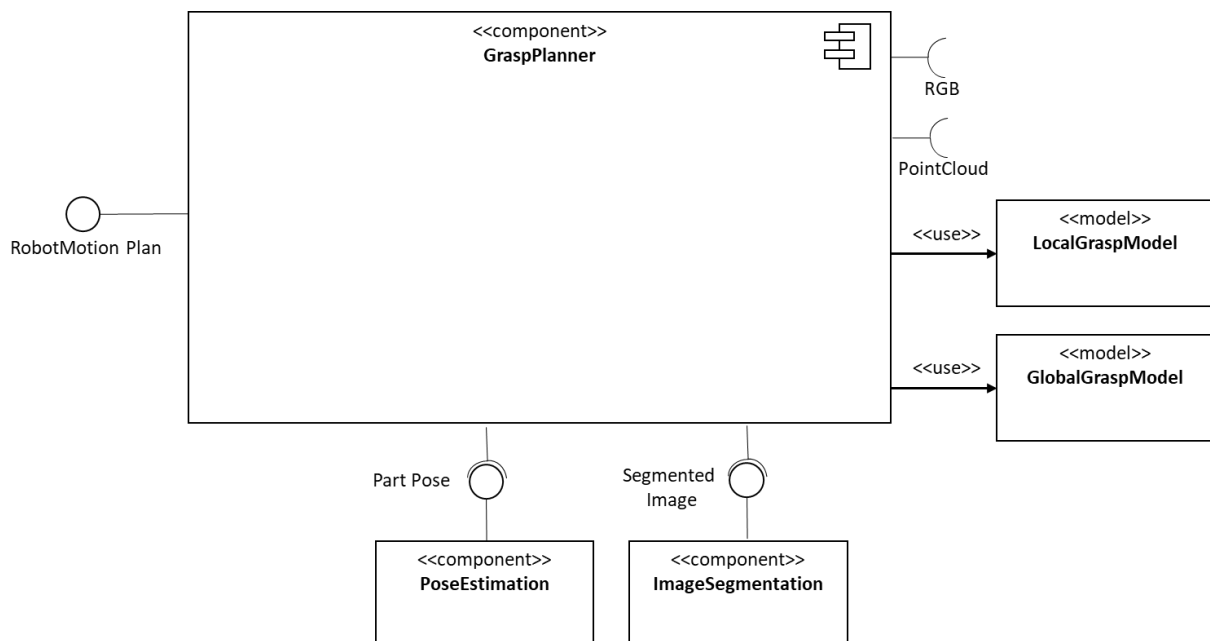


Figure 12: HARTU FBB - Application Layer – GraspPlanner Component Diagram

4.6.2.4 PoseEstimation

The goal of this component is to identify the position and orientation of each segmented object, according to the requirements of each use case. It receives data from the ImageAcquisition and ImageSegmentation modules to perform this task.

- Calls the service for the RGB, Depth, and PointCloud data from the ImageAcquisition module when PoseEstimation is requested.

- Receives the SegmentationMask data from the ImageSegmentation module.
- Estimates and publishes the position and orientation of each object. If the request is performed on an object with a known CAD, this module uses the HARTU_PoseEstimationModel for estimation. If the request is performed on planes or primitive shapes, the pose is calculated with classic algorithms.

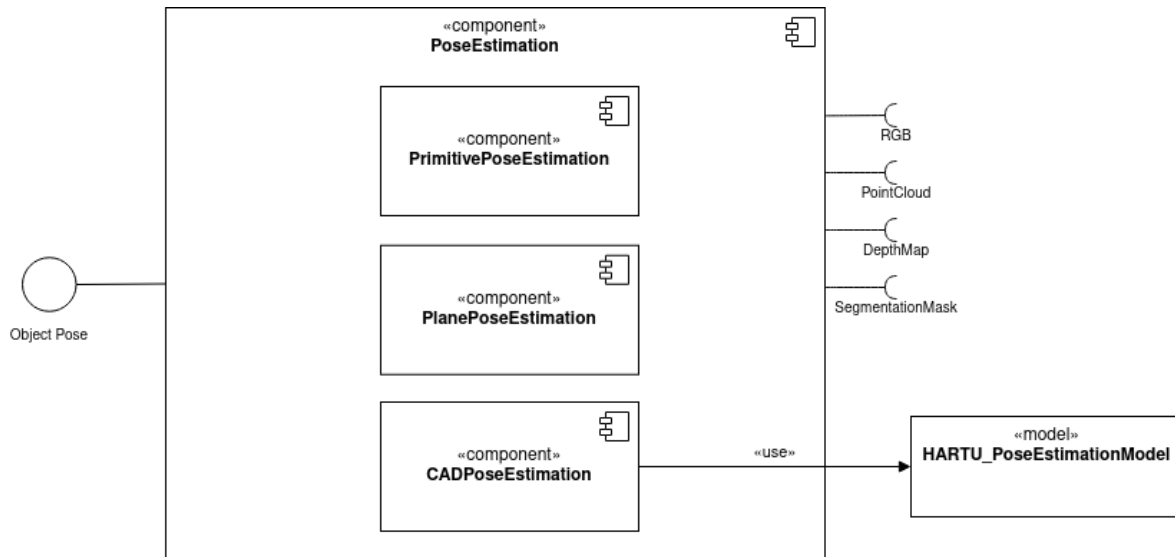


Figure 13: HARTU FBB - Application Layer – PoseEstimation Component Diagram

4.6.2.5 ReleasePlanner

This component is responsible for generating the robot motion to place an object in the destination position. It provides different decision alternatives depending on the application.

- In some cases it needs to monitor the way the object has been picked (to create a mosaic), information provided by the GraspedPartMonitor component.
- It uses the information provided by the ReleaseZoneMonitor component to know the free space available at the target area, e.g., the free space in a container where we need to create a mosaic with the picked parts.
- Sometimes it needs to identify the reference picked by means of the Barcode label, information provided by the BarCodeReader Component.
- It calls the MosaicGenerator component to estimate the best pose of the part in the target container (in the case of mosaic generation). In some particular cases, it is an external system who calculates it (e.g., a warehouse system).

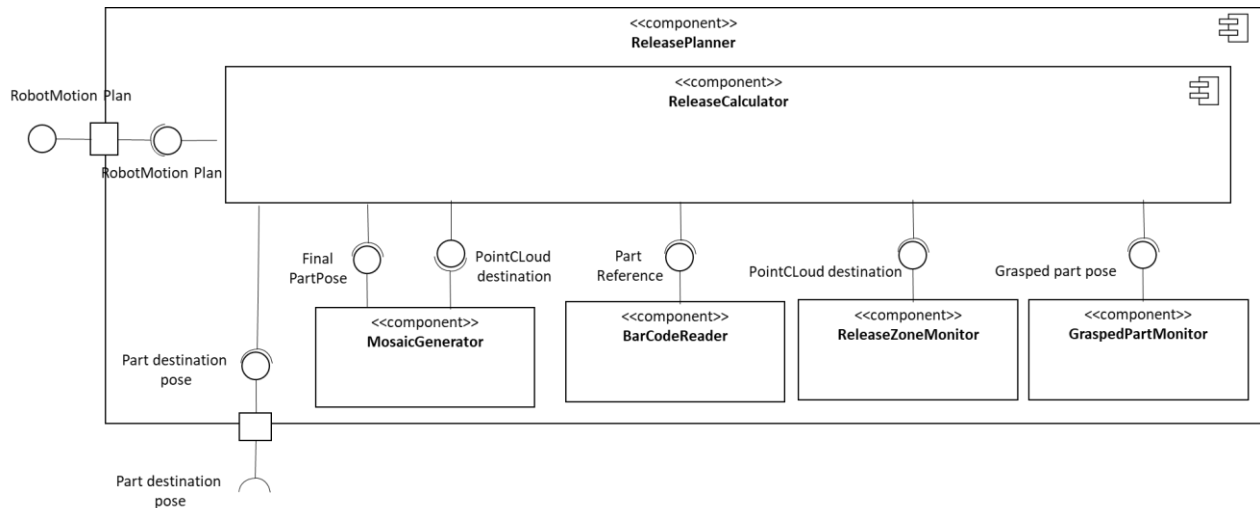


Figure 14; HARTU FBB - Application Layer – ReleasePlanner Component Diagram

4.6.2.6 AdaptiveMPC

Adaptive MPC is a torque-based robot controller that adapts its control parameters to the current situation using Gaussian Mixture Regression (GMR)¹¹. The current situation can be deduced from the given task and the perceived contact forces using ART-based contact classification.

