SustainML

Application Aware, Life-Cycle Oriented Model-Hardware Co-Design Framework for Sustainable, Energy Efficient ML Systems

Framework architecture, back-end, front-end design and release - M24

Deliverable D5.2

WP5 - Framework Architecture and Implementation

This project has received funding from the European Union's Horizon Europe research and innovation programme (HORIZON-CL4-2021-HUMAN-01) under grant agreement No 101070408

eProsima
The Middleware Experts

DFKI

Inria

upmem

RPTU Rheinland-Pfälzische Technische Universität Kaiserslautern Landau

IBM

Project

Title:	SustainML: Application Aware, Life-Cycle Oriented Model-Hardware Co-Design Framework for Sustainable, Energy Efficient ML Systems
Acronym:	SustainML
Coordinator:	eProsima
Grant agreement ID:	101070408
Call:	HORIZON-CL4-2021-HUMAN-01
Program:	Horizon Europe
Start:	01 October 2022
Duration:	36 months
Website:	https://sustainml.eu
E-mail:	sustainml@eprosima.com
Consortium:	**eProsima (EPROS)**, Spain **DFKI**, Germany **Rheinland-Pfälzische Technische Universität Kaiserslautern-Landau (RPTU)**, Germany **University of Copenhagen (KU)**, Denmark **National Institute for Research in Digital Science and Technology (INRIA)**, France **IBM Research GmbH**, Switzerland **UPMEM**, France

Deliverable

Number:	**D5.2**
Title:	**Framework architecture, back-end, front-end design and release - M24**
Month:	24
Work Package:	WP5 - Framework Architecture and Implementation
Work Package leader:	eProsima (EPROS)
Deliverable leader:	eProsima (EPROS)
Deliverable type:	Report (R)
Dissemination level:	Public (PU)
Date of submission:	30/09/2024
Version:	v2.2
Status:	Finished

Version history

Version	Date	Responsible	Author/Reviewer	Comments
v1.0	31/03/2023	eProsima	Mario Domínguez López	D5.1 - First Draft
v1.1	10/09/2023	eProsima	Jesús Poderoso Martín	D5.1 - Review
v1.2	13/09/2023	eProsima	Mario Domínguez López	D5.1 - Apply review suggestions
v1.3	23/09/2023	eProsima	Raúl Sánchez-Mateos Lizano	D5.1 - Final review
v2.0	16/09/2024	eProsima	Jesús Poderoso Martín	D5.2 - First Draft
v2.1	17/09/2024	eProsima	Jesús Poderoso Martín	D5.2 - 1st Review
v2.2	30/09/2024	eProsima	Raúl Sánchez-Mateos	D5.2 - Final Review

© 2024 SustainML | HORIZON-CL4-2021-HUMAN-01 | 101070408

Executive summary

SustainML project aims to develop a design framework and an associated toolkit, so-called SustainML, that will foster energy efficiency throughout the whole life-cycle of Machine Learning (ML) applications: from the design and exploration phase that includes exploratory iterations of training, testing and optimizing different system versions through the final training of the production systems (which often involves huge amounts of data, computation and epochs) and (where appropriate) continuous online re-training during deployment for the inference process. The framework will optimize the ML solutions based on the application tasks, across levels from hardware to model architecture. It will also collect both previously scattered efficiency-oriented research, as well as novel Green-AI methods. Artificial Intelligence (AI) developers from all experience levels can make use of the framework through its emphasis on human-centric interactive transparent design and functional knowledge cores, instead of the common blackbox and fully automated optimization approaches.

This report corresponds to *Deliverable D5.2 - Framework architecture, back-end, front-end design and release - M24* of the SustainML project. This deliverable establishes the software requirements and covers the detailed architecture and design for the different components comprised within the *SustainML Design Framework*, as well as the release procedure. This document will be updated to reflect the most recent state of the project as it progresses.

1 Revision update

Revision of the deliverable *D5.1 - Framework architecture, back-end, front-end design and release - M12*.

1.1 Major Changes

- **New Module Nodes**: Existing Nodes have been updated and new Application-level Requirements and Hardware constraints for inference Nodes have been created to fulfill partners' requirements. Refer to the SustainML Pipeline Overview.
- **Module Node data model update**: Several fields have been added to the existing Data Model, and new data models for the new Nodes have been included.
- **Orchestrator Node**: New *baseline* mechanism has been implemented in the Orchestrator Node to iterate through the SustainML Framework. Implementation has been adapted to Python so it can be instantiated as another node.
- **Front-end design**: New input fields have been included with the required information, and results are shown in a comparative manner.

1.2 Progress

- **Work Packages integration**: Some of the Module Nodes have been populated with partners' implementation, avoiding the mocked methods. The remaining Nodes are integration work-in-progress.

1.3 New modules

- **Application-level requirements Node**: Work Package 1 (DFKI) module node that provides a sequence of application-level requirements to be considered in the selection of the machine learning model.
- **Hardware constraints for inference Node**: Work Package 2 (RTPU) module node that represents the group of constraints defined (or not) by the user when describing the problem.
- **Model Provider Node**: This Node has been detached for the WP1, in such a manner both DFKI (Work Package 1) and KU (Work Package 3) need to collaborate to implement and integrate their work in the Model Provider Node.
- **Hardware Provider**: As the previous node, both RPTU (Work Package 2) and UPMEM (Work Package 2) need to collaborate to implement and integrate their WP work in the Hardware Provider Node.

© 2024 SustainML | HORIZON-CL4-2021-HUMAN-01 | 101070408

Contents

Executive Summary 4

1 Revision update 5
1.1 Major Changes . . . 5
1.2 Progress . . . 5
1.3 New modules . . . 5

Contents 7

Acronyms 8

2 Introduction 9
2.1 Work Packages and partners . . . 9
2.2 Purpose of this document . . . 10
2.2.1 Purpose of this document (M12 - Deprecated) . . . 11
2.3 Structure of this document . . . 13

3 Background context 14
3.1 Enabling technologies . . . 14
3.1.1 Data Distribution Service (DDS) . . . 14
3.1.2 eProsima Fast DDS . . . 15
3.1.3 Python Bindings . . . 17
3.2 Software requirements . . . 17
3.2.1 General requirements for all the software modules . . . 18
3.2.2 WP1 requirements . . . 18
3.2.3 WP2 requirements . . . 19
3.2.4 WP3 requirements . . . 19
3.2.5 WP4 requirements . . . 20
3.2.6 WP5 requirements . . . 20
3.3 Software requirements (M12 - Deprecated) . . . 20
3.3.1 General requirements for all the software modules (M12 - Deprecated) . . . 21
3.3.2 WP1 requirements (M12 - Deprecated) . . . 21
3.3.3 WP2 requirements (M12 - Deprecated) . . . 21
3.3.4 WP3 requirements (M12 - Deprecated) . . . 22
3.3.5 WP5 requirements (M12 - Deprecated) . . . 22

4 Framework architecture 23
4.1 API for the Software Modules . . . 23
4.2 Framework architecture (M12 - Deprecated) . . . 24

5 Software design 25
5.1 Design guidelines . . . 25
5.2 Back-End . . . 27
5.2.1 Module Nodes . . . 27
5.2.2 Orchestrator Node . . . 30
5.2.3 Orchestrator Python Module . . . 31
5.3 Front-End . . . 32

6 Release procedure 33
6.1 Versioning . . . 34

7 Results **35**
7.1 Module Nodes demonstrator . . . 35
7.1.1 Description . . . 35
7.1.2 Technologies used . . . 36
7.1.3 Experiments . . . 36
7.2 SustainML Framework demonstrator . . . 37
7.2.1 Description . . . 37
7.3 Orchestrator Python Node . . . 40

8 Conclusions **43**
8.1 Future work . . . 43

A Data Model **45**

B Data Model (M12 - Deprecated) **50**

C API usage examples for the software modules **54**

D API usage examples for the software modules (M12 - Deprecated) **62**

E Screenshots of the Module Nodes demonstrator **66**

References **67**

Acronyms

AI	Artificial Intelligence.
ALRN	Application-level Requirements Node.
API	Application Programming Interface.
CN	Carbontracker Node.
DDS	Data Distribution Service.
GUI	Graphical User Interface.
HWCFIN	Hardware Constraints for Inference Node.
HWPN	Hardware Provider Node.
IDL	Interactive Data Language.
ML	Machine Learning.
MLMMN	ML Model Metadata Node.
MPN	Model Provider Node.
ON	Orchestrator Node.
QoS	Quality of Service.
RC	Release Candidate.
UML	Unified Modeling Language.
WP	Work Package.

2 Introduction

The *SustainML* project aims to develop a design framework and associated toolkit to improve energy efficiency throughout the entire life cycle of machine learning applications: from the design and exploration phase to initial iterations of training, testing, and optimizing various system versions until final training in the production systems (often requiring large amounts of data, computations, and epochs) and (where appropriate) continuous online retraining of the inference process during deployment.

The developed framework (*SustainML Design Framework*) aimed to be developed in this project, will provide an energy-optimized hardware solution and the corresponding ML model for solving a desired machine learning problem formulated by the user.

The following sections describe in detail the SustainML project workforce and fundamentals.

2.1 Work Packages and partners

The *SustainML Project* is divided into a series of Work Package (WP) that are briefly described below:

WP1 *Energy Consumption Oriented ML Task Modeling* - Leader: DFKI. The aim of this WP is to develop means for parameterizing ML tasks (including associated user-defined boundary conditions and optimization criteria) in abstract quantifiable descriptions that will allow the algorithms from WP4 to search for most appropriate implementation, based on the models of the ML methods (WP3) and associated hardware (WP2).

WP2 *Hardware Architectures for Low Power ML* - Leader: RPTU. This work package will explore and refine hardware architectures for machine learning accelerations with the focus on energy efficient implementations. Results and inputs from WP1 will be used to search for the most appropriate implementation, while considering the models of the ML methods (WP3) as well. The overall integration will be done in WP5 together with EPROS.

WP3 *Energy Consumption Optimized ML Toolkit and Methods* - Leader: KU. The aim of this work package is to develop a toolkit of ML methods and optimization approaches that comprehensively considers the resource footprint of different components in the ML pipeline. The toolkit will also include ML cores to handle specific tasks for knowledge recycling.

WP4 *Interaction and User Studies* - Leader: INRIA. The goal of this work package is to create new interaction and visualization approaches that will allow users to interactively explore the trade-offs of competing ML models together with intelligent agents. Exploring ML model alternatives during the development process, before the models enter the training cycles, requires users to express potentially ambiguous project objectives and to understand the trade-offs of ML model alternatives, e.g. time, computing hardware, or estimated CO2 footprint for a particular task.

WP5 *Framework Architecture and Implementation* - Leader: EPROSIMA. This Work Package will implement the overall algorithm-hardware co-designed framework in this work package. This work package connects all WP1-WP4 together and provides the engine directly underneath the design interface.

WP6 *Application Oriented Validation* - Leader: IBM. The aim of this work package is to demonstrate in a quantitative way the improvement of the innovative technology developed in the project. Specifically, we aim to validate the framework developed in WP5 in the context of real-world use cases in two application domains: visual inspection and structure health monitoring in the civil engineering domain, and efficient model-transfer for classification in natural language processing.

WP7 *Dissemination, Exploitation and Management* - Leader: EPROSIMA. The main objectives of this package are listed below:

1. To ensure smooth project execution and community building, dissemination and communication.
2. To monitor project progress and achievements as well as risks and contingencies.
3. To maintain Grant Agreement and Consortium Agreement.
4. To promote communication and information exchange among consortium participants and to the target groups.
5. To produce timely reporting to the Commission.
6. To effectively manage all administrative and financial aspects.

2.2 Purpose of this document

The current report focuses on the software design of the *SustainML Design Framework*. Several partners are contributing to the project's mainline code across the different *Work Packages*. Figure 1 depicts the interactions between partners in their related modules among the software-related *Work Packages*.

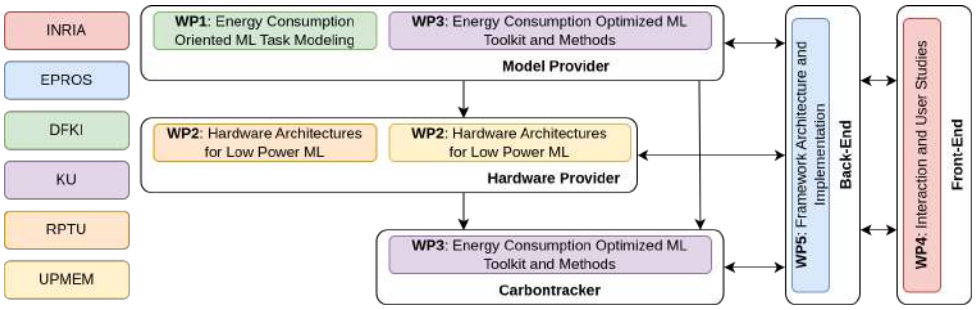

Figure 1: Software based on WP cooperation modules

The *SustainML Design Framework* is composed of different *Software Modules*, each one related to one or more WPs, which are specialized in solving the different parts of the ML problem architecture definition, starting from the user's problem description. These steps are basically:

1. Record user inputs
2. Task encoding
3. Generate the Machine Learning model
4. Select an Optimized Hardware for running the proposed model
5. Predict the Carbon footprint
6. Present the results to the user and iterate if required

The main data flows, expected inputs and outputs for each package are showed in figure Figure 2.