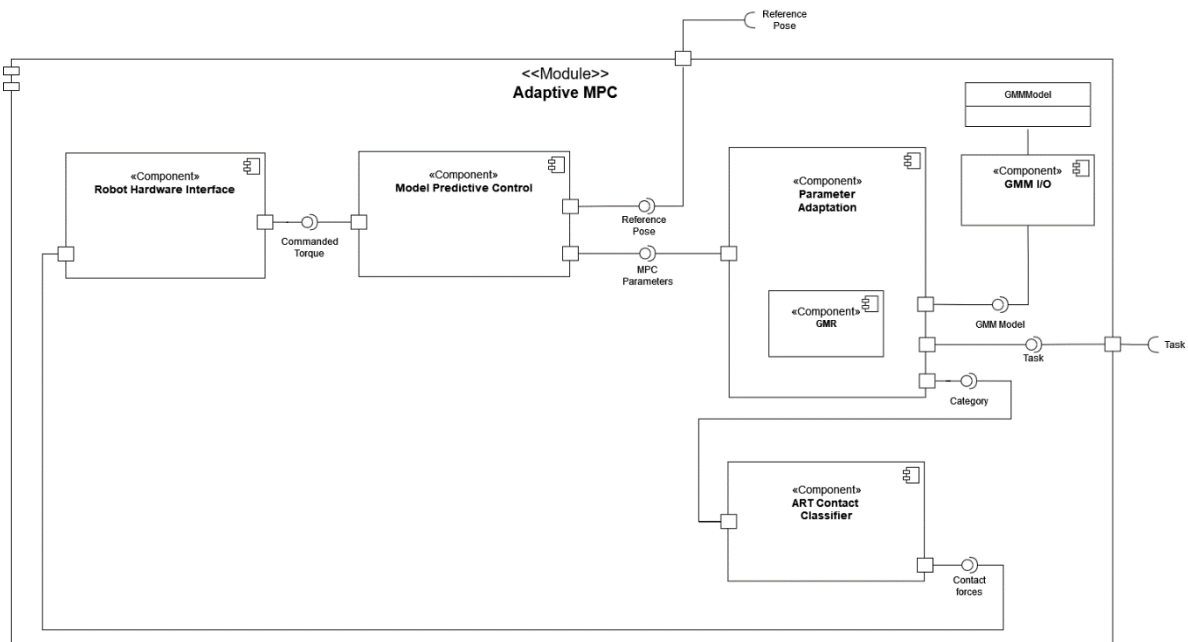


Figure 15: HARTU FBB - Application Layer – Adaptive MPC Component Diagram

Interfaces:

- [In] Reference Pose: Desired position/orientation of the robot as continuous data stream
- [In] Task: The current assembly task at hand. Will be used to select the appropriate set of control parameters.

¹¹ <https://github.com/AlexanderFabisch/gmr>

4.6.2.7 Imitation Learning

Imitation learning provides an architecture for intuitive specification and execution of assembly tasks on robotic manipulators (single- or dual-arm), i.e., tasks that are subject to complex contact forces with the environment. It can be operated in two modes: *Recording (Learning) Mode*, where the operator teaches a task, and the system generates a dynamic movement primitive (DMP), and *Execution Mode* where a pre-learned DMP is executed given a task specific goal pose, which typically is provided by the perception system of the robot.

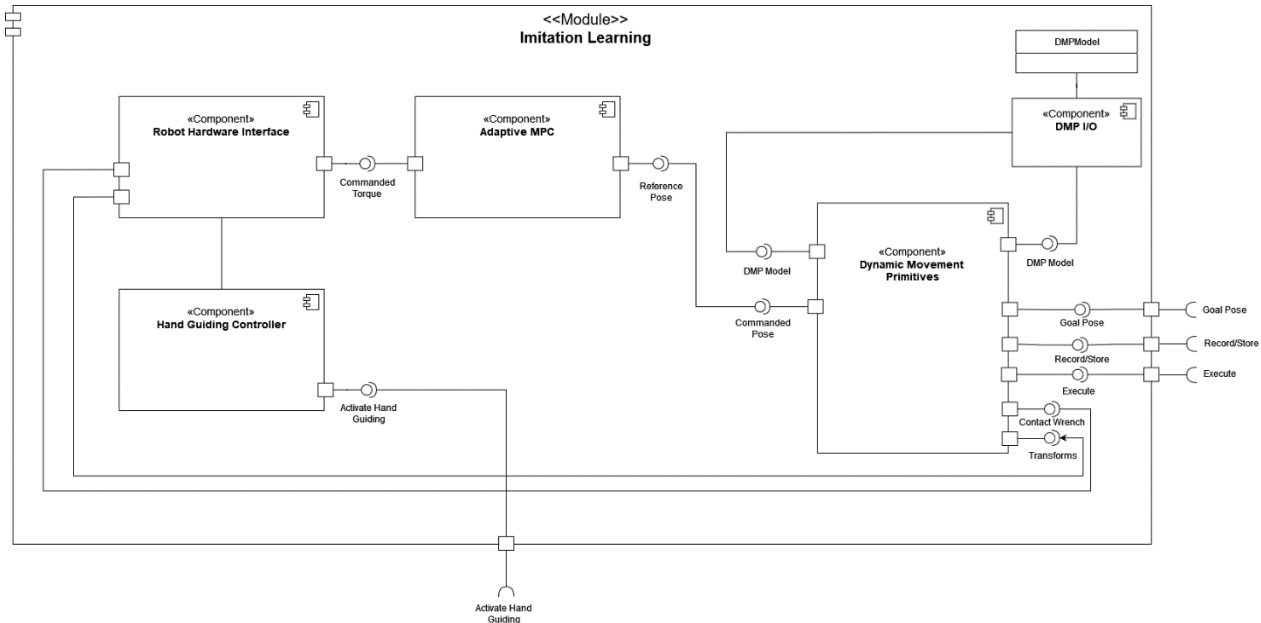


Figure 16: HARTU FBB - Application Layer – Imitation Learning Component Diagram

Interfaces:

- [In] Goal Pose: Target pose of the DMP, provided by the perception system.
- [In] Record/Store: Trigger to start recording of the trajectory.
- [In] Execute: Trigger to start execution of an assembly task.
- [In] Activate Hand Guiding: Trigger to activate hand guiding mode.

4.6.3 Information Layer FBBs Specification

The Information Layer contains one functional block or Data & Models (MOD.IL.DEM). This block interacts with other Application Layer’s functional blocks and with the components of each block using its data models as input to perform certain operations or for configuring the component to start its operational phase within the system. Figure 17 shows the internal structure of the component and the main relationships with the components of the Application Layer blocks:

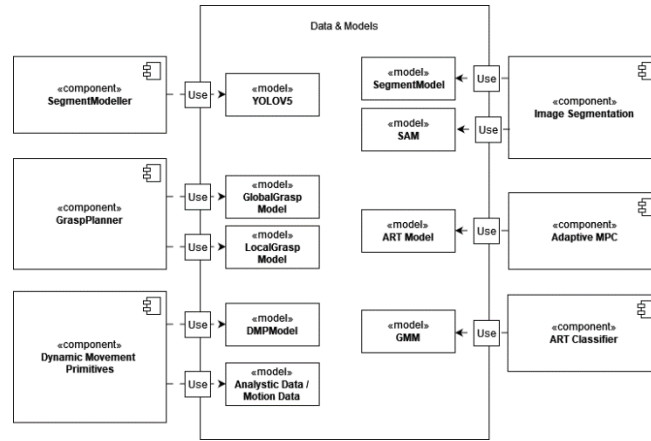


Figure 17: HARTU FBB – Information Layer – Components Diagram

4.6.3.1 HARTU_SegmentModel

It is the result of training YOLOv5 with the set of images created for a specific part reference, by the **SegmentModeller** component. Later, the ImageSegmentation component uses this model as input for the SAM model that, finally, provides the segmented image.

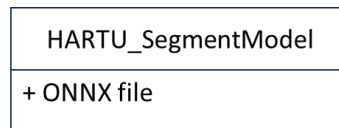


Figure 18: HARTU FBB - Information Layer – HARTU_SegmentModel

4.6.3.2 LocalGraspModel

The LocalGraspModel contains the information of the valid grasping points for a given part reference and gripper. It is created by the LocalGraspModeller offline, once per object.

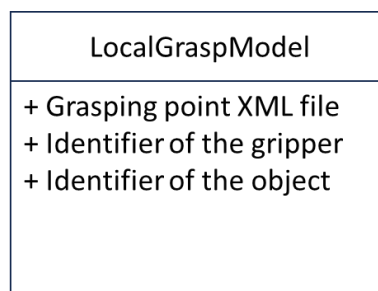


Figure 19: HARTU FBB - Information Layer – LocalGraspModel

4.6.3.3 GlobalGraspModel

This model includes the configuration for two different Global grasping strategies:

- The weights for the heuristic version
- The weights of the trained DRL model for the AI-based version

Additionally, the model includes the Identifier of the end effector and others that will be defined during the development of the GlobalGraspModeller (scheduled by M24).

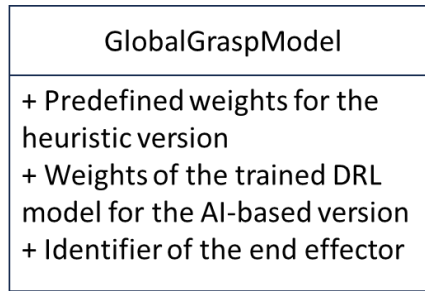


Figure 20: HARTU FBB - Information Layer – GlobalGraspModel

4.6.3.4 HARTU_PoseEstimationModel

This includes the configuration of model parameters and the weights of the trained models. The configurations are included in YAML files, while the model weights are saved as an ONNX model.

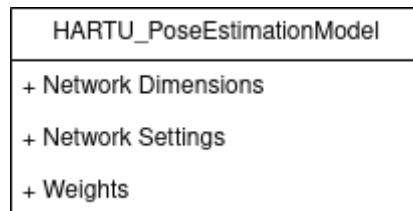


Figure 21: HARTU FBB - Information Layer – HARTU_PoseEstimationModel

4.6.3.5 HARTU_DMPModel

Represents the parameters of a dynamic movement primitive, e.g., dimension, weights of the forcing term, execution time, etc. YAML files are used for disk I/O.

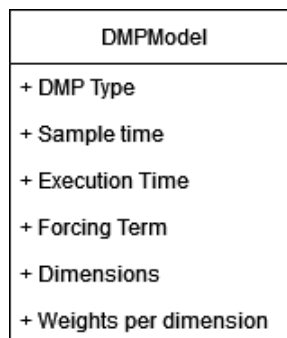


Figure 22: HARTU FBB - Information Layer – DMPModel

4.6.3.6 HARTU_ARTModel

Represents the parameters of the neural networks used inside the ART classifier, in principle only the weights of the neurons. MessagePack¹² is used for disk I/O.

¹² <https://msgpack.org/>

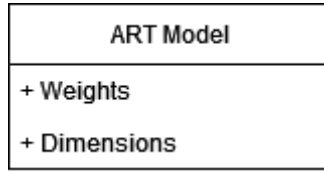


Figure 23: HARTU FBB - Information Layer – ART Model

4.6.3.7 HARTU_GMMModel

Represents the parameters of a Gaussian Mixture Model, i.e., priors, means, covariances of the Gaussians. According to the implementation by Fabisch¹³. Pickle¹⁴ is used for disk I/O.

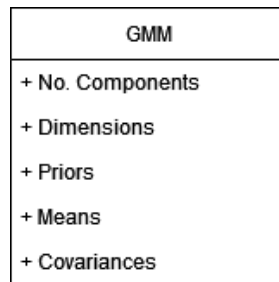


Figure 24: HARTU_GMMModel

4.6.4 Middleware Layer FBBs Specification

The Middleware Layer contains one functional block or HARTU Client Library (MOD.ML.CLL). Mainly this block contains low-level components native to the ROS operating system, which interact directly with the robot and in general with the physical part of the system.

4.6.4.1 Adaptive Resonance Theory (ART) Classifier

ROS2 component for classification of contact situations based on proprioceptive robot sensors.

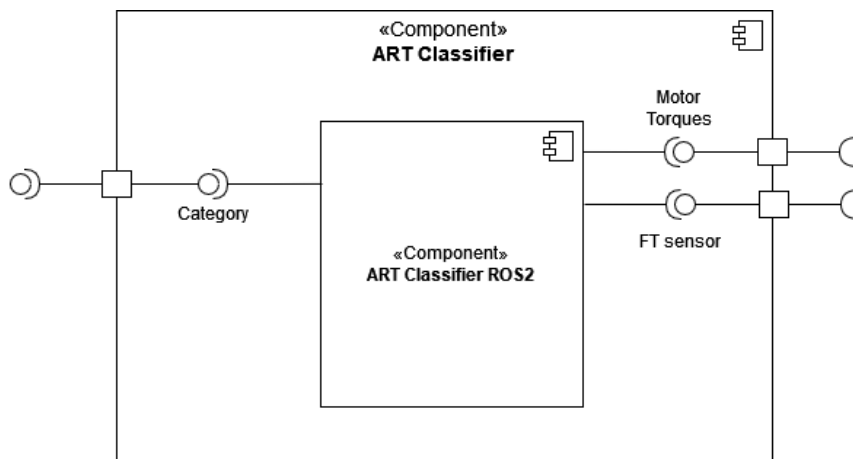


Figure 25: HARTU FBB - Middleware Layer – ART Classifier Component Diagram

Interfaces:

¹³ <https://github.com/AlexanderFabisch/gmr/>

¹⁴ <https://docs.python.org/3/library/pickle.html>

- [In] Joint torques: Measured motor torques of the manipulator.
- [In] Contact wrenches: Measured force/torque at the end effector.
- [Out] Category: Predicted label of the contact situation.

4.6.4.2 Inverse Reinforcement Learning

The component for inverse reinforcement learning module has not yet been specified, as the corresponding work has not started yet.

4.6.4.3 Dynamic Movement Primitives

ROS2 component for recording, storing, and executing single- or dual-arm Dynamic Movement Primitives (DMP). It uses an open-source movement primitives library provided by DFKI¹⁵

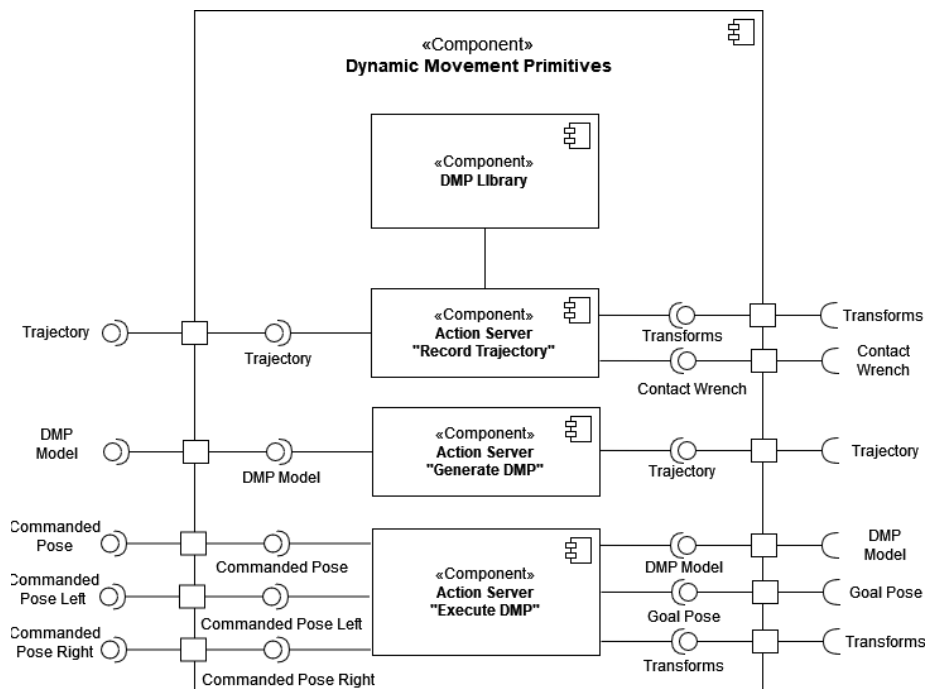


Figure 26: HARTU FBB - Middleware Layer – Dynamic Movement Primitives Component Diagram

Interfaces:

- [In] *Record Trajectory*: ROS2 action server which records the pose of the robot end effector(s) and stores it as a time-dependent trajectories.
- [In] *Generate DMP*: ROS2 action server which generates a DMP from the recorded data and stores it as yaml-file.
- [In] *Execute DMP*: ROS2 action server which executes the generated DMP point-by-point given the initial and desired final pose.
- [In] *Transforms*: The ROS2 robot frame transformation tree.
- [In] *Contact Wrench*: The force/torque measured at the robot end effector.

¹⁵ https://github.com/dfki-ric/movement_primitives

- [Out] *Commanded Pose, Commanded Pose left/right*: Next end effector pose(s) computed by the DMP.

4.6.4.4 Model Predictive Controller

A robot controller which uses optimization to regulate one or multiple tasks while respecting the physical constraints of the robot and the environment. It is based on the Crocoddyl library¹⁶ for optimal multi-contact point control.

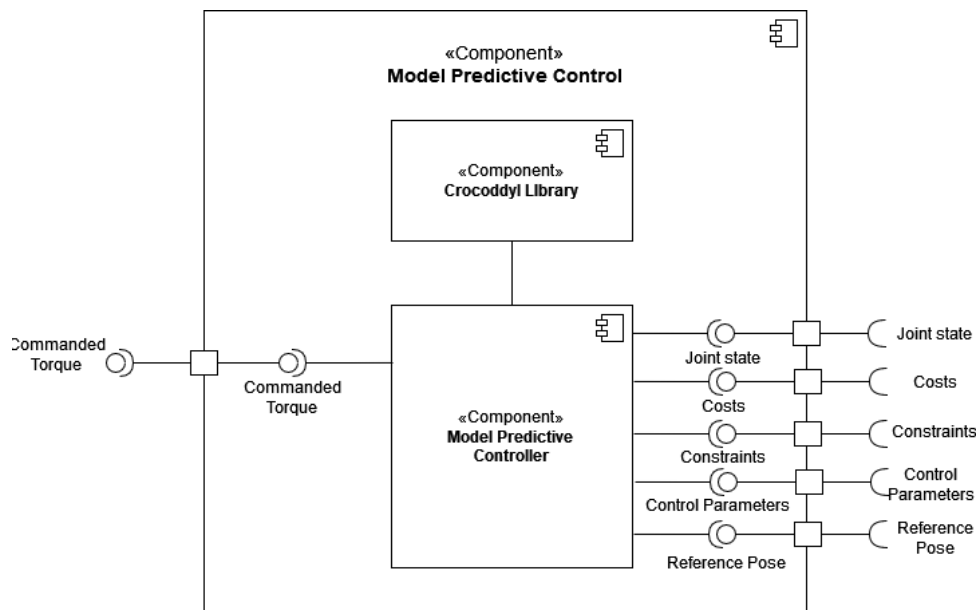


Figure 27: HARTU FBB - Middleware Layer – Model Predictive Control Component Diagram

Interfaces:

- [In] *Reference Pose*: Desired end effector position/orientation
- [In] *Constraints*: Constraints of the physical system to control.
- [In] *Costs*: Cost functions to optimize.
- [In] *Control parameters*: E.g., stiffness and damping.
- [Out] *Commanded Torques*: Joint torques to achieve all desired tasks while respecting the physical constraints.