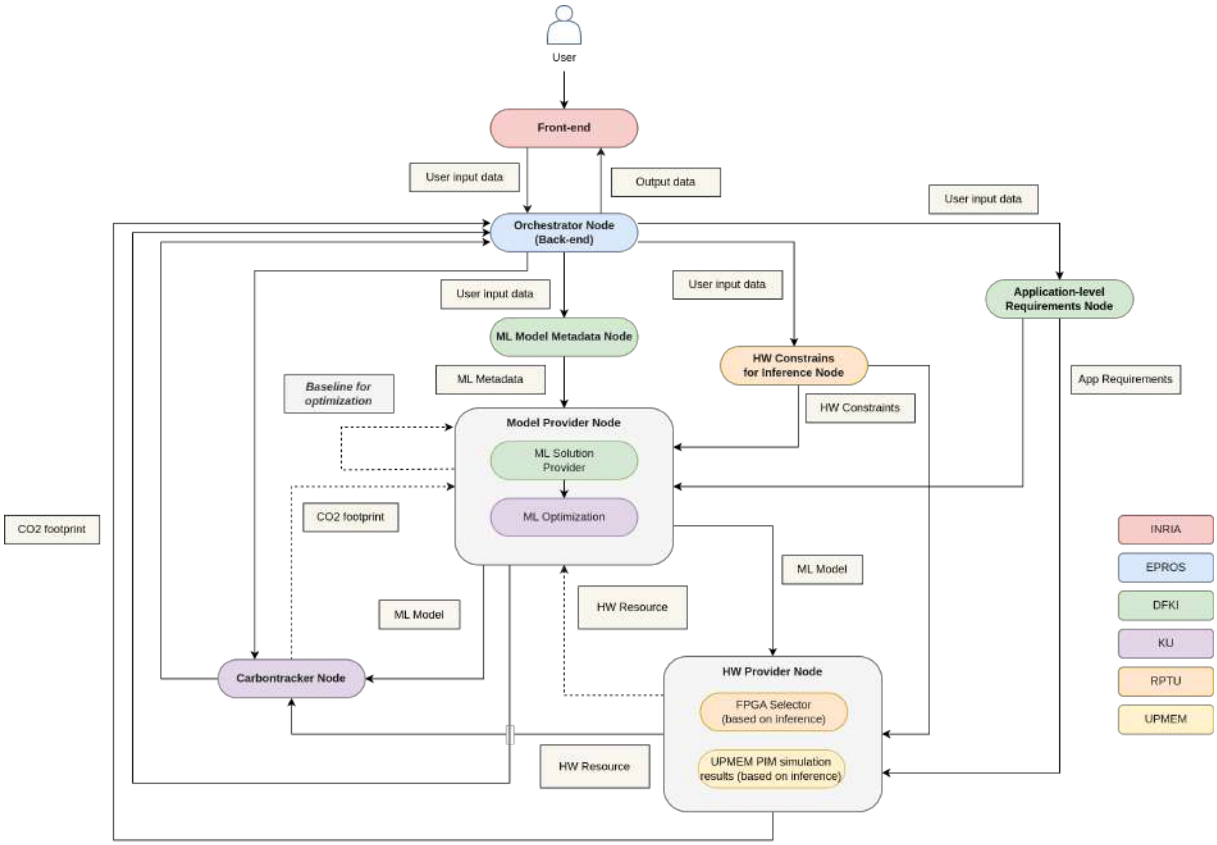

Figure 2: SustainML Pipeline Overview

2.2.1 Purpose of this document (M12 - Deprecated)

The current report focuses on the software design of the *SustainML Design Framework*. Several partners will be contributing to the project's mainline code across the different *Work Packages*. Figure Figure 3 depicts the interactions among the software-related *Work Packages*.

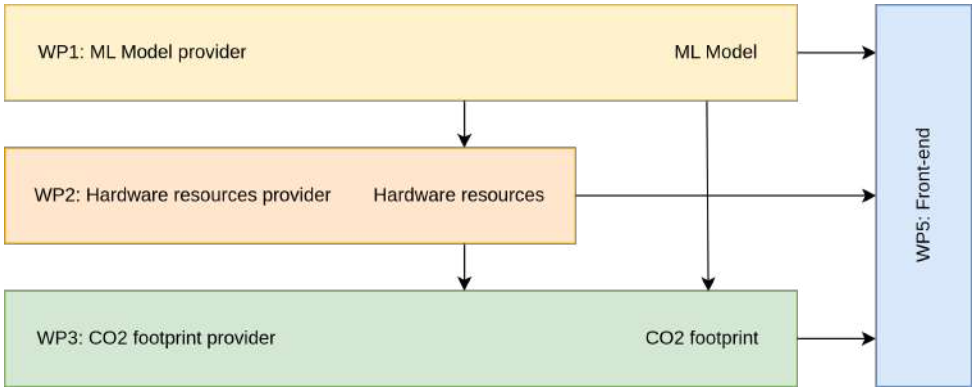

Figure 3: Software based WP interactions (M12 - Deprecated)

The *SustainML Design Framework* is composed of different *Software Modules*, each one related to a WP, which are specialised in solving the different parts of the ML problem architecture definition, starting from the user's problem description. These steps are basically:

1. Task encoding
2. Generate the Machine Learning model
3. Select an Optimized Hardware for running the proposed model
4. Predict the Carbon footprint
5. Present the results to the user and iterate if required

The main data flows, expected inputs and outputs for each package are showed in figure Figure 4.

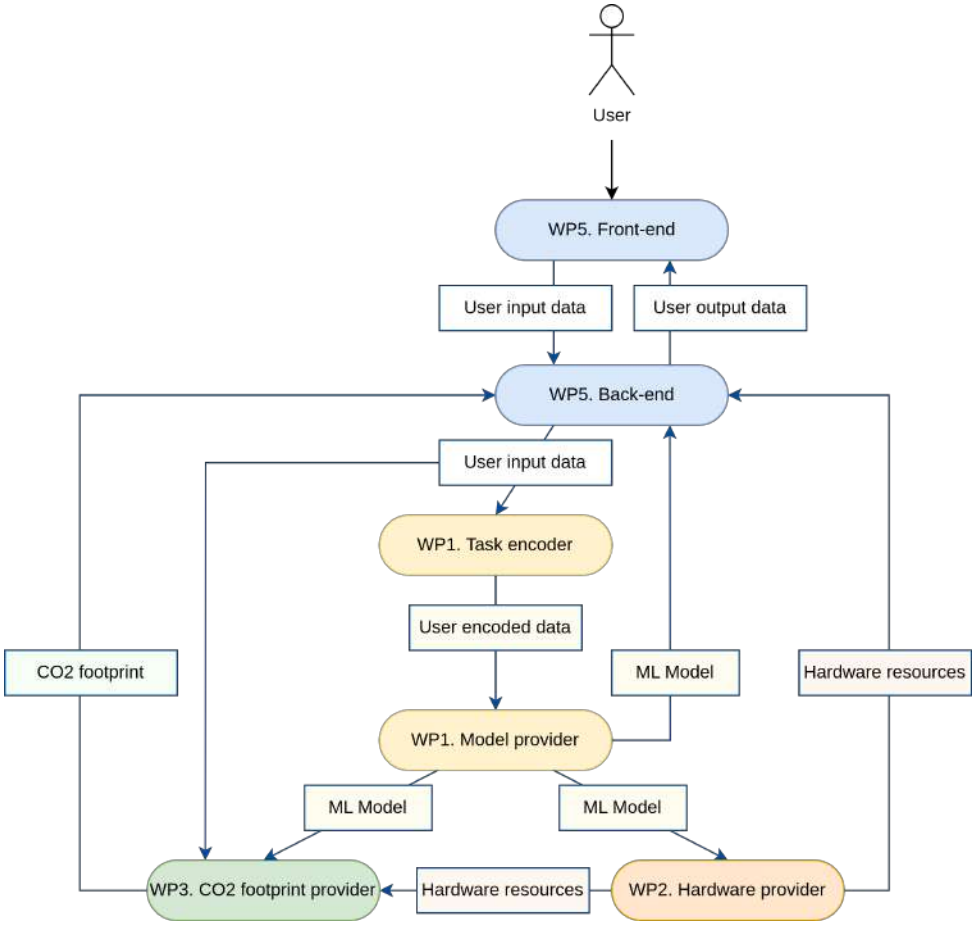

Figure 4: SustainML Pipeline Overview (M12 - Deprecated)

2.3 Structure of this document

The present document is divided into the following sections:

Background context This section discusses the context and technologies to be used in the project and presents the general and WP-specific software requirements for the *SustainML Design Framework*.

Framework architecture This section describes in depth the main components, data model and API for the software modules (developed by the different partners) that make up the *SustainML Design Framework*.

The **Software design** section details the software diagrams and proper design of the main components that comprise the framework.

Release procedure This sections relates how the software is going to be released, the versioning system and routines to be performed prior to a new software release.

The **Results** section presents several proof of concepts and demonstrators that represent the development stage of the project.

Future work This section analyses the current development stage and proposes the next steps to be addressed.

3 Background context

3.1 Enabling technologies

This section details the different technologies that support the development of the *SustainML Design Framework*. All of them focus on the communication between software modules, the protocols used to support such communication and the tools used to handle the different types of data to be transmitted.

3.1.1 Data Distribution Service (DDS)

Given the distributed nature of the SustainML ecosystem, the use of Data Distribution Service (DDS) is proposed to enable real-time communication between the different SustainML software modules (nodes). As explained in the eProsima Fast DDS documentation [1]:

> *The Data Distribution Service (DDS) is a data-centric communication protocol used for distributed software application communications. It describes the communications Application Programming Interfaces (APIs) and Communication Semantics that enable communication between data providers and data consumers.*
>
> *Since it is a Data-Centric Publish-Subscribe (DCPS) model, three key application entities are defined in its implementation: publication entities, which define the information-generating objects and their properties; subscription entities, which define the information-consuming objects and their properties; and configuration entities that define the types of information that are transmitted as topics, and create the publisher and subscriber with its Quality of Service (Quality of Service (QoS)) properties, ensuring the correct performance of the above entities.*
>
> *DDS uses QoS to define the behavioral characteristics of DDS Entities. QoS are comprised of individual QoS policies (objects of type deriving from QoSPolicy).*

The different entities involved in the DDS protocol are also defined in [1].

> *In the DCPS model, four basic elements are defined for the development of a system of communicating applications.*
>
> - *Publisher. It is the DCPS entity in charge of the creation and configuration of the DataWriters it implements. The DataWriter is the entity in charge of the actual publication of the messages. Each one will have an assigned Topic under which the messages are published.*
> - *Subscriber. It is the DCPS Entity in charge of receiving the data published under the topics to which it subscribes. It serves one or more DataReader objects, which are responsible for communicating the availability of new data to the application.*
> - *Topic. It is the entity that binds publications and subscriptions. It is unique within a DDS domain. Through the TopicDescription, it allows the uniformity of data types of publications and subscriptions.*
> - *Domain. This is the concept used to link all publishers and subscribers, belonging to one or more applications, which exchange data under different topics. These individual applications that participate in a domain are called DomainParticipant. The DDS Domain is identified by a domain ID. The DomainParticipant defines the domain ID to specify the DDS domain to which it belongs. Two DomainParticipants with different IDs are not aware of each other's presence in the network. Hence, several communication channels can be created. This is applied in scenarios where several DDS applications are*

> *involved, with their respective DomainParticipants communicating with each other, but these applications must not interfere. The DomainParticipant acts as a container for other DCPS Entities, acts as a factory for Publisher, Subscriber and Topic Entities, and provides administrative services in the domain. See Domain for further details.*

The dynamic addition of new entities to the network is also covered by the the DDS protocol, allowing the dynamic discovery of new instantiated DDS entities. The Discovery Protocol is the mechanism that DDS provides for the discovery of new entities.