¹⁶ <https://github.com/loco-3d/crocoddyl>

4.6.4.5 Behaviour Trees

ROS2 component that allows the execution of a behaviour tree previously created through the AppManager (Builder).

- It makes use of the behaviortreecpp_v3 library.
- It takes as input the AppConfiguratiOnFile generated with the AppManager (Builder).
- The BTExecutor is the component in charge of loading our custom nodes using the behaviortreecpp_v3 library and the structure of the tree defined in the AppConfiguratiOnFile.
- Then, the BTExecutor sends the ticks to the tree in a main loop and verifies the result (Success or Failure).

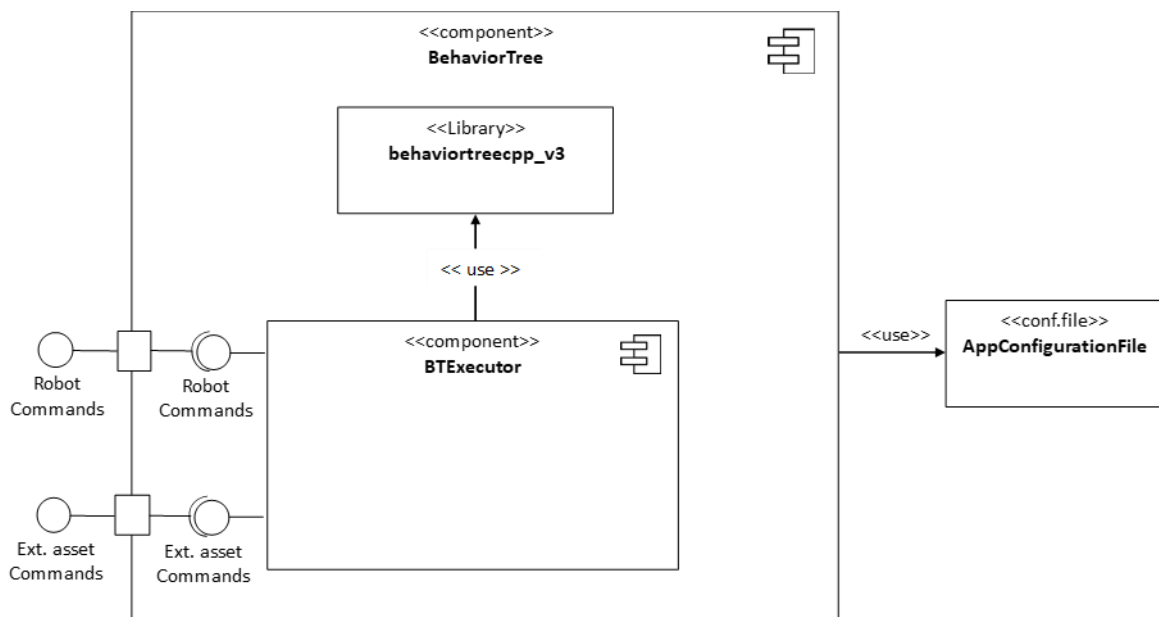


Figure 28: HARTU FBB - Middleware Layer – BehaviorTree Component Diagram

4.6.4.6 Deep Reinforcement Learning

The component is used by the GlobalGraspPlanner to define the model to select the part to be picked in a cluttered scene.

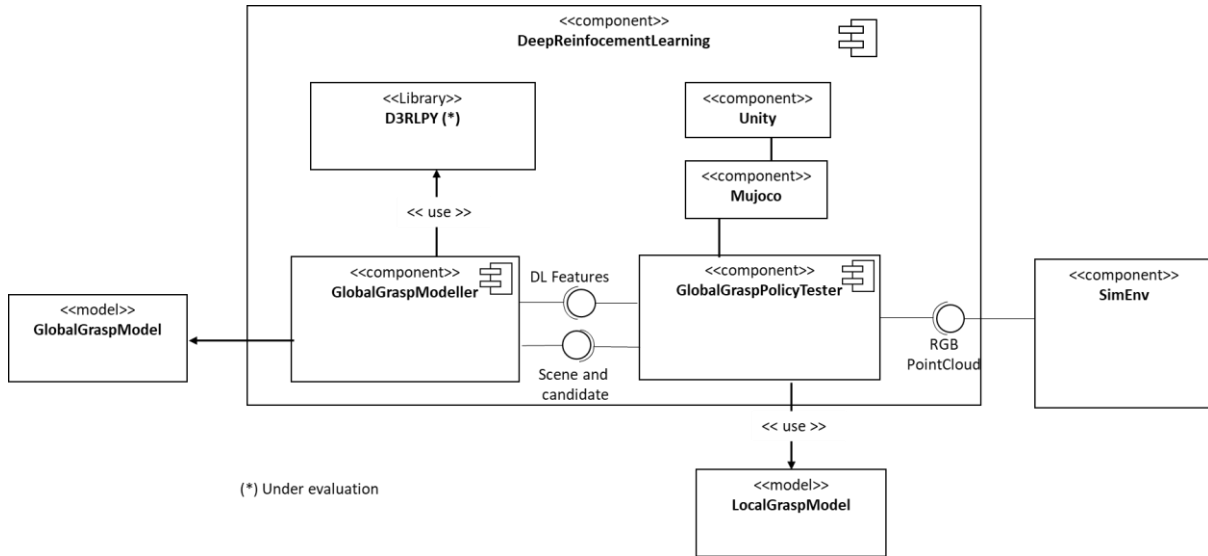


Figure 29: HARTU FBB - Middleware Layer – DeepReinforcementLearning Component Diagram

4.7 Requirements Traceability Matrix

The traceability matrix maps in a table the identified Functional Building Blocks (FBB) and presented in section 4.6 of this deliverable with the requirements identified in section 9 of D1.1, highlighting which block implements and satisfy a specific requirement, as well as to give a clear understanding of each block objectives.

Some requirements contained in D1.1 are not contained in this table as they will not have an impact in terms of functionality in the demonstrators (thus, they will not be included in the prototypes). Furthermore, MOD.ML.CLL functional block has not been included in the table because it is a purely back-end and not directly connected to the requirements which are mostly satisfied in the application layer.

The following table also contains different types of requirements like User, Functional and Non-Functional ones:

Table 3. Requirements – Functional Building Blocks (FBB) traceability matrix

Requirements	MOD.FL.UIT	MOD.AL.PER	MOD.AL.APC	MOD.AL.GRP	MOD.IL.DEM
FR-01		X	X	X	X
FR-02		X		X	
FR-03		X	X	X	
FR-04		X		X	
FR-05		X	X		
FR-06		X	X		
FR-07		X			X
FR-08			X	X	X
FR-09		X			
FR-10			X		
FR-11			X		
FR-12		X	X	X	
FR-13		X			

FR-14		X	X	X	X
FR-15		X			
FR-16					
FR-17		X			X
FR-18		X	X	X	
FR-19			X		
FR-20		X			X
FR-21			X	X	X
FR-22			X		
UR-01			X		
UR-02	X				
UR-03	X		X		
UR-04	X		X		
UR-05	X	X	X		
UR-06	X	X			
UR-08	X				
NR-01		X	X	X	
NR-02			X		
NR-03		X	X	X	X
NR-04		X			X
NR-05	X	?	X		
NR-06		X	X	X	
NR-07		X	X		
NR-08		X	X	X	
NR-09			X		
NR-10		X			
NR-13		X	X	X	
NR-14		X	X	X	
NR-15		X	X	X	
NR-16		X	X	X	X
NR-18		X			X
NR-19			X		
NR-20		X	X	X	
NR-21		X		X	
NR-22			X		
NR-23		X		X	
NR-24		X	X	X	X
NR-25		X	X	X	
NR-26		X	X		X
NR-27			X		
NR-28		X	X	X	X
NR-29		X	X	X	
NR-30			X	X	

5 HARTU Integration Plan

The integration plan details the process that will be implemented to release all the software components in each FBB. The plan is related to the development phase of the component and not to the deployment.

The following software releases are considered:

- **Alpha release** (a preliminary version of software component)
- **Beta release** (a more mature version of software component)
- **Final release** (a consolidated version of software component)

As a second step, the integration will detail the software type for each single component, both in case of “internal” component behaviour (e.g., porting from ROS to ROS2), and if integration will rely on external components. They will also be provided guidelines for component development and the SCM process. The environment that will be used for software change management will be Gitlab¹⁷.

5.1 General Integration Status

The General Integration status contains the month (M) when an entire functional block will be released and made available for system and integration test between software components or ready to be used and tested by end users. It includes three different versions: Alpha, Beta and Final release.