3.1.2 eProsima Fast DDS

As described in the *Fast DDS* documentation [2]:

> *eProsima Fast DDS is a C++ implementation of the DDS (Data Distribution Service) Specification, a protocol defined by the Object Management Group (OMG). The eProsima Fast DDS library provides both an Application Programming Interface (API) and a communication protocol that deploy a Data-Centric Publisher-Subscriber (DCPS) model, with the purpose of establishing efficient and reliable information distribution among Real-Time Systems. eProsima Fast DDS is predictable, scalable, flexible, and efficient in resource handling. For meeting these requirements, it makes use of typed interfaces and hinges on a many-to-many distributed network paradigm that neatly allows separation of the publisher and subscriber sides of the communication.*

Among the many features of *Fast DDS*, there are two key functionalities for the development of the *SustainML Design Framework*: *a*) the wide variety of Transports, and *b*) the highly configurable Quality Of Service policies.

Fast DDS provides multiple transport protocols and communication mechanisms, ranging from zero-copy mechanisms to the TCP transport protocol targetting data transmission between DDS entities in WAN networks.

The purpose of the *SustainML Design Framework* is to implement a decoupled and distributed AI design framework able to adapt to local or distributed scenarios optimally.

Moreover, according to [3]:

> *The transport layer provides communication services between DDS entities, being responsible of actually sending and receiving messages over a physical transport. The DDS layer uses this service for both user data and discovery traffic communication. However, the DDS layer itself is transport-independent, it defines a transport API and can run over any transport plugin that implements this API. This way, it is not restricted to a specific transport, and applications can choose the one that best suits their requirements, or create their own.*
>
> *eProsima Fast DDS comes with five transports already implemented:*
>
> - *UDPv4: UDP Datagram communication over IPv4. This transport is created by default on a new DomainParticipant if no specific transport configuration is given.*
> - *UDPv6: UDP Datagram communication over IPv6.*
> - *TCPv4: TCP communication over IPv4.*
> - *TCPv6: TCP communication over IPv6.*
> - *SHM: Shared memory communication among entities running on the same host. This transport is created by default on a new DomainParticipant if no specific transport configuration is given (see Shared Memory Transport).*

The *SustainML Design Framework* will exploit the Quality Of Service parametrization described in [4] as follows:

> *QoS (Quality of Service) is a general concept that is used to specify the behavior of a service. Programming service behavior by means of QoS settings offers the advantage that the application developer only indicates 'what' is wanted rather than 'how' this QoS should be achieved. Generally speaking, QoS is comprised of several QoS policies. Each QoS policy is then an independent description that associates a name with a value. Describing QoS by means of a list of independent QoS policies gives rise to more flexibility.*

The *Fast DDS* Quality of Service policies to accommodate the middleware to the different application scenarios, fit particular requirements or restrictions in hardware or network, for instance.

There are some key Quality of Service policies that the *SustainML Design Framework* will take advantage of to provide distributed reliable communications and optimum resource utilization: *DurabilityQoSPolicy*, *ReliabilityQoSPolicy* and *HistoryQoSPolicy*. The following can be gleaned from the *Fast DDS* online documentation [5]:

- *DurabilityQosPolicy: A DataWriter can send messages throughout a Topic even if there are no DataReaders on the network. Moreover, a DataReader that joins the Topic after some data has been written could be interested in accessing that information. There are four possible kind values:*
  - *VOLATILE_DURABILITY_QOS: Past samples are ignored and a joining DataReader receives samples generated after the moment it matches.*
  - *TRANSIENT_LOCAL_DURABILITY_QOS: When a new DataReader joins, its History is filled with past samples.*
  - *TRANSIENT_DURABILITY_QOS: When a new DataReader joins, its History is filled with past samples, which are stored on persistent storage (see Persistence Service).*
  - *PERSISTENT_DURABILITY_QOS: (Not Implemented): All the samples are stored on a permanent storage, so that they can outlive a system session.*

  [...]
- *ReliabilityQosPolicy: This QoS Policy indicates the level of reliability offered and requested by the service. There are two configurable fields: kind and max_blocking_time.*
  - *BEST_EFFORT_RELIABILITY_QOS: It indicates that it is acceptable not to retransmit the missing samples, so the messages are sent without waiting for an arrival confirmation. Presumably, new values for the samples are generated often enough that it is not necessary to re-send any sample. However, the data samples sent by the same DataWriter will be stored in the DataReader history in the same order they occur. In other words, even if the DataReader misses some data samples, an older value will never overwrite a newer value.*
  - *RELIABLE_RELIABILITY_QOS: It indicates that the service will attempt to deliver all samples of the DataWriter's history expecting an arrival confirmation from the DataReader. The data samples sent by the same DataWriter cannot be made available to the DataReader if there are previous samples that have not been received yet. The service will retransmit the lost data samples in order to reconstruct a correct snapshot of the DataWriter history before it is accessible by the DataReader.*

This max_blocking_time option may block the write operation, hence the max_blocking_time is set that will unblock it once the time expires. But if the max_blocking_time expires before the data is sent, the write operation will return an error. To maintain the compatibility between ReliabilityQosPolicy in DataReaders and DataWriters, the DataWriter kind must be higher or equal to the DataReader kind. And the order between the different kinds is:

$$|BEST_EFFORT_RELIABILITY| < |RELIABLE_RELIABILITY|$$

[...]

- *HistoryQosPolicy This QoS Policy controls the behavior of the system when the value of an instance changes one or more times before it can be successfully communicated to the existing DataReader entities. List of QoS Policy data members:*
  - *kind: Controls if the service should deliver only the most recent values, all the intermediate values or do something in between. See HistoryQosPolicyKind for further details. There are two possible values:*
    - *KEEP_LAST_HISTORY_QOS: The service will only attempt to keep the most recent values of the instance and discard the older ones. The maximum number of samples to keep and deliver is defined by the depth of the HistoryQosPolicy, which needs to be consistent with the ResourceLimitsQosPolicy settings. If the limit defined by depth is reached, the system will discard the oldest sample to make room for a new one.*
    - *KEEP_ALL_HISTORY_QOS: The service will attempt to keep all the values of the instance until it can be delivered to all the existing Subscribers. If this option is selected, the depth will not have any effect, so the history is only limited by the values set in ResourceLimitsQosPolicy. If the limit is reached, the behavior of the system depends on the ReliabilityQosPolicy, if its kind is BEST_EFFORT the older values will be discarded, but if it is RELIABLE the service blocks the DataWriter until the old values are delivered to all existing Subscribers.*
  - *depth: Establishes the maximum number of samples that must be kept on the history. It only has an effect if the kind is set to KEEP_LAST_HISTORY_QOS and it needs to be consistent with the ResourceLimitsQosPolicy, which means that its value must be lower or equal to max_samples_per_instance.*

(...)

3.1.3 Python Bindings

eProsima will deliver a communication library that enables the SustainML abstract all the communications SustainML nodes from within a Python environment by binding the C++ library.

This is a very useful feature for the SustainML project, as most of the code produced for *SustainML Design Framework* is written in Python, and Python will most likely be the chosen language for the main framework with which the end users will interact.

3.2 Software requirements

The *SustainML Design Framework* architecture needs to integrate all the modules developed in each WP, offering a way of connecting them in a distributed approach and presenting the data to the user so that the ML problem statement described by the user can be correctly characterized.

To provide a clear and comprehensive description of what the software is intended to achieve, outlining its functionality, features, and constraints, the following requirements and agreements have been set:

3.2.1 General requirements for all the software modules

Following are the list of general requirements for the *SustainML Design Framework*:

SUSML-FRMW-REQ:01 Ubiquitous language the output from each WP shall be delivered as a Python executable module.

SUSML-FRMW-REQ:02 Distributed Communications the developed software modules shall communicate over the DDS protocol.

SUSML-FRMW-REQ:03 API Compliance the developed Python modules shall comply with the interfaces (API's) defined in the corresponding specific requirements.

SUSML-FRMW-REQ:04 Application Lyfecycle The developed Python modules shall accept a *Status* input so that the execution status can be returned as feedback.

The WP specific requirements are depicted in the following sub-sections:

3.2.2 WP1 requirements

The Work Package 1 is in charge of interpreting the user ML problem definition, giving the optimum, energy-optimized ML model for the user-described problem definition.

WP1-ALRN-REQ:01 : The WP1 shall define a Application-level Requirements Node (ALRN).

WP1-ALRN-REQ:02 : The ALRN shall accept as input the resulting raw data from the ML application description given by the user.

WP1-ALRN-REQ:03 : The ALRN shall output a sequence of *application-level requirements* to be considered in the selection of the ML model.

WP1-MLMMN-REQ:01 : The WP1 shall define a ML Model Metadata Node (MLMMN).

WP1-MLMMN-REQ:02 : The MLMMN shall accept as input the resulting raw data from the ML application description given by the user.

WP1-MLMMN-REQ:03 : The MLMMN shall output a sequence of *keywords*

WP1-MLMMN-REQ:04 : The MLMMN shall output a sequence of *ML model metadata*.

WP1-MPN-REQ:01 : The WP1 shall define a Model Provider Node (MPN).

WP1-MPN-REQ:02 : The MPN shall accept as input the resulting raw data from the ML application description given by the user, as well as the ALRN output and the MLMMN output.

WP1-MPN-REQ:03 : The MPN shall output a path to the file containing the machine learning model.

WP1-MPN-REQ:04 : The machine learning model provided by the MPN shall be in the *ONNX* format.

WP1-MPN-REQ:05 : The MPN shall output a path to the file containing the machine learning model properties.

WP1-MPN-REQ:06 : The machine learning model properties file provided by the MPN shall be in the *json* format.

WP1-MPN-REQ:07 : The MPN shall output a sequence of *input batch* as matrix of Batch x Channels x Height x Width (each one representing a batch).

WP1-MPN-REQ:08 : The MPN shall output a target latency or fps for computer vision tasks or target processing latency in seconds for other tasks, like time-series analysis.

3.2.3 WP2 requirements

WP2-HWCFIN-REQ:01 : The WP2 shall define a Hardware Constraints for Inference Node (HWCFIN).

WP2-HWCFIN-REQ:02 : The HWCFIN shall accept as input the resulting raw data from the ML application description given by the user.

WP2-HWCFIN-REQ:03 : The HWCFIN shall output the *maximum memory carbon footprint* allowed to be considered in the selection of the hardware.

WP2-HWPN-REQ:01 : The WP2 shall define a Hardware Provider Node (HWPN).

WP2-HWPN-REQ:02 : The HWPN shall accept as an input, the application-level requirements provided by the WP1-ALRN-REQ:03.

WP2-HWPN-REQ:03 : The HWPN shall accept as an input, the hardware constraints for inference provided by the WP2-HWCFIN-REQ:03.

WP2-HWPN-REQ:04 : The HWPN shall accept as an input, the machine learning model provided by the WP1-MPN-REQ:03.

WP2-HWPN-REQ:05 : The HWPN shall accept as an input, the machine learning model properties provided by the WP1-MPN-REQ:05.

WP2-HWPN-REQ:06 : The HWPN shall output an optimum detailed hardware description.

WP2-HWPN-REQ:07 : The HWPN shall provide the power consumption estimate of the proposed hardware.

WP2-HWPN-REQ:08 : The HWPN shall provide the estimation of latency of the given model for the given input batch.

WP2-HWPN-REQ:09 : The HWPN shall provide the maximum memory footprint that can be implemented on the target FPGA.

3.2.4 WP3 requirements

WP3-CN-REQ:01 : The WP3 shall define a Carbontracker Node (CN).

WP3-CN-REQ:02 : The CN shall accept as an input, the geographic location provided by the user. In addition, it can optionally query the local power grid carbon intensity for the region.

WP3-CN-REQ:03 : The CN shall accept as an input, the machine learning model provided by the WP1-MLMMN-REQ:01.

WP3-CN-REQ:04 : The CN shall accept as an input, the hardware architectures from WP2-HWPN-REQ:07 and WP2-HWPN-REQ:08.

WP3-CN-REQ:05 : The CN shall provide the estimated CO2 footprint.

WP3-CN-REQ:06 : The CN shall provide the energy consumption.

WP3-CN-REQ:07 : The CN shall provide the carbon intensity.

3.2.5 WP4 requirements

WP4-FE-REQ:01 : The WP4 shall define a Front-end software for presenting the data and guiding the user through the SustainML interactive-design framework.

WP4-FE-REQ:02 : The Front-end shall support a multi-view layout so that the user can request multiple ML problems simultaneously.

WP4-FE-REQ:03 : The Front-end shall record the modality of the data provided by the user.

WP4-FE-REQ:04 : The Front-end shall record the user problem definition.

WP4-FE-REQ:05 : The Front-end shall record a sequence of batches of data for input and output provided by the user.

WP4-FE-REQ:06 : The Front-end shall record the maximum and minimum number of samples required by the user.

WP4-FE-REQ:07 : The Front-end shall record the geographic location provided by the user.

WP4-FE-REQ:08 : The Front-end shall display the model from WP1-MPN-REQ:03.

WP4-FE-REQ:09 : The Front-end shall display the hardware description from WP2-HWPN-REQ:06.

WP4-FE-REQ:10 : The Front-end shall display the carbon footprint and intensity from WP3-CN-REQ:05 and WP3-CN-REQ:07.

3.2.6 WP5 requirements

WP5-ON-REQ:01 : The WP5 shall develop a Python library communication layer, which, using Fast DDS, shall abstract the communications. This library will be later imported and used by the rest of the modules.

WP5-ON-REQ:02 : The WP5 shall define a Orchestrator Node (ON).

WP5-ON-REQ:03 : The ON shall manage the pipeline of the SustainML Framework.

WP5-ON-REQ:04 : The ON shall collect all the data from the rest of the Work Packages nodes.

WP5-ON-REQ:05 : The ON shall distribute new and existing data through the framework pipelines to achieve iterations with the user.

WP5-ON-REQ:06 : The ON shall implement methods to collect user data from WP4-FE-REQ:03, WP4-FE-REQ:04, WP4-FE-REQ:05, WP4-FE-REQ:06 and WP4-FE-REQ:07.

WP5-ON-REQ:07 : The ON shall expose methods to allow the Front-end to query data results as WP4-FE-REQ:08, WP4-FE-REQ:09 and WP4-FE-REQ:10.

3.3 Software requirements (M12 - Deprecated)

The *SustainML Design Framework* architecture needs to integrate all the modules developed in each WP, offering a way of connecting them in a distributed approach and presenting the data to the user so that the ML Model Provider Node problem statement described by the user can be correctly characterized.

To provide a clear and comprehensive description of what the software is intended to achieve, outlining its functionality, features, and constraints, the following requirements and agreements have been set:

3.3.1 General requirements for all the software modules (M12 - Deprecated)

Following are the list of general requirements for the *SustainML Design Framework*:

SUSML-FRMW-REQ:01 Ubiquitous language the output from each WP shall be delivered as a Python executable module.

SUSML-FRMW-REQ:02 Distributed Communications the developed software modules shall communicate over the DDS protocol.

SUSML-FRMW-REQ:03 API Compliance the developed Python modules shall comply with the interfaces (API's) defined in the corresponding specific requirements.

SUSML-FRMW-REQ:04 Application Lyfecycle The developed Python modules shall accept a *Status* input so that the execution status can be returned as feedback.

The WP specific requirements are depicted in the following sub-sections:

3.3.2 WP1 requirements (M12 - Deprecated)

The Work Package 1 is in charge of interpreting the user ML Model Provider Node (MLN) problem definition, giving the optimum, energy-optimized ML model for the user-described problem definition.

WP1-TASKENC-REQ:01 : The WP1 shall define a Task Encoder Node (TEN).

WP1-TASKENC-REQ:02 : The TEN shall accept as input the resulting raw data from the ML application description given by the user.

WP1-TASKENC-REQ:03 : The TEN shall output a sequence of *keywords*.

WP1-MLMODEL-REQ:01 : The WP1 shall define a MLN.

WP1-MLMODEL-REQ:02 : The MLN shall input a sequence of keywords.

WP1-MLMODEL-REQ:03 : The MLN shall output a path to the file containing the machine learning model.

WP1-MLMODEL-REQ:04 : The machine learning model provided by the MLN shall be in the *ONNX* format.

WP1-MLMODEL-REQ:05 : The MLN shall output a path to the file containing the machine learning model properties.

WP1-MLMODEL-REQ:06 : The machine learning model properties file provided by the MLN shall be in the *json* format.

3.3.3 WP2 requirements (M12 - Deprecated)

WP2-HWRES-REQ:01 : The WP2 shall define a Hardware Resources Node (HWN).

WP2-HWRES-REQ:02 : The HWN shall accept as an input, the machine learning model provided by the WP1-MLMODEL-REQ:01.

WP2-HWRES-REQ:03 : The HWN shall accept as an input, the machine learning model properties provided by the WP1-MLMODEL-REQ:05.

WP2-HWRES-REQ:04 : The HWN shall output an optimum detailed hardware description.

WP2-HWRES-REQ:05 : The HWN shall provide the power consumption estimate of the proposed hardware.

3.3.4 WP3 requirements (M12 - Deprecated)

WP3-CO2FOOTPRT-REQ:01 : The WP3 shall define a Carbon Footprint Node (CFN).

WP3-CO2FOOTPRT-REQ:02 : The CFN shall accept as an input, the machine learning model provided by the WP1-MLMODEL-REQ:01.

WP3-CO2FOOTPRT-REQ:03 : The CFN shall accept as an input, the hardware architectures from WP2-HWRES-REQ:04 and WP2-HWRES-REQ:05.

WP3-CO2FOOTPRT-REQ:04 : The CFN shall accept as an input, the geographic location provided by the user. In addition, it can optionally query the local power grid carbon intensity for the region.

3.3.5 WP5 requirements (M12 - Deprecated)

WP5-ARCH-REQ:01 : The WP5 shall develop a Python library communication layer, which, using Fast DDS, shall abstract the communications. This library will be later imported and used by the rest of the modules.

WP5-ARCH-REQ:02 : The WP5 shall develop a backend software that collects all the data from the rest of the Work Packages.

WP5-ARCH-REQ:03 : The WP5 shall develop a Graphical User Interface (GUI) based frontend software for presenting the data and guiding the user through the SustainML interactive-design framework.

WP5-ARCH-REQ:04 : The GUI application shall support a multi-view layout so that the user can request multiple Machine Learning problems simultaneously.

4 Framework architecture

This section describes the main components that make up the *SustainML Design Framework* and presents the data model for the different types of messages that are exchanged among the software modules. Finally, the API to be used by the other partners to develop their software modules is detailed.

The entry point of the *SustainML Design Framework* architecture is a multi-view GUI web application (WP4-FE-REQ:01) with which the user interacts by requesting the energy-optimized solution for new ML problems (tasks).[1]

The Back-End is the collection of modules in charge of gathering all the required data for the *Front-End* to manage it in a decoupled approach, and complying with the SUSML-FRMW-REQ:01 Ubiquitous language, an orchestrator node exchanges information over DDS with each of the developed Python modules. Each of the developed Python modules conforms a *Node*.

The resultant architecture overview is shown in the figure below:

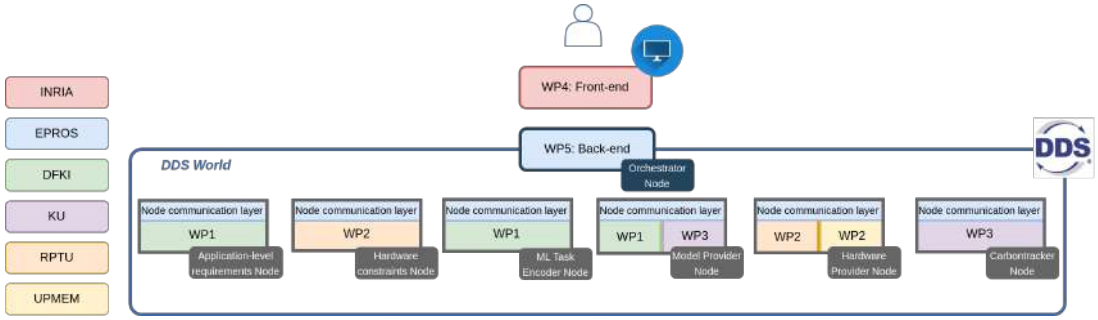

Figure 5: SustainML Framework DDS node architecture

All the modules shall import their corresponding Python library developed by EPROS (WP5-ON-REQ:01), so that each module can be abstracted from the DDS communications [2].

In some cases, the module requires cooperation work between different partners, even from different WP.

4.1 API for the Software Modules

To comply with the SUSML-FRMW-REQ:01 Ubiquitous language for each Work Package, a common API and operation principle is defined for the Python communication library.

For each module, the first step would be importing the corresponding node and data types. As stated in the Framework architecture, each of the working units of the Framework is named as a *Node*. Consequently, each module shall instantiate its *Node*.

Every node will implement its functionality into an appropriate callback, receiving the corresponding inputs in each case, plus the status object and its output data structure. In order for the node to be

[1] This visual GUI is mainly developed in cooperation with INRIA within the Front-End. Each of the views (independent Machine Learning problem) comprising the layout is uniquely identified by a *task_id*.

[2] In this initial stage, the ML problem description will be provided as a form. This form shall be filled by the user and could include sections like "Provide the Problem description", "At which location is the algorithm going to be run?".

able to invoke the callback when the required inputs are received for a particular task_id, it is needed to register it in the node. That callback can use different Python modules and methods developed by the partners to achieve the expected *Node* behavior. During the callback execution, it is advised to perform the corresponding calls to the *update()* method of the *NodeStatus*[3] object to track the status of the order identified by the task_id, for monitoring and debugging purposes.

Before finishing the callback, if there were no errors and the execution was fine, the corresponding output data structure shall be filled.

Finally, each module should call the *spin()* method of the *Node* so that the node can start running.

All the template script examples can be found in the API usage examples for the software modules appendix.

4.2 Framework architecture (M12 - Deprecated)

This section describes the main components that make up the *SustainML Design Framework* and presents the data model for the different types of messages that are exchanged among the software modules. Finally, the API to be used by the other partners to develop their software modules is detailed.

The entrypoint of the *SustainML Design Framework* architecture is a multi-view GUI application (WP5-ARCH-REQ:03) with which the user interacts by requesting the energy-optimized solution for new ML Model Provider Node problems (tasks).

The Back-End is the collection of modules in charge of collecting all the required data for the *Front-End*. To manage it in a decoupled approach, and complying with the SUSML-FRMW-REQ:01 Ubiquitous language, an orchestrator node exchanges information over DDS with each of the developed Python modules. Each of the developed Python modules conforms a *Node*.

The resultant architecture overview is showed in the figure below:

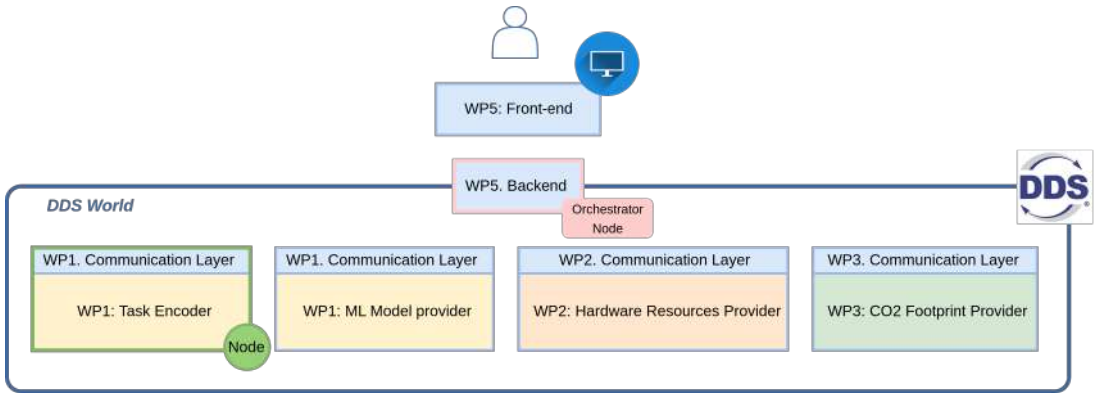

Figure 6: SustainML Framework Architecture (M12 - Deprecated)

[3]The corresponding *enum* names describing the different statuses can be used for the purpose.

5 Software design

In this section, the *SustainML Design Framework* software design is presented and discussed. It is important to note that this does not intend to be the final product design, but a dynamic design that represent the state of the actual project requirements and tackles possible future demands.

5.1 Design guidelines

Software design is a critical phase in the project's development lifecycle that lays the foundation for creating an efficient, maintainable, and robust framework. It encompasses the process of transforming requirements and specifications (see Software requirements (M12 - Deprecated)) into a comprehensive plan that outlines the architecture, components, interfaces, and interactions of the software system.

Effective software design is essential to ensure that the final *SustainML Design Framework* meets user needs, is adaptable to future changes, and exhibits high performance by carefully considering factors such as modularity, scalability, security, and user experience.

Several key aspects underpin successful software design.

- Modularity, the practice of breaking down a complex system into smaller, manageable components, promotes reusability and ease of maintenance.
- Encapsulation emphasizes encapsulating data and behavior within classes or modules, enhancing data integrity and promoting a clear separation of concerns.
- Inheritance and polymorphism enable designers to create hierarchies of classes that facilitate code reuse and extensibility.
- Design patterns, established reusable solutions to common design problems, provide blueprints for addressing specific challenges while ensuring best practices.

Furthermore, iterative and incremental design approaches allow for the continuous refinement of design decisions as the project evolves, accommodating changing requirements and fostering adaptability.

In particular, the *SustainML Design Framework* adheres to the *SOLID Software Design Principles* and makes use of the *Unified Modeling Language* to create and represent the software design.

SOLID design principles

In order to create codebases that are more maintainable, flexible, and resilient to changes. These principles encourage modular design, clear separation of concerns, and a focus on the long-term evolution of the project's requirements.