The following table shows “when” in the project, each release will be available for test or integration. Each release will contain features consistent or not depending on the status (alpha-beta-final):

Table 4 HARTU FBB General Integration Status.

FBB	Related Use Case	Release		
		Alpha	Beta	Final
<i>MOD.FL.UIT</i>	UC#1, UC#2,	M18	M24	M30
<i>MOD.AL.PER</i>	UC#3, UC#4,	M18	M27	M30
<i>MOD.AL.APC</i>	UC#5, UC#6,	M18	M24	M30
<i>MOD.AL.GRP</i>	UC#7	M18	M24	M30
<i>MOD.IL.DEM</i>		M14	M24	M30
<i>MOD.ML.CLL</i>		M14	M18	M24

5.2 Component Detailed Plan

The Component Plan shows an overview of all the features provided for by the HARTU system and the type i.e., UI, service, module, model, etc. Furthermore, if the component needs to be integrated with another component (within the same FBB block, or between components of different blocks), it reports the component with which it will be integrated.

¹⁷ <https://about.gitlab.com/>

This information is reported in the following table:

Table 5. HARTU FBB Component, Detailed Integration Plan

Component	Reference Component	Integration Description	Features				
			GUI	Back-end service	AI Module	AI Model	Data store
Adaptive Model Predictive Control	Robot Hardware Interface	ROS2, joint-level commands (position/torque)			X		
Dynamic Movement Primitives	Adaptive Model Predictive Control	ROS2, Cartesian commands (Pose, Twist)			X		
Hand Guiding Controller	Robot Hardware Interface	For user-demonstration / teach-in, hardware-specific		X			
ART Classifier	Model Predictive Control	ROS2, Via Parameter adaptation component			X		
Pose Estimation	Imitation Learning	ROS2, Perception delivers goal pose for DMP			X		
<i>App Manager (Builder)</i>	LocalGraspModeler	Launches the Component ROS 2 node) and communicates through ROS2 interfaces				X	
<i>App Manager (Builder)</i>	SegmentModeller,	Launches the Component (ROS 2 node) and communicates through ROS2 interfaces				X	
<i>App Manager (Builder)</i>	SyntheticImageDataGenerator and Unity	UNITY Application		X			
<i>App Manager (Builder)</i>	LocalGraspPointTester	Launches the UNITY App. using MuJoCo as physics engine.				X	
<i>App Manager (Builder)</i>	GlobalGraspPolicyTester	Launches the UNITY App. using MuJoCo as physics engine.				X	

<i>App Manager (Control)</i>	FBBs in the Application Layer	Communicates with the components when required using ROS2 interfaces	X				
<i>ImageAcquisition</i>	CameraNode and CameraAPI	Integration through ROS2 service		X			
<i>GraspPlanner</i>	PoseEstimation, ImageSegmentation	Integration through ROS2 interfaces		X			
<i>ReleasePlanner</i>	MosaicGenerator, BarCodeReader, ReleaseZoneMonitor, GraspedParMonitor	Integration through ROS2 interfaces		X			
<i>Pose Estimation</i>	Perception	Integration as a ROS2 service		X			

5.3 Software Configuration Management

Software Configuration Management (SCM) is a set of processes, policies, and tools used to manage changes to software systems. It encompasses version control, change tracking, release management, and other activities aimed at ensuring the integrity and consistency of software throughout its lifecycle.

The subsequent sections delineate the structure of this type of activities and detail the efficient configuration of tools to facilitate collaboration among distributed development teams. This aims to precisely plan and monitor the coordinated progress, quality, and integration status of various components.

5.3.1 Source code repository

The adoption of a Source Code Version Control tool brings several benefits to software development teams. It offers a set of tools that facilitate and document changes in the source code, offering a centralized repository for storage. These tools prevent conflicts by restricting simultaneous editing of the same module by multiple SW developers, ensuring a smooth workflow. Every modification made to the source code is recorded, allowing for comprehensive tracking. SCM tools empower SW developers to check out modules from the repository, make and document changes, and then save the edited modules back to the repository. The ability to discard changes, when necessary, provides flexibility, allowing a return to a previous baseline. Advanced SCM tools go beyond basic functionalities, supporting parallel development and accommodating geographically dispersed teams, like the scenario of European projects such as this one.

In HARTU project, GitLab has been selected due to its assortment of complimentary tools designed to assist developers throughout the entire software development lifecycle, of which the most

relevant are outlined below. Also, GitLab is a leader in the Gartner Magic Quadrant for DevOps Platforms¹⁸.

1. GitLab permits the creation of both **private and public repositories**, and **it's free** for both.

Prior to open sourcing the code developed for HARTU, it is advisable for development teams to utilize an exclusive sandbox environment, enabling operations within a private repository.

2. GitLab offers an internal, **integrated tool for CI/CD/CT** that leverages the capabilities of Docker container technology.

This represents one of the most important features as it allows the adoption of simple, powerful build & test automation software, not split among different tools.

3. It offers all the elements needed for the entire DevOps lifecycle within a **unified framework**.

GitLab provides various valuable features, such as the "**Issues**" feature, which allows the team to monitor features and bugs associated with each release of the components developed. In this context, the term "team" is used in a broader sense, encompassing not only SW developers but also domain experts who contribute to establishing requirements for ongoing enhancements.

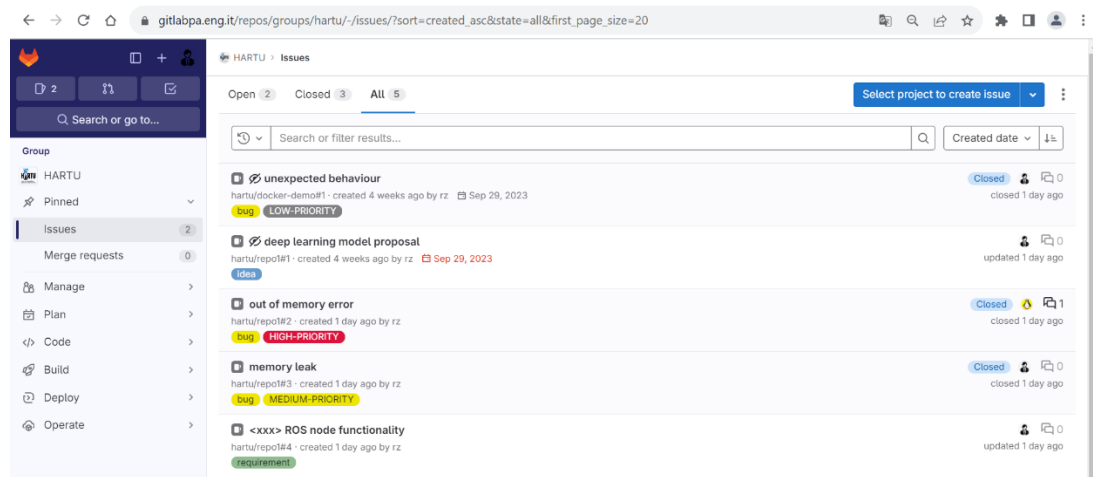


Figure 30: GitLab "Issues" feature example

4. Self-Managed¹⁹ Core version aka Community Edition (CE) is **open sourced**²⁰.

Having the traditional advantages of an open-source product, like cost-effectiveness and freedom from vendor lock-in, adds valuable flexibility.

Hence, to capitalize on these capabilities, an SCM system based on GitLab has been established and is configured and managed by Engineering. The system also features a dedicated server which runs a **GitLab Runner**²¹ instance configured for executing the jobs of CI/CD pipeline, as elaborated in the subsequent sections, starting with the building stage.

¹⁸ <https://www.gartner.com/doc/reprints?id=1-2DHNOAC7&ct=230504&st=sb>

¹⁹ <https://about.gitlab.com/pricing/#self-managed>

²⁰ <https://gitlab.com/rluna-gitlab/gitlab-ce>

²¹ <https://docs.gitlab.com/runner/>

To initiate the use of this system, which is open for utilization upon request, project partners only need to furnish their email addresses and inform Engineering about their intended role, which could be, for example:

- Developer
- Tester
- Product owner (e.g., pilot domain expert)

Then, the partner will receive an e-mail with instructions to setup autonomously the account created, as in the following example:

```
Hello <user>!  
The Administrator created an account for you.  
Now you are member...  
Login .....<your-email>  
Click here to set your password  
This link is valid for 2 days.  
After it expires, you can request a new one.
```

5.3.2 CI/CD setup and automation

Continuous Integration (CI) is a collaborative software development practice where team members integrate their work frequently, typically on a daily basis, ensuring multiple integrations per day. Automated builds, including tests, are performed with each integration to swiftly identify and address integration errors. This approach has proven to significantly reduce integration issues and accelerate the development of cohesive software for many teams. Therefore, the adoption of this practice is delegated to each laboratory, tailored to its individual repository and team. However, the tools and processes established and provided within the HARTU project facilitate this implementation and coordination between different laboratories and SW development teams. In fact, every member of these teams is motivated by the presence of these tools and guided by agile best practices and basic guidelines -as those outlined at §5.3.4- to regularly submit their code (e.g. push to the remote repository). This proactive approach aims to prevent the occurrence of conflicts arising from misalignments in source code commits whose resolution can be both frustrating and time-consuming.