The SOLID [6] principles consists of five design principles in object-oriented programming in which each letter in the acronym represents a specific principle:

Single Responsibility Principle (SRP) : This principle states that a class should have only one reason to change, meaning it should have a single responsibility or purpose. A class should encapsulate a single piece of functionality or behavior. This promotes modularity, making it easier to understand, maintain, and modify individual components without affecting the entire system.

Open/Closed Principle (OCP) : The Open/Closed Principle suggests that software entities, such as classes, modules, or functions, should be open for extension but closed for modification. In other words, you should be able to add new functionality to a system without altering existing code. This is typically achieved through the use of inheritance, interfaces, and abstract classes, allowing you to extend behavior without changing the existing codebase.

Liskov Substitution Principle (LSP) : The Liskov Substitution Principle emphasizes that objects of a derived class should be able to replace objects of the base class without affecting the correctness of the program. In simpler terms, if a class is a subclass of another class, it should be able to be used interchangeably with its parent class without causing unexpected behavior. This principle helps maintain consistency and predictability in object-oriented systems.

Interface Segregation Principle (ISP) : The Interface Segregation Principle states that clients should not be forced to depend on interfaces they do not use. In other words, a class should not be obligated to implement methods that it doesn't need. This principle encourages the creation of smaller, focused interfaces rather than large, monolithic ones. By doing so, it prevents unnecessary dependencies and minimizes the impact of changes.

Dependency Inversion Principle (DIP) : The Dependency Inversion Principle suggests that high-level modules should not depend on low-level modules; both should depend on abstractions. Furthermore, abstractions should not depend on details; details should depend on abstractions. This principle promotes the use of interfaces or abstract classes to define the interactions between components, enabling easier substitution of implementations and reducing tight coupling between different parts of the system.

Unified Modeling Language

The Unified Modeling Language (UML) [7] is a standardized visual modeling language used in software engineering to describe, visualize, and document the structure and behavior of software systems and processes. UML provides a set of graphical notations and diagrams that allow software developers, analysts, and designers to communicate and capture different aspects of a system's design and functionality. It was developed to address the need for a common language that can bridge the gap between various stakeholders involved in software development.

UML diagrams cover a wide range of perspectives, including structural aspects (like classes, objects, components), behavioral aspects (such as interactions, state machines), and even architectural views. Some common types of UML diagrams include class, use case, sequence, activity, state machine, component or deployment diagrams.

5.2 Back-End

5.2.1 Module Nodes

Within the *SustainML Design Framework*, a node refers to each one of the software modules that perform a particular task, conforms to a single executable unit, and can be locally or remotely deployed, as described in the SustainML Framework Architecture (M12 - Deprecated).

In essence, the node library provides an abstraction layer over the DDS communications and data flow management so that each one of the partners can focus on its corresponding ML task in the project (see Introduction).

Class diagram

The class diagram in Figure 7 depicts the main classes and responsibilities, consistent with SOLID design principles, and includes the CN as a Node example usage:

- The Node class aggregates all the DDS-related entities in order to publish and subscribe to data. It also contains the dispatcher.
- The *Dispatcher* carries information from the number of samples received in the different inputs. Each node expects a sample from all its inputs with the same *task_id* in order to perform its task. It is important to note that the *task_id* must be unique among samples of the same type. As a consequence, it is invalid to receive two samples with the same task_id in the same input queue.
- The *NodeListener* class implements the subscription callbacks. When a new sample is received from any of the inputs, the *NodeListener* stores the sample in its *SamplePool* and notifies the *Dispatcher*. The *SamplePool* is an utility class that uses a fixed block of memory that guarantees contiguity, allowing for a better performance.
- The Node class provides the basic methods and interfaces so that it can be specialized by each of the final node classes. Each specialization must provide the implementation for the *publish_to_user()* method.

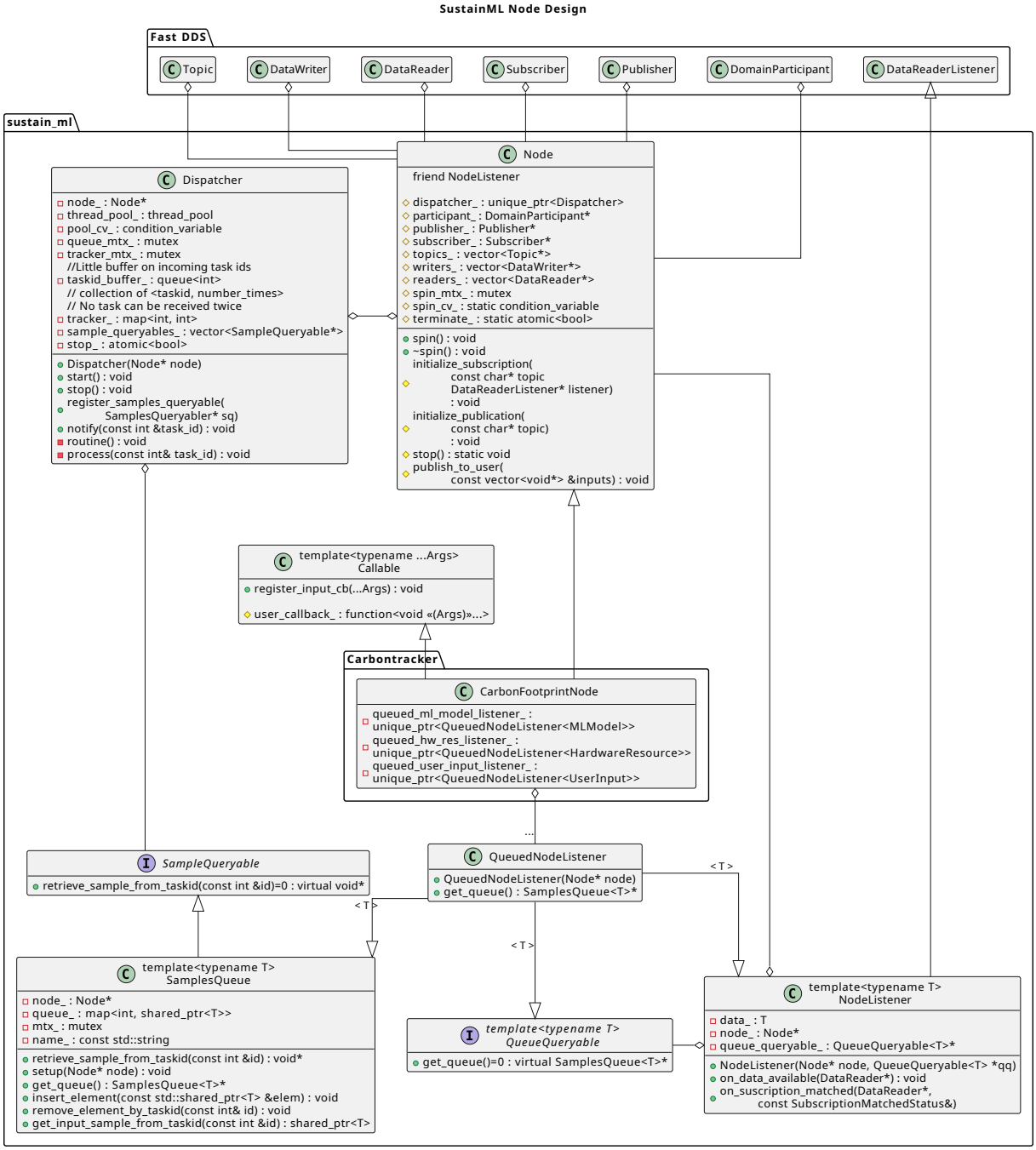

Figure 7: Node UML Design

Flow Diagrams

Following (Figure 8 and Figure 9) are the flow diagrams depicting the initialization of a node and retrieval of a new sample. The different flows represent the interaction among the main entities included in the Node UML Design.

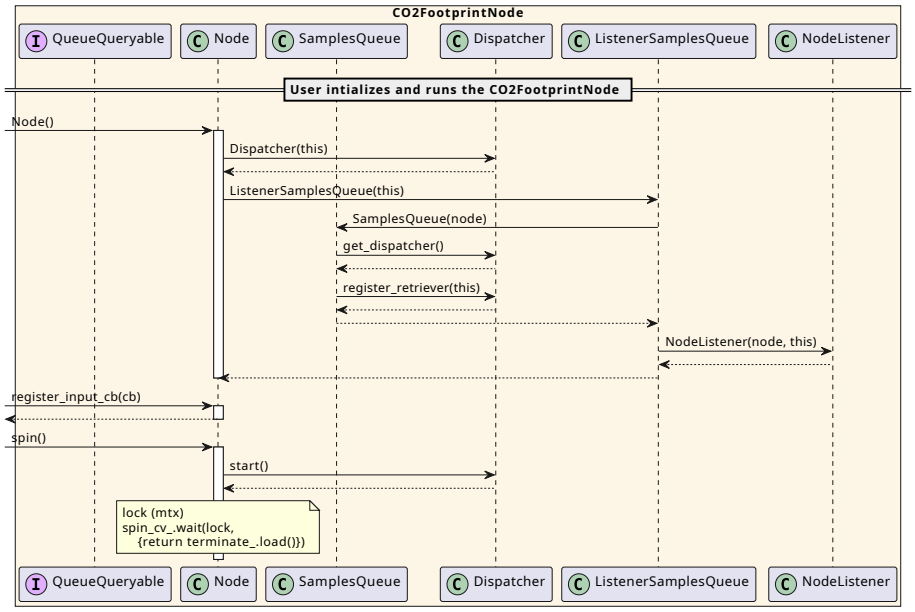

Figure 8: Initialization Flow Chart

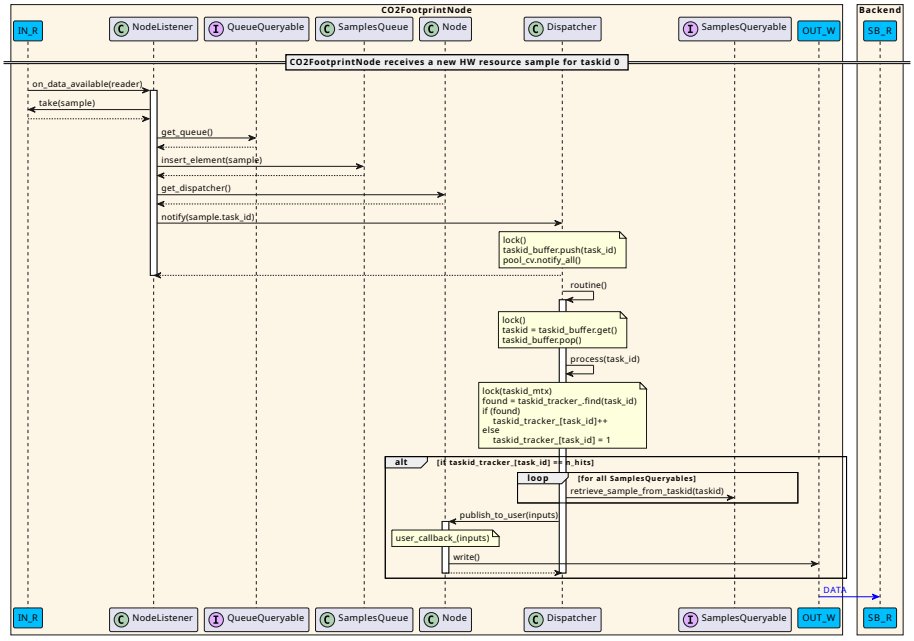

Figure 9: New Sample Flow Chart

5.2.2 Orchestrator Node

The orchestrator node is the entity that manages the orders received from the Front-end (actions derived from the user's interaction with the GUI) to the different nodes, and provides feedback of the status of a particular task to the Front-end to be displayed.

The orchestrator subscribes to each node's intermediate output (DDS topic) to provide traceability of a particular task and track its status. In addition, the orchestrator will act as the lifecycle manager of the different nodes, being capable of starting, pausing, resetting, or shutting down nodes when convenient (see the reserved event names defined in Data Model).

In addition, the orchestrator also manages the *baseline for optimization* mechanism to further iterate in the SustainML Framework.

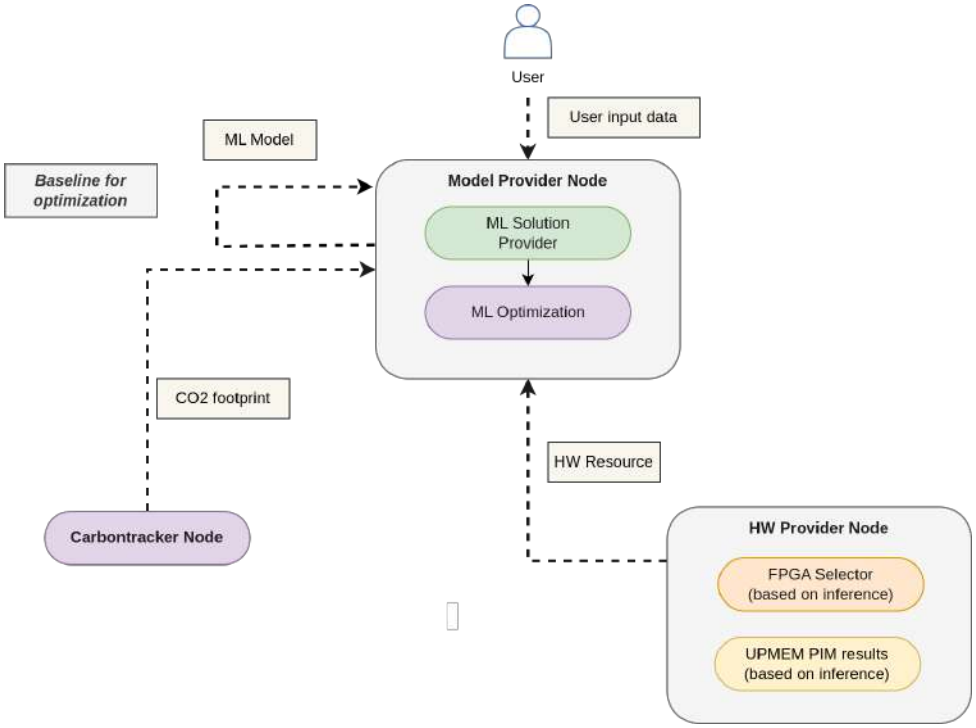

Figure 10: SustainML Framework baseline topics

This mechanism is based on an existing *task_id* associated with a specific user submission. It takes already valid data to repeat the MPN workflow (and its derivatives) of the SustainML Pipeline using *baseline* information. The *baseline* information is taken from the already existing data in a previous iteration for the same *task_id*.

Once the orchestrator has recorded the results from all the nodes, it feds the MPNalong with the *baseline* information. This ends up in an internal pipeline loop that optimizes the results until certain conditions are met (desired carbon footprint achieved or until certain iterations have been performed). This mechanism allows obtaining detailed information under the same *task_id* resulting from different approaches, such as optimization based on the estimation of the carbon footprint generated per different models in different hardware. Therefore, different alternatives can be considered and proposed to the user based on the same given description, displaying a set of possible solutions and their comparison with their advantages and caveats.

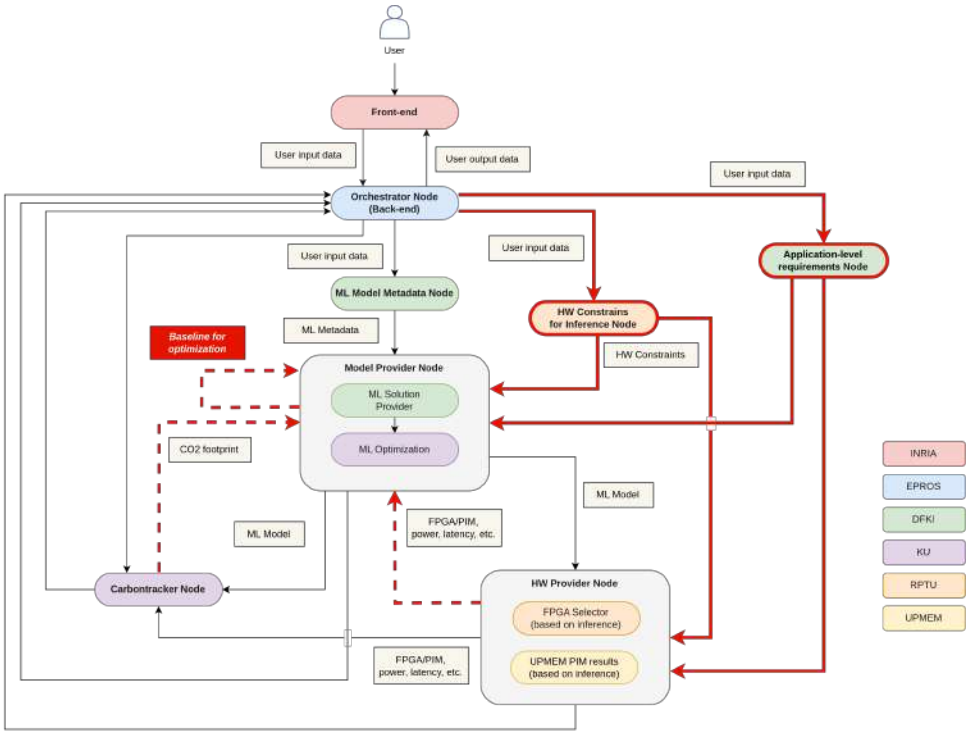

Figure 11: SustainML Framework baseline pipeline

As the project is in its development stage, the orchestrator (more specifically the baseline mechanism) is currently under design stage and development. Hence, this section will be updated in upcoming versions of this document.

5.2.3 Orchestrator Python Module

The Orchestrator Node ON entity was designed and developed as a C++ library, so a C++ application (as the Front-End prototype) could instantiate it and use it to retrieve SustainML Framework information. Working towards the integration of the WP4-FE-REQ:01 INRIA web application Front-End in the SustainML Framework, the ON has been exposed in the SustainML library, migrated to a Python module, and combined with a Python web server. This communication is no longer performed through DDS protocol, but instead, HTTP REST Application Programming Interface (API) is used. In such a way, the WP4-FE-REQ:01 INRIA Front-End is managed to get and post HTTP requests and responses to the ON to provide the required user information to the remaining nodes and display the SustainML Framework solutions and results to the user.

The Hypertext Transfer Protocol (HTTP) is a standardized communication protocol used to transfer data, primarily web content, over the Internet. An HTTP request is a message initiated by a client to a server, requesting specific data or services. The server processes the request and responds with an HTTP response, containing the requested data or a status code indicating the outcome of the request. A REST API follows the design principles of the REST architectural style, where a request made via a RESTful API transfers the corresponding information in one of several formats via HTTP: JSON (Javascript Object Notation), HTML, XLT, Python, PHP, or plain text. JSON is the most generally popular file format to use because, despite its name, it's language-agnostic, as well as readable by both humans and machines.

Flask is a lightweight, flexible Python web framework that provides tools and structure for building web applications. A Flask web server is a server application built using the Flask framework. It listens for incoming HTTP requests from clients (like web browsers) and processes them according to the defined routes and logic within the Flask application. When a request arrives, the Flask server determines the appropriate route based on the URL and HTTP method, then executes the corresponding Python code to generate a response.

The ON python module launches a Flask web server that receives those HTTP requests. That web server Python code embedded in the ON python module retrieves the requested information from the SustainML Framework and builds the HTTP response to be sent to the web application Front-End in JSON format.

5.3 Front-End

The Front-end component within the software architecture of the *SustainML Design Framework*, is the user-facing software component. It's the part of the application that users interact with directly. The Front-end is responsible for displaying information, receiving user input for defining a new task, and providing a means for users to interact with the underlying functionality of the *SustainML Design Framework*.

The Front-end includes all the visual elements conforming the user interface, such as windows, dialogs, buttons, text fields, menus, icons, and other graphical elements. The UI of the *SustainML Design Framework* will be designed to be intuitive and user-friendly to exploit its dissemination strategies and reach all kind of end-users: from novel to experts in AI. Besides, the Front-end will handle user input, such as mouse clicks, keyboard input and event handling, it will process these inputs, and initiate actions or update the display, synchronised with the Back-end.

Creating a positive user experience for the wide range of the target audience in SustainML is a primary goal for the Front-end. This involves considering factors like usability, accessibility, performance, and user satisfaction. The overall design for the UI, UX (User experience) and layout of the Front-end will take into account the results and feedback from the WP4 (subsection 2.1).

6 Release procedure

With the aim of establishing an standardized way of proceeding throughout software releases and versions, the SustainML project sets a series of guidelines to correctly address the evolution and traceability of the developed software.

Beta and alpha releases will be important in early stages of the software development lifecycle, each serving distinct purposes that contribute to the overall quality and success of a software product.

Alpha releases will the first versions of the *SustainML Design Framework* product made available for the community. These versions are not feature-complete and may contain significant bugs and incomplete functionality. They are used primarily for testing. They help identify and rectify critical issues, bugs, and performance bottlenecks before the software reaches a wider audience. Subsequently, it allows developers to catch and address issues at an early stage, reducing the likelihood of critical problems in later stages of development.

Following alpha releases, the SustainML project will make use of beta release versions that will allow for a wider testing (stress, compatibility) with the core functionality developed and will be assessed in different scenarios by the community and AI audience of different levels of expertise.

In addition, prior to each release, a new Release Candidate (RC) and release testing must be generated and addressed. Those, are an essential part of the *SustainML Design Framework* software development release procedure for several reasons:

Quality Assurance: RCs represent a near-final version of the software. They are thoroughly tested to ensure that they are stable and meet the quality standards expected in the final release. This extensive testing helps identify and fix any lingering bugs or issues, ensuring a more reliable product.

Stress Testing: Release candidates are subjected to stress and performance testing to assess how the software performs under heavy loads and in various scenarios. This helps identify potential bottlenecks and scalability issues.

Compatibility Testing: RCs are tested across a range of hardware configurations, operating systems, and software environments to ensure compatibility. This is crucial for ensuring that the software works seamlessly for a broad user base.

Feature Freeze: RCs typically represent a feature freeze, meaning that no new features are added at this stage. This stability allows for a final review of all features and functionality, ensuring they are complete and meet the requirements.

Documentation and Training: RCs are used to finalize documentation, user manuals, and training materials. This ensures that users have access to accurate information when the final release is launched.

Preparation for Deployment: The RC phase provides an opportunity to conduct dry runs of the deployment process, ensuring that all necessary procedures and resources are in place for a smooth release.

Risk Mitigation: Identifying and addressing any issues in the RC stage reduces the risk of critical defects or problems occurring in the final release. This helps prevent post-release hotfixes and patches, which can be disruptive to users.

The SustainML partnership will agree and elaborate the release calendar of the software releases from which to define the freeze and release testing dates for the different versions.

6.1 Versioning

The versioning [8] rationale within *SustainML Design Framework* will consist of three segments: *Major*, *minor*, and *patch* in the format *X.Y.Z*. Each segment having a specific purpose:

- Major Version: This number represents a significant, often backward-incompatible changes to the software. A major version change usually indicates substantial updates, a public API backward-compatibility break or the introduction of major new features.
- Minor Version: The minor version number refers to smaller, backward-compatible feature additions or enhancements. It indicates substantial but non-disruptive changes.
- Patch Version: The patch version number is increased for minor, backward-compatible bug fixes, security updates, or other minor improvements that don't introduce new features.

For instance, if the current version is *1.5.0* and a patch is released to address a few bugs, the new version might be labeled as *1.5.1*. This signifies that it's a backward-compatible update with minor changes aimed at improving the stability and reliability of the software.

Typically, the alpha release versions will correspond to 0.0.X and beta releases will take the range 0.1.X.

7 Results

This section presents several proofs of concepts and demonstrators that represent the development stage of the *SustainML Design Framework* project. This section is meant to be continuously updated.

7.1 Module Nodes demonstrator

The *SustainML Design Framework* is composed of several software modules (Figure 6). One of the core concepts and components of the system are the different *module nodes* that solve a particular task within the project: extracting the key words from the user ml problem definition to conform an encoded task, choosing an optimal machine learning model that fits the problem, selecting an energy-optimized hardware and estimating the carbon footprint.

To achieve that, each of the module nodes integrates a communications library (a.k.a Node library) responsible for all the DDS communications. The current demonstrator exemplifies the distributed communications through DDS.

7.1.1 Description

This is a CLI (Command Line Interface) terminal-based demonstrator in which initially four nodes (MLMMN, MPN, HWPN and CN) have been created and its implementation has been mocked to wait for some time in order to simulate the *processing time*. In addition, there is a first User Input node that acts as an entrypoint i.e representing the input ML problem of the user. All the messages exchanged among the nodes complies with the Data Model.

The diagram in Figure 12 depicts the demonstrator schema according to Figure 4.