The initial measure taken to facilitate Continuous Delivery (CD) involved automating the build process for modules. This is achieved by executing the associated Docker files, where adopted, which comprise the necessary instructions for constructing each Docker image. This automation is initiated with every push to the remote repository, resulting in the generation of an internal build as depicted in the picture below.

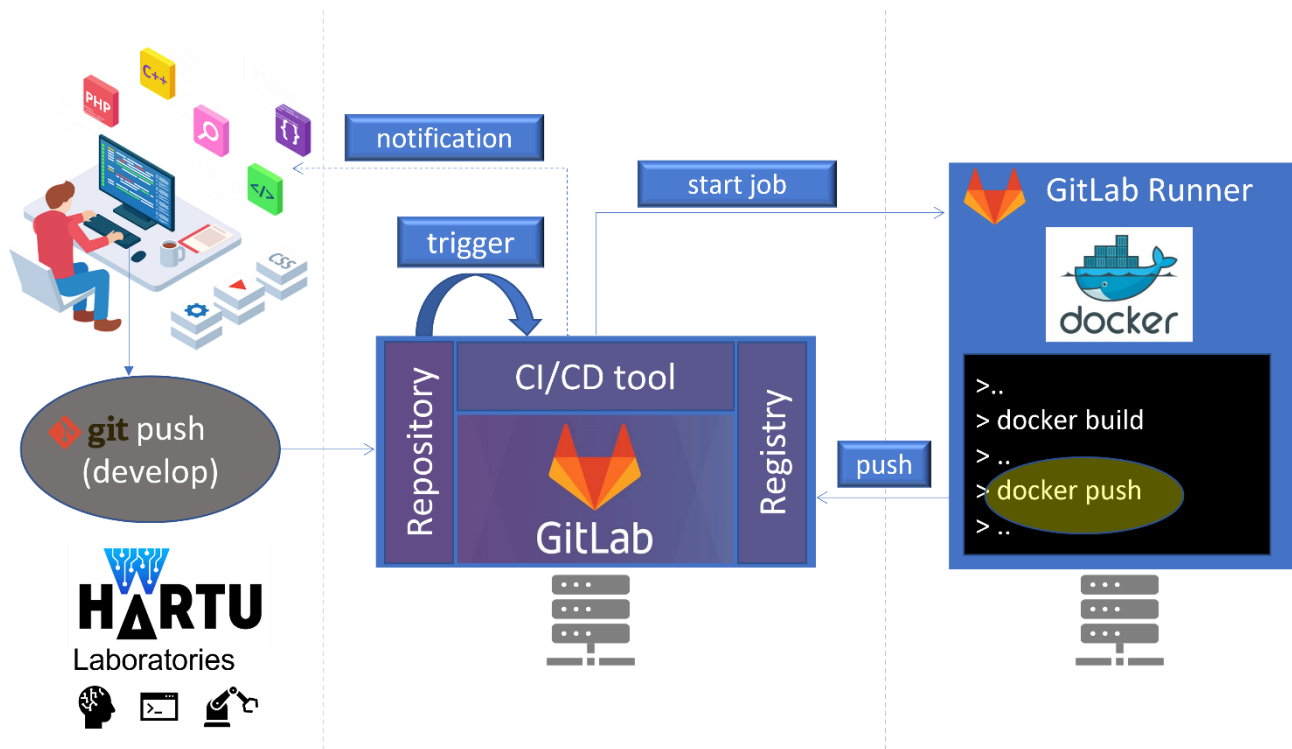


Figure 31: Docker based build process automated with GitLab in HARTU

Starting from the left, when a developer pushes to the 'develop' branch of the HARTU repository, this action triggers a job execution in GitLab CI/CD pipeline. The job is pre-configured and relies on GitLab Runner, as introduced previously in §5.3.1, a component dedicated to executing jobs for different stages, such as the “build” one, dependent on project-specific configurations. In this workflow, the Docker-based Runner, configured at the group level to be accessible across all HARTU repositories within the project (Figure 32), automatically downloads the latest source code version from the GitLab server. Subsequently, it initiates the building of the Docker image for the specific component. If errors occur during the build, the job is halted, and an automatic notification is sent back to the developer via the GitLab server.

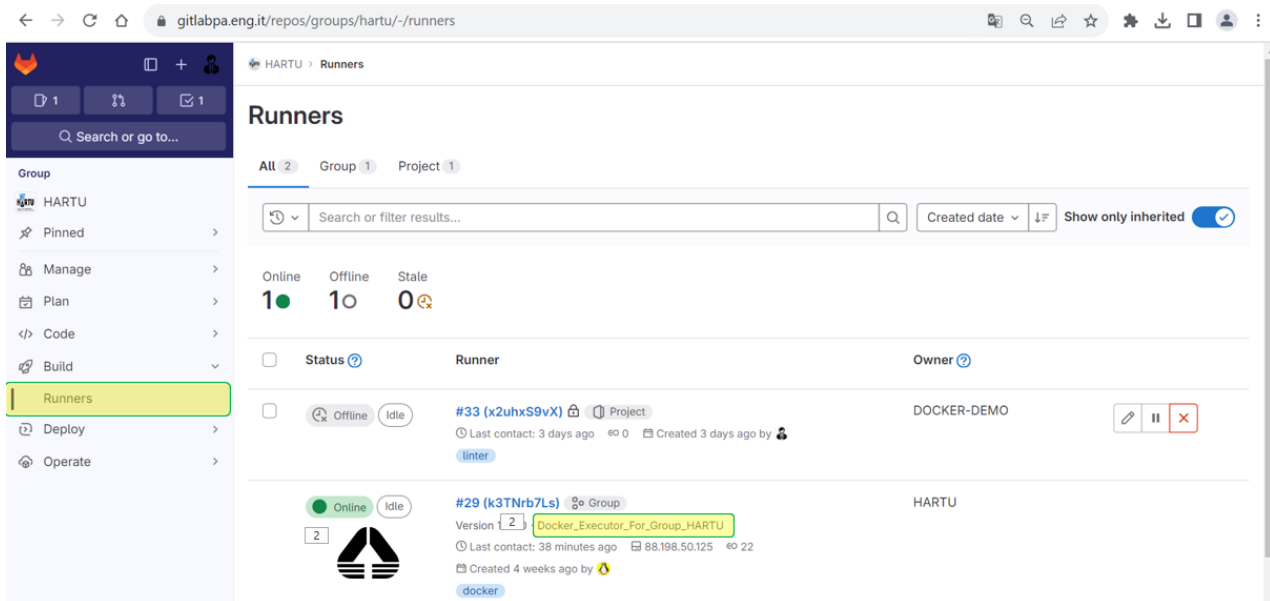


Figure 32: GitLab Runner setup for HARTU repositories

Upon the successful completion of image construction, the Docker image is pushed to the internal private registry²², a service provided with the GitLab server which significantly simplifies the continuous delivery (CD) in the early stages, at least. In fact, this internal registry makes the Docker image of a HARTU component accessible for download and testing potentially in any remote locations, such as the different HARTU laboratories.

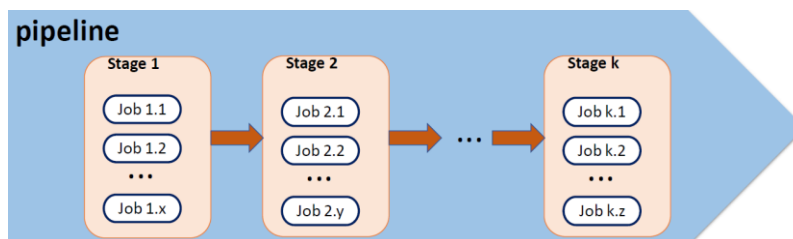


Figure 33: GitLab pipeline basic representation

The picture above shows the basics of a GitLab pipeline, which are:

- Jobs are the fundamental elements of a CI/CD pipeline as contain the instructions of the actions to be performed.
- Jobs are grouped in stages determining when they must be executed.
- One stage can start only if all the jobs of the previous stage have been completed successfully.
- Jobs belonging to one stage can be executed in parallel.
- Each job is executed by a **GitLabRunner executor** running outside the GitLab server.

The following picture illustrates the pipeline definition configured for a SW component that can be Dockerised for the deployment.

²² <https://docs.docker.com/registry/>

```

 8  stages:      # List of stages for jobs, and their order of execution
13  build-job:   # This job runs in the build stage, which runs first.
14      stage: build
15
16      tags:
17      - docker
18
19      image:
20      name: gcr.io/kaniko-project/executor:v1.14.0-debug
21      entrypoint: [""]
22
23      script:
24      - echo "Building docker image"
25      - /kaniko/executor
26      - --context "${CI_PROJECT_DIR}"
27      - --dockerfile "${CI_PROJECT_DIR}/docker/Dockerfile"
28      - --destination "${CI_REGISTRY_IMAGE}:latest"
29      - echo "Docker image built and now available at ${CI_REGISTRY_IMAGE}:latest"
30

```

Figure 34: `.gitlab-ci.yml` -> pipeline definition to automate Docker build process within GitLab

5.3.3 Quality control

The quality control of the source code, a good practice also known as code-review, consists of verifying lines of code written against rules which are defined by coding style conventions; different conventions are available for any programming language, such as those listed and made publicly available by Google style guides²³.