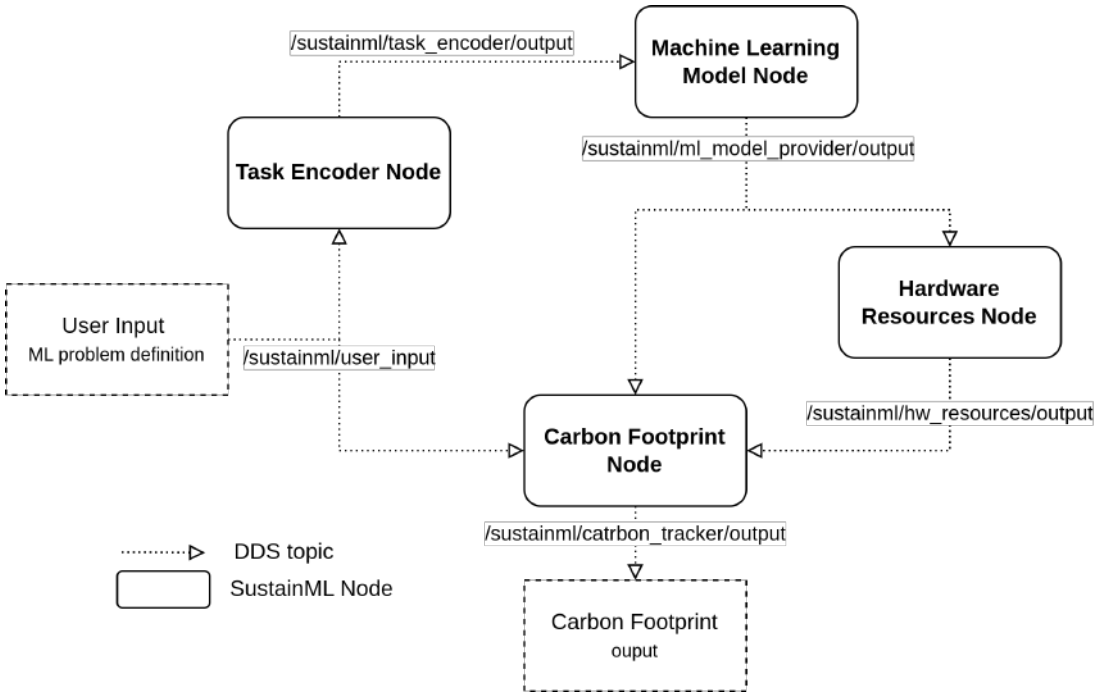

Figure 12: Module Nodes and Topics

The user inputs a new task with the keyboard in the User Input node, which publishes the message to the MLMMN and CN. After processing the message, the MLMMN outputs the encoded data to the MPN that generates a sample MLModel message and publishes it for the HWPN and CN. After the HWPN has finished, it publishes its output to the CN which, having all the three required inputs, processes and generates a mocked CO2 footprint estimation.

7.1.2 Technologies used

In this initial stage of the project, C++ has been used to produce the demonstrator, but the node library will later be provided in Python so that each partner can develop its Python module, complying with the General requirements for all the software modules.

The eProsima Fast DDS library has also been used for the demonstrator.

7.1.3 Experiments

The Module Nodes demonstrator comprises two scenarios: single and multiple processes, that are detailed below:

Single process scenario In this first scenario, all the nodes are executed in the same process, as shown in Figure 13.

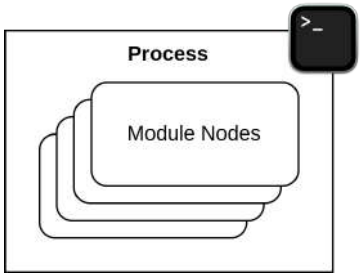

Figure 13: Module Nodes Demonstrator in Single Process scenario

The real capture screenshots can be found in Screenshots of the Module Nodes demonstrator appendix section.

Inter-process scenario In this other scenario, all the nodes are executed within different processes, as shown in Figure 14. In this case, if launched in the same host, the Shared Memory Transport (SHM) is used to communicate data between the different processes, taking advantage of the Fast DDS capabilities.

In particular, this scenario exemplifies the native distributed approach that can be achieved in the project, allowing for a loosely coupled component design and better scalability and re-usability.

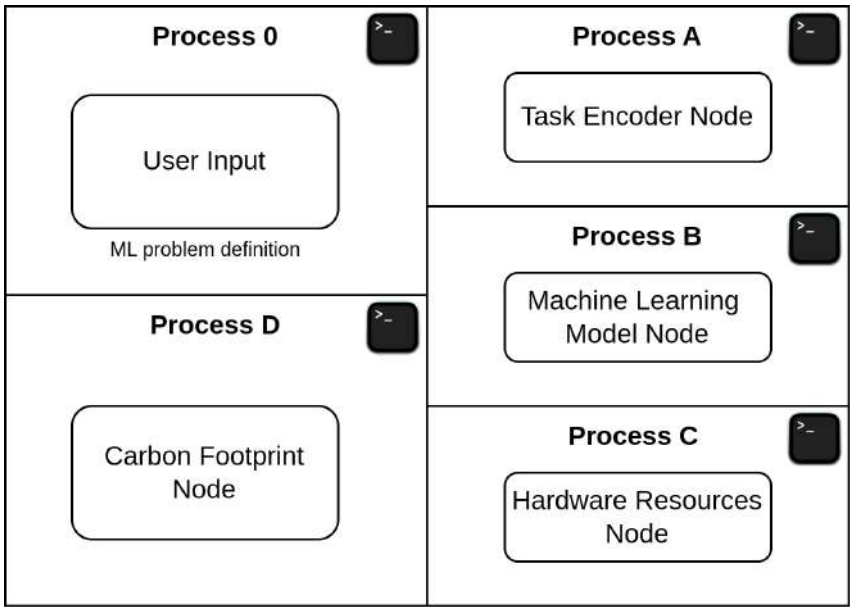

Figure 14: Module Nodes Demonstrator in Inter Process scenario

The corresponding screenshots can be found in Screenshots of the Module Nodes demonstrator.

7.2 SustainML Framework demonstrator

The *SustainML Design Framework* is composed of several software modules (Figure 6). One of the core concepts and components of the system are the different *module nodes* that solve a particular task within the project.

The orchestrator node receives all those outputs and transmits them to the Front-end. The Front-end has been designed to record all the required information for the nodes as a form in a desktop application.

7.2.1 Description

The SustainML Framework has been prepared to be deployed locally using docker. A Dockerfile configures Ubuntu 22.04 with the framework requirements, and a docker compose yaml file opens different docker containers with the Front-end and the desired Module Nodes. In the following figures, taken for a video recorded with the complete process, the MLMMN and the MPN nodes have been run separately integrating the code provided by DFKI, so their system outputs are displayed.

The following image describes the GUI as a form that collects the different Module Nodes required data.

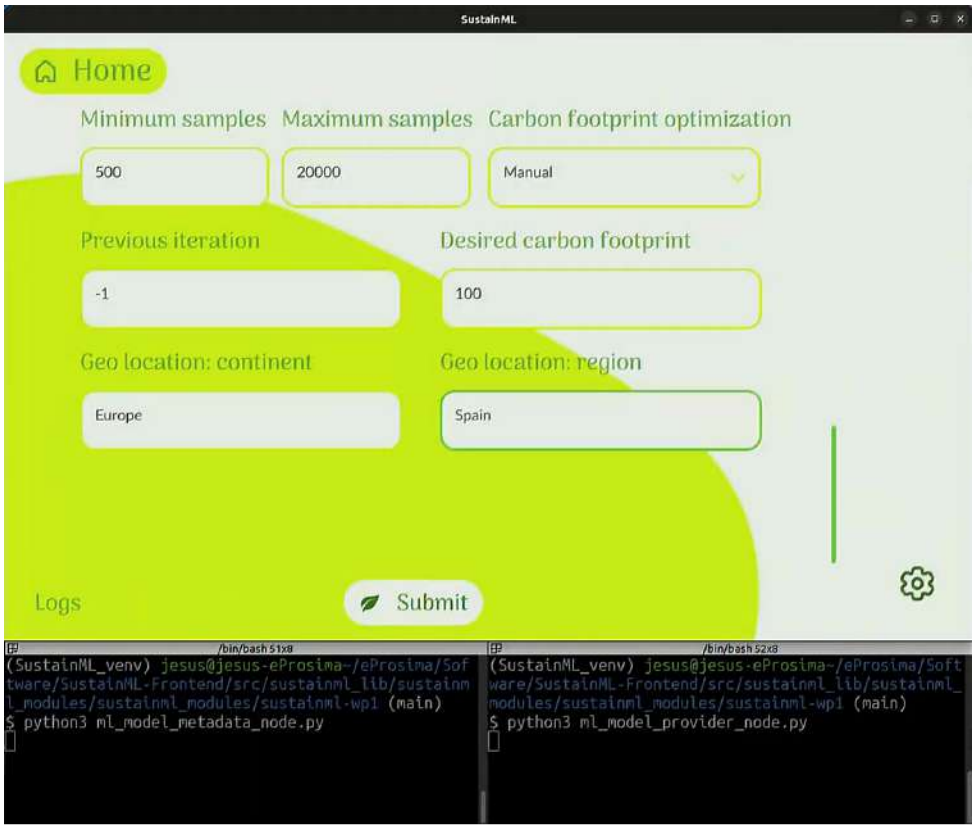

Figure 15: SustainML Framework's demonstrator form

Then the input information is piped to the framework as explained in the Figure 4. Once all the information has been processed, and taking into account that the remaining Nodes have been mocked, this image describes the GUI representing the resulting data.

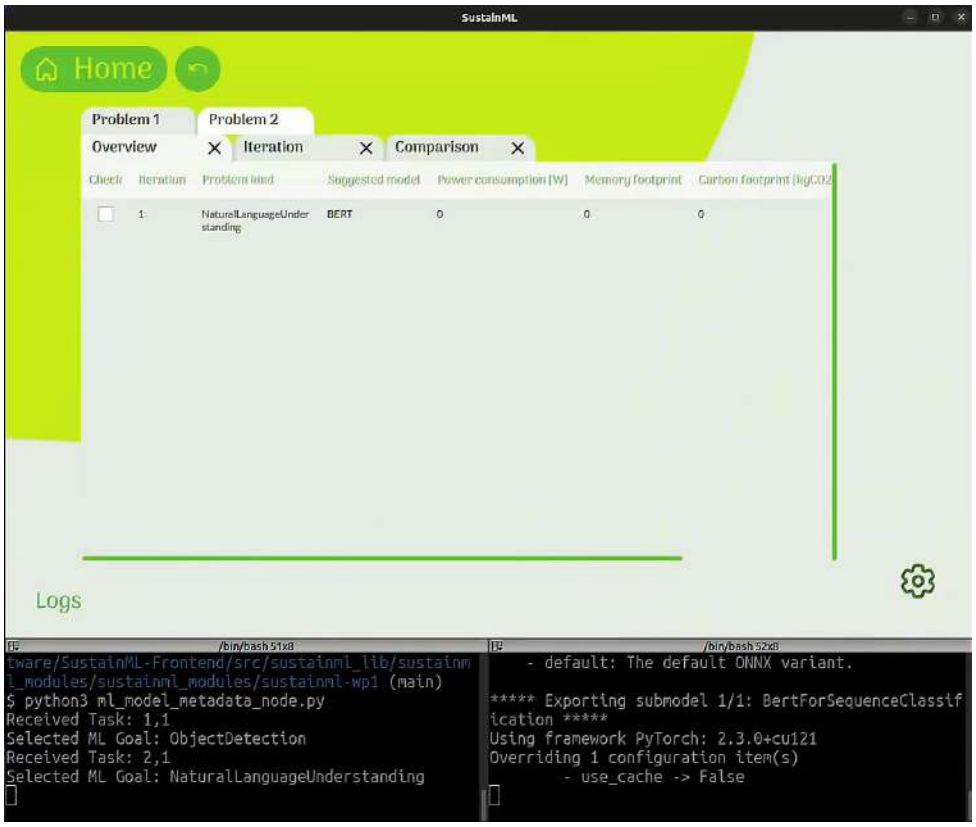

Figure 16: SustainML Framework's demonstrator results

Finally, the traces and node states can be checked in the logs section, which displays each Node status marked as *"idle"* because no tasks are being processed, and the received raw data per Node.