On one hand, this good practice primarily ensures the seamless maintainability²⁴ of the source code, encompassing changeability, modularity, understandability, testability, and reusability. On the other hand, by proactively identifying bugs, undefined behaviour, and risky coding constructs, this type of test contributes to improving program execution and, consequently, enhances quality at the runtime level.

In the contemporary landscape, developers have access to various tools and platforms that facilitate the automatic verification of coding standards and conventions, ensuring readability and monitoring code complexity within acceptable bounds. Consequently, significant attention has been directed toward determining the coding standards, conventions, and best practices for the APRO modules, along with the supporting tools necessary for implementation.

These tools predominantly function as linting utilities, conducting a form of static analysis aimed at identifying problematic patterns or weaknesses, such as those outlined in the Common Weakness Enumeration (CWE)²⁵, and ensuring code adherence to specific style guidelines.

²³ <https://github.com/google/styleguide#google-style-guides>

²⁴ <https://www.it-cisq.org/standards/code-quality-standards/>

²⁵ <https://www.it-cisq.org/pdf/cisq-weaknesses-in-ascqm.pdf>

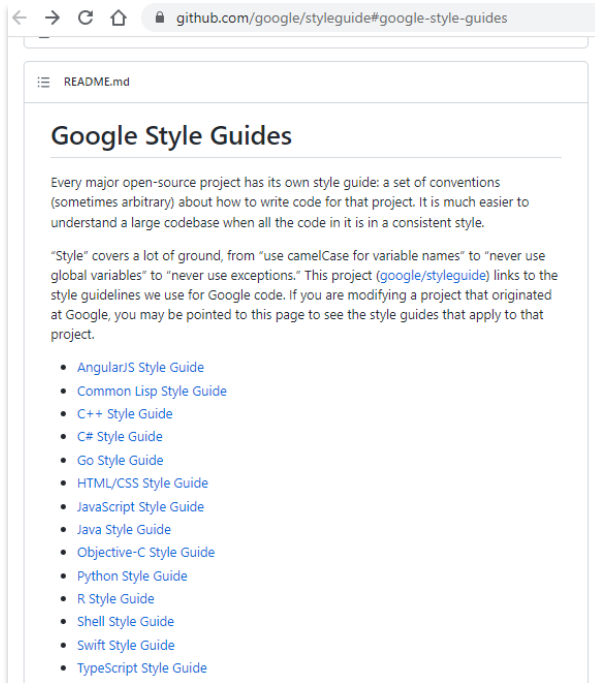


Figure 35: excerpt of Google Style Guides



CWE #	Descriptor	Weakness Description
CWE-407	Algorithmic Complexity	An algorithm in a product has an inefficient worst-case computational complexity that may be detrimental to system performance and can be triggered by an attacker, typically using crafted manipulations that ensure that the worst case is being reached.
CWE-478	Missing Default Case in Switch Statement	The code does not have a default case in a switch statement, which might lead to complex logical errors and resultant weaknesses.
CWE-480	Use of Incorrect Operator	The programmer accidentally uses the wrong operator, which changes the application logic in security-relevant ways.
CWE-484	Omitted Break Statement in Switch	The program omits a break statement within a switch or similar construct, causing code associated with multiple conditions to execute. This can cause problems when the programmer only intended to execute code associated with one condition.
CWE-561	Dead code	The software contains dead code that can never be executed. (Thresholds are set at 5% logically dead code or 0% for code that is structurally dead. Code that exists in the source but not in the object does not count.)
CWE-570	Expression is Always False	The software contains an expression that will always evaluate to false.

Figure 36: excerpt of Common Weakness Enumeration (CWE)

A linter, as a static code analysis tool, plays a crucial role in scrutinizing source code to flag potential issues, including programming errors, bugs, stylistic errors, and suspicious constructs. In the followings are briefly outlined relevant linters for the most widely used programming and scripting languages within the APRO components. Additionally, a table is included below (**¡Error! No se encuentra el origen de la referencia.**), mapping these languages to the respective components where they are going to be adopted.

- C++:

Cppcheck²⁶ is a static analysis tool tailored for C/C++ code, offering distinctive code analysis capabilities to identify bugs. Its emphasis lies in the detection of undefined behaviour and hazardous coding constructs, with the aim of minimizing false positives. Additionally, Cppcheck is engineered to analyze C/C++ code, even when it exhibits non-standard syntax, a feature particularly relevant in embedded projects.

- Python:

PEP8 Style Guide for Python Code²⁷ establishes coding conventions for Python code, encompassing the standard library included in the primary Python distribution.

- JavaScript:

ESLint²⁸ is a static code analysis tool designed to detect problematic patterns present in JavaScript code. The rules within ESLint are configurable, allowing for the definition and loading of customized rules. ESLint addresses both code quality and coding style issues.

²⁶ <https://cppcheck.sourceforge.io/>

²⁷ <https://peps.python.org/pep-0008/>

²⁸ <https://eslint.org/>

Table 6. APRO SW component - programming language map

Programming language \ APRO component	Python	C++	C#
APP Builder		X	
ImageAcquisition		X	
ImageSegmentation	X		
PoseEstimation	X	X	
ReleasePlanner	X	X	
SegmentModeller	X		
GlobalGraspPlanner	X	X	
GraspPlanner		X	
ART Classifier		X	
Dynamic Movement Primitives	X	X	
Adaptive MPC	X	X	
SimEnv	X		X

As previously mentioned, a linter is a tool used for coding reviews with the goal of enhancing code quality. Numerous linters are available for various programming languages, and many developers currently integrate these tools into their preferred IDEs. Some have automated them as an additional step in their CI processes, while others employ them in both ways, a practice being adopted and in preparation for APRO components.

To automate these checks within the APRO CI/CD process outlined in §5.3.2, the "Code Quality" feature of GitLab has been explored. This feature, along with its widget, facilitates the seamless integration of linters by executing them through the GitLab Runner. The GitLab Runner, as previously discussed, is a GitLab component dedicated to running code for building and testing stages based on specific configurations. In this case, existing configurations for the APRO CI/CD pipelines can be extended to allow linter execution.

Utilizing the GitLab "Code Quality" feature and its widget, linters based on Code Climate²⁹ engines³⁰ are employed. These components are modular plugins available for nearly any programming language, offering extensibility and open-source freedom. In essence, a Code Climate Engine is a Docker Image that invokes a program to parse a configuration file and analyse source code files, potentially generating formatted output indicating detected issues.

Through a wrapper within the GitLab Code Quality project³¹, these engines can be easily integrated and run within pipelines, starting from a Docker image built within the project itself. This image includes default Code Climate configurations, and only the Docker image(s) for the specific linter(s) need to be pulled. Default configurations can be overridden to meet the specific needs of each APRO

²⁹ <https://codeclimate.com/quality>

³⁰ <https://docs.codeclimate.com/docs/list-of-engines>

³¹ <https://gitlab.com/gitlab-org/ci-cd/codequality>

component (languages, types of quality checks) by creating dedicated config files for each APRO module repository.

The benefits of this solution are numerous, including:

- Integration into GitLab's pipelines³² in consistent alignment with the CI/CD workflow described in §5.3.2 and the Docker-based approach for deployments.
- Displaying a comprehensive list of code quality violations generated by a pipeline at the end of the process.
- Utilizing Code Climate Engines, which are free and open source.
- No subscription requirement.
- Extensibility with new plugins.

From a practical standpoint, to configure these engines for automatic execution in GitLab, in addition to having a pre-configured GitLab Runner for job execution, two files need setup. These files are the `.gitlab-ci.yml` (already present if a pipeline has been defined, such as for automating testing and building, as introduced in §5.3.2) and the `.codeclimate.yml`. Both of them must be placed in the root folder of the GitLab repository, and their content is briefly outlined as follows.

`.gitlab-ci.yml`:

The excerpt displayed in Figure 37 illustrates a section of this configuration file.

To execute the job involving the linters execution, three key configuration lines have been added, highlighted by as many blue bullets in the figure.

The first step involves incorporating the template, where a significant portion of the necessary work is already predefined.

```

1  variables:
5  include:
6  - template: Code-Quality.gitlab-ci.yml
7
8  stages:           # List of stages for jobs, and their order of execution
9  - build
10 - test
11 - deploy
12
13 build-job:       # This job runs in the build stage, which runs first.
14   stage: build
15
16   tags:
17     - docker
18
19   image:
20     name: gcr.io/kaniko-project/executor:vl.14.0-debug
21     entrypoint: [""]
22
23   script:
24     - echo "Building docker image"
25     - /kaniko/executor
26       --context "${CI_PROJECT_DIR}"
27       --dockerfile "${CI_PROJECT_DIR}/docker/Dockerfile"
28       --destination "${CI_REGISTRY_IMAGE}:latest"
29     - echo "Docker image built and now available at ${CI_REGISTRY_IMAGE}:latest"
30
31 code_quality:    #Declare a job with the same name as the Code Quality job, after the template's inclusion.
32   # overriding template:
33   services:      # Shut off Docker-in-Docker
34     stage: test
35   tags:
36     - linter     # Set this job to only run on the specialized runner tagged as "linter"
37   artifacts:
38     paths: [gl-code-quality-report.json]
39

```

Figure 37: extract from `gitlab-ci.yml` showing code-review setup in to a pipeline

Following this, it's essential to add a "test" stage and a job named "code_quality" to seamlessly integrate with the template. Within the "code_quality" job, settings can be configured to override defaults set in the template. For instance, to enhance security and performance³³, Docker-in-Docker is deactivated, and a custom tag (named "linter") is established to ensure this job runs exclusively on a private runner with a matching tag. Finally, the configuration specifies the file name and format

³² <https://docs.gitlab.com/ee/ci/pipelines/>

³³ https://docs.gitlab.com/ee/ci/testing/code_quality.html#improve-code-quality-performance-with-private-runners

for collecting job output, making it accessible to users and developers for download as an artifact from the GitLab GUI.

`.codeclimate.yml`:

This file³⁴, as shown in Figure 38, allows to specify the linters that will be applied to the source code, determining which quality checks will be carried out. Each enabled plugin in the configuration corresponds to a dedicated Docker image that is automatically fetched and executed as needed in the running job configured in the `.gitlab-ci.yml`. These specific plugins complement a set of general checks—currently, there are ten maintainability checks³⁵— which are enabled by default (and therefore not displayed in this file excerpt). These checks cover various types of measures, as follows:

1. **Argument count** (argument-count): methods or functions defined with a high number of arguments.
2. **Complex logic** (complex-logic): boolean logic that may be hard to understand.
3. **File length** (file-lines): excessive lines of code within a single file.
4. **Identical blocks of code** (identical-code): duplicate code which is syntactically identical (but may be formatted differently).
5. **Method complexity** (method-complexity): functions or methods that may be hard to understand (Cognitive Complexity).
6. **Method count** (method-count): classes defined with a high number of functions or methods.
7. **Method length** (method-lines): excessive lines of code within a single function or method.
8. **Nested control flow** (nested-control-flow): deeply nested control structures like if or case.
9. **Return statements** (return-statements): functions or methods with a high number of return statements.
10. **Similar blocks of code** (similar-code): duplicate code which is not identical but shares the same structure (e.g., variable names may differ).

```

1  ---
2  version: "2"
3  plugins:
4  csslint:
5  coffeelint:
6  duplication:
7  enabled: true
8  config:
9    languages:
10     #- ruby
11     #- javascript
12     - python
13     #- php
14  eslint:
15  fixme:
16  enabled: true
17  rubocop:
18  enabled: false
19  cppcheck:
20  enabled: true
21  pep8:
22  enabled: true
23  exclude_patterns:
24  - config/
25

```

Figure 38: extract from `codeclimate.yml`

5.3.4 Guidelines to ease collaboration

Some basic practical rules have been shared among the partners contributing to the development of APRO platform modules and their components. These rules are intended to ensure standardized and cohesive work between the different teams and include:

1. One single GitLab repo is dedicated to each HARTU module (set of components e.g. a FBB) in the HARTU Group.

³⁴ <https://docs.codeclimate.com/docs/advanced-configuration#section-configuration-formats>

³⁵ <https://docs.codeclimate.com/docs/maintainability#section-checks>

2. Each HARTU repo shall have mainly two branches, '**master**' and '**develop**' (developers can create further branches of 'develop', where needed).
3. Developers shall always `git push` to the '**develop**' branch (default choice for HARTU Group).
4. The '**master**' branch should be aligned (e.g. through a merge request) upon the release of a new version, adhering to the following rule.
5. The master branch should contain code that is stable and includes tagged version numbers.
6. **Naming convention:** The internal organization of each HARTU module repository should adhere to the structure illustrated in the screenshot below.

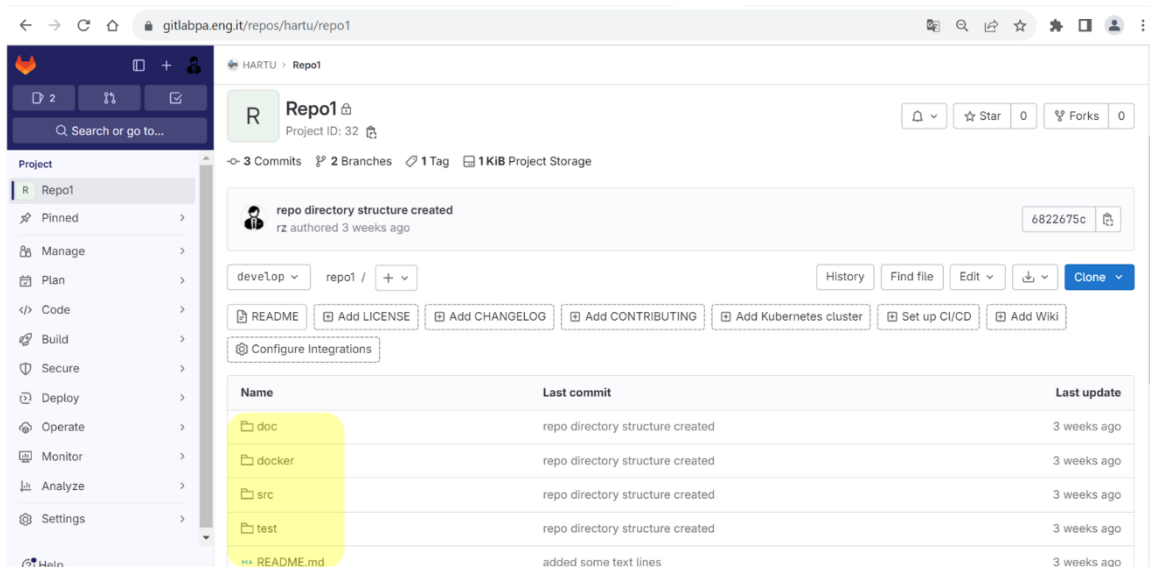


Figure 39: HARTU repositories naming convention

It's important to note that these simple rules should be considered a preliminary example for the stated purpose and are subject to change during the project as needed.

6 Conclusion

This deliverable contains the HARTU RA providing a complete and exhaustive design of the HARTU software solution, outlining also its implantation (to be further refined during the implementation phase in the technical WPs (i.e., in WP2, WP3 WP4 and WP5) before starting the piloting and validation activities within the scope of WP1. These results have been driven by the previous work and results of WP1, notably the analysis of requirements and reference applicative business scenario.

Key highlights include:

HARTU will provide automated AI based grasp and release planning, electro-active soft grippers and contact-rich assembly results for manufacturing scenarios ensuring compliance with ethics and legal requirements providing guidance on Human-AI teaming and defining user profile and skills for the new working scenario in Industry 5.0.

HARTU RA covers and specifies the functionalities that the software solution addresses in three different but interconnected areas: shopfloor, functional and technological.

HARTU RA provides simulation and adaptation capabilities in real and physical factory environments, including real-time operations considering the variability of the process that an operator is performing, considering the main variability that can affect the correct execution of a routine or the execution of a process pipeline.

The HARTU RA presents all the functional and technical features that the system will have to implement. The HARTU solution enables the execution of a wide range of business scenarios, building its foundation on principles of modularity, interoperability, scalability, flexibility, and adaptability.

This document also provides the basis for the implementation of the HARTU software solution and integration activities, that will be performed as part of the technical work packages, in particular WP2, WP3 and WP4. In particular, it defines the main components and structuring principles of the HARTU solution, also in terms of implementation, ensuring that interdependent activities can be streamlined in the best possible way, and in any case will provide a framework to cover the functional and technical requirements defined by the project partners.

Without any doubt, the design and implementation of HARTU RA will also guide the deployment and validation of use cases within WP1. As a rule, use cases may require some customizations, without however affecting the components of the solution described in this document. The identified components could be extended during solution development, as long as they remain backward compatible according to the specifications described in this document, thus not affecting the systems using the interfaces they expose.

However, design revisions may occur as a result of development and integration activities:

- new technologies selected to further improve the solution or a specific activity.
- given some changes in requirements and use cases.
- general evolution of the project.

This document will be kept alive, for future refinements and/or improvements (in terms of revisions) of the overall HARTU design solution and specifications, in order to report (if any) a continuous update of the internal structure of the blocks, their communication with other blocks, the necessity to add new components or new features for the whole system (e.g. if new requirements arise for use cases or during the testing phase to integrate new technologies to fully satisfy the results).

7 References

- [1] P. Adolphs and U. Epple, “Status Report: Reference Architecture Model Industrie 4.0 (RAMI4.0),” accessed: 2023-11-13. [Online]. Available: [GMA-Status-Report-RAMI-40-July-2015.pdf \(zvei.org\)](#).
- [2] [29] X. Ye and S. H. Hong, “Toward Industry 4.0 Components: Insights Into and Implementation of Asset Administration Shells,” IEEE Industrial Electronics Magazine, vol. 13, no. 1, pp. 13–25, Mar. 2019, doi: 10.1109/MIE.2019.2893397.
- [3] T. Miny, G. Stephan, T. Usländer, and J. Vialkowitsch, “Functional View of the Asset Administration Shell in an Industrie 4.0 System Environment,” 2021, accessed: 2023-11-13. [Online]. Available: <https://www.plattform40.de/IP/Redaktion/DE/Downloads/Publikation/Funktional-View.html>