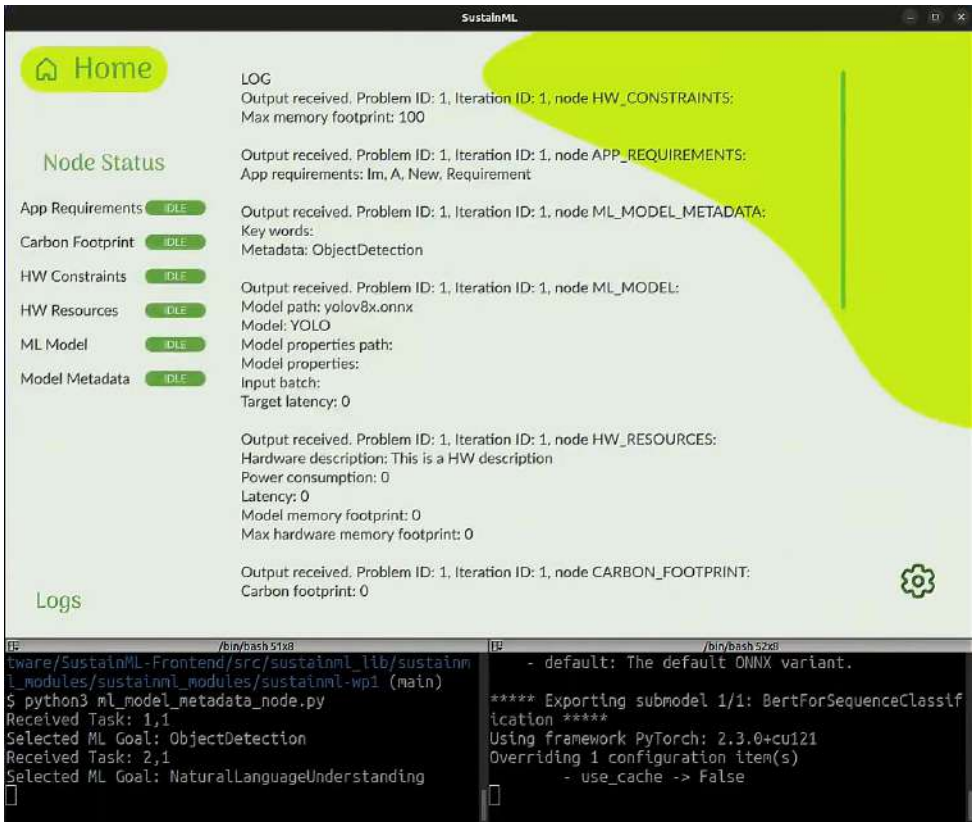

Figure 17: SustainML Framework's log demonstrator

7.3 Orchestrator Python Node

Working towards the integration of the INRIA web application Front-End in the SustainML Framework, the Orchestrator Node has been exposed in the SustainML library, migrated to a Python module, and combined with a Python web server.

The Orchestrator Python Node listens to HTTP requests and builds the corresponding response to be sent to the web application Front-End.

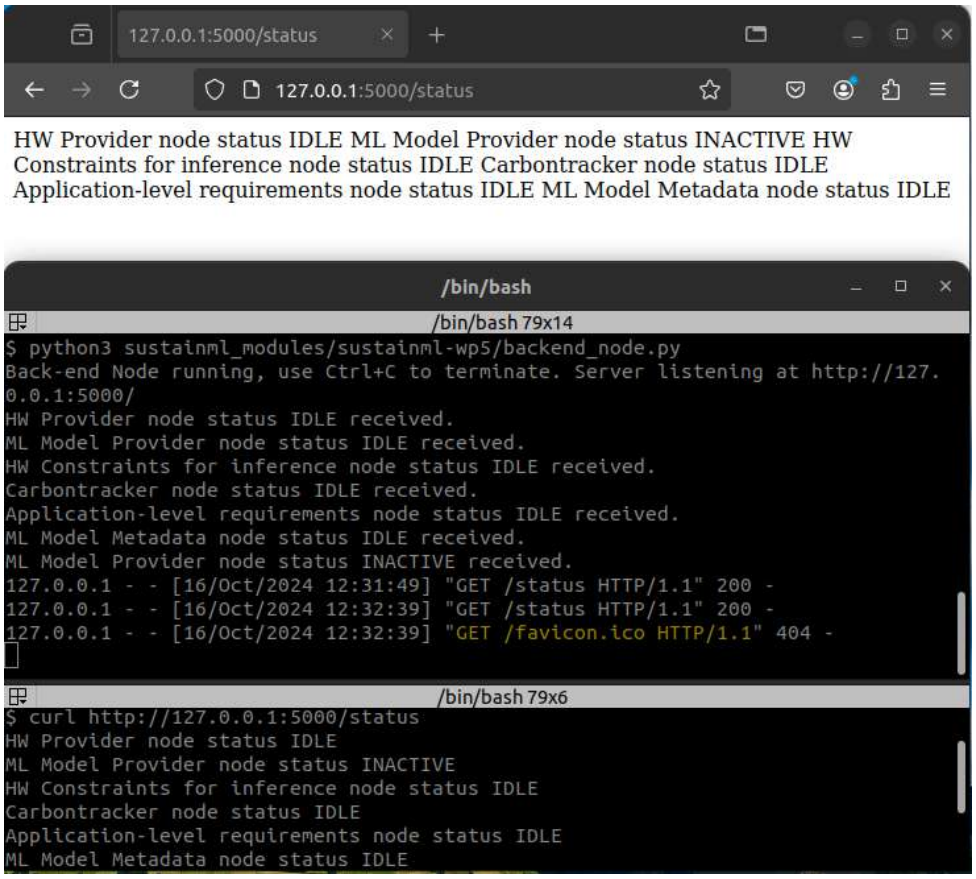

Figure 18: Orchestator Python Node HTTP status request demonstrator

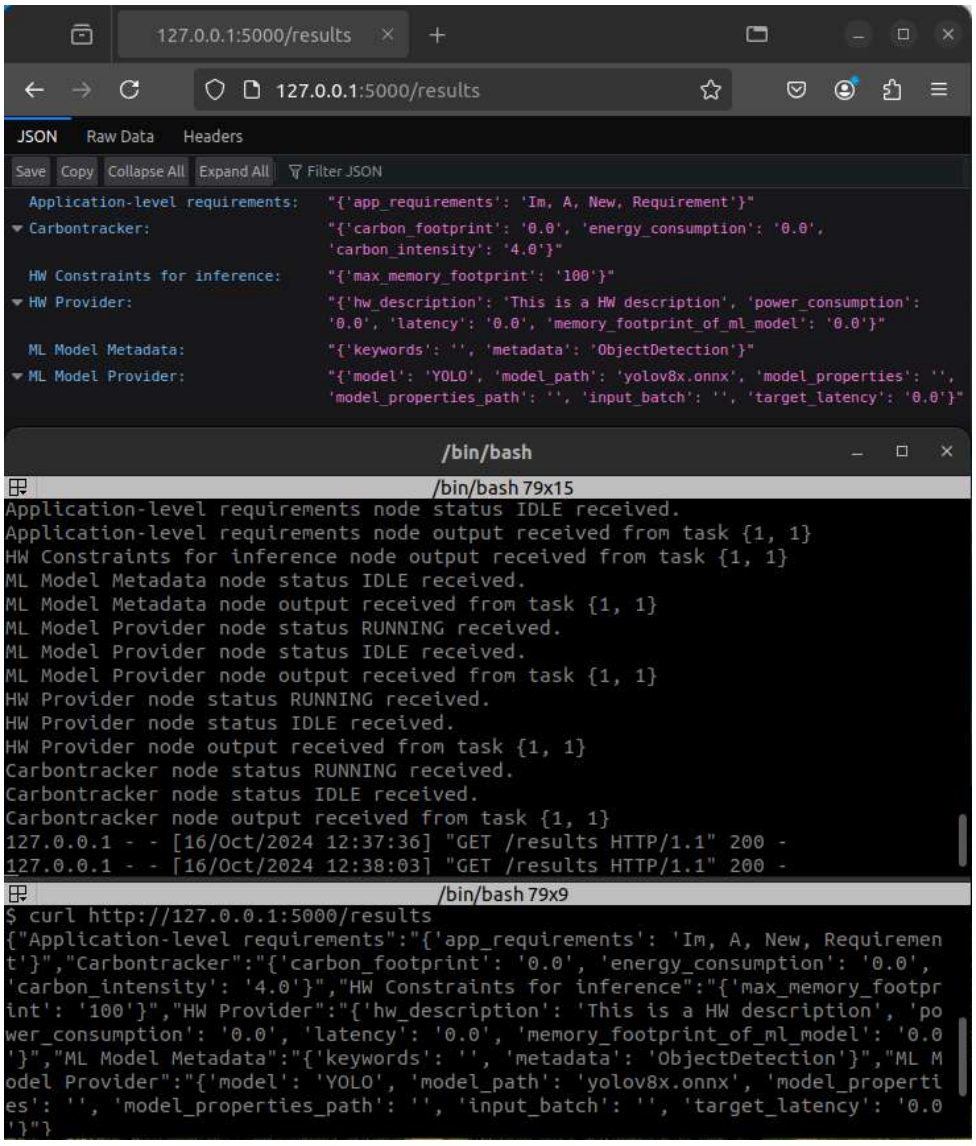

Figure 19: Orchestator Python Node HTTP results request demonstrator

8 Conclusions

This document presented the software design of the *SustainML Design Framework*. It is not intended to be the final design but an insight on how it is going to be built, and it could change as the project advances depending on the requirements that may arise or the limitations encountered during the development process. However, it is intended to be a complete, robust and functional design fulfilling all the SustainML project Software requirements (M12 - Deprecated).

The *SustainML Design Framework* software design has the following strengths:

Decoupled modular design The modular design nature of the *SustainML Design Framework* exploits the distributed capabilities of the underlying DDS protocol and makes easier the integration of the different software modules developed by each of the projects' partners.

Scalability The software design internals of the *SustainML Design Framework* are meant to be for long-term, highly scalable and flexible software product. The design principles applied ensure robustness and a quick response to quick changes.

Adaptability The *SustainML Design Framework* is endowed with a highly configurable network Quality Of Service policies (Enabling technologies) to adapt to a wide variety of network scenarios, capable of being optimally adjusted to each of them i.e loosy networks, restricted bandwidth scenarios, WAN (Wide Area Network) or LAN (Local Area Network), intra-process/inter-process, or a mix between all of them.

In addition, and complying with SustainML project objectives, the *SustainML Design Framework* can be carefully customized to fit low power consumption hardware devices with limited resources. This kind of hardware will be frequently preferred, as it represents a more sustainable, energy-optimized AI solution with a lower CO2 footprint.

Security The security of communications when working across networks is crucial, so it is a matter to be taken into account in every framework including network communications. *SustainML Design Framework* internal nodes, exchange data to provide an accurate, green AI solution to the problem described by the user.

The possibility of adding user authentication, access permissions and data encryption over the network is given by *Fast DDS* communications middleware, avoiding the need for extra security features. This, and other secure mechanisms to join a network of SustainML nodes will be considered for future versions.

8.1 Future work

The software design of the *SustainML Design Framework* software design comprises two core modules: the Front-End the Back-End. Both have been improved in this stage, improving forms and information about the Module Nodes output in the GUI, and the *baseline* mechanism in the Orchestrator Node. In turn, Back-End can be divided in two components: the Module Nodes and the Orchestrator Node (Figure 2). Some of the Module nodes (Figure 12) already include initial development from their corresponding partners, but there are still some of them that are integration work-in-progress. The Orchestrator Node baseline mechanism is in its initial state and it is open to changes to fulfill the future requirements while integrating the remaining work from the partners in their corresponding Module Nodes.

Further work regarding the Front-End is to be addressed with the feedback of WP4 (Work Packages and partners) with the aim of creating a positive user experience. It has already been defined a new

service requirement to allow a direct communication between Front-End and some of the Module Nodes to query information about available resources, in such way the options offered to the user are always available.

A Data Model

Following are the Data Structures definitions using the Interactive Data Language (IDL). Every node in the Framework needs to provide a continuous feedback status to the Orchestrator Node. This is modeled with the *NodeStatus* data structure.

```
enum Status
{
    NODE_INACTIVE,
    NODE_ERROR,
    NODE_IDLE,
    NODE_INITIALIZING,
    NODE_RUNNING,
    NODE_TERMINATING
};

enum TaskStatus
{
    TASK_WAITING,
    TASK_ERROR,
    TASK_RUNNING,
    TASK_SUCCEEDED
};

enum ErrorCode
{
    NO_ERROR,
    INTERNAL_ERROR
};

struct TaskId
{
    unsigned long problem_id;
    unsigned long iteration_id;
};

struct NodeStatus
{
    Status node_status;
    TaskStatus task_status;
    ErrorCode error_code;
    string error_description;
    @key string node_name;
    @key TaskId task_id;
};
```

Listing 1: Node Status Data Type IDL description.

In order to correctly manage the lifecycle of nodes and tasks, a *NodeControl* data structure is defined. This data structure is internally used in the communication library.

```
enum CmdNode
{
    NO_CMD_NODE,
    START_NODE,
    STOP_NODE,
    RESET_NODE,
    TERMINATE_NODE
};

enum CmdTask
{
    NO_CMD_TASK,
    STOP_TASK,
    RESET_TASK,
    PREEMPT_TASK,
    TERMINATE_TASK
};

struct NodeControl
{
    CmdNode cmd_node;
    CmdTask cmd_task;
    string target_node;
    @key string source_node;
    @key TaskId task_id;
};
```

Listing 2: Node Control Data Type IDL description.

The *UserInput* data structure carries information about the input from the user, when describing a new task (Machine Learning problem). It is comprised of the following fields:

- The name of the task.
- The modality of the input data e.g image/video, text, audio, sensor,...
- The type of problem to solve e.g classification, regression, clustering,...
- A sequence of serialized batches of input data.
- A sequence of serialized batches of output data.
- The minimum number of samples required.
- The maximum number of samples required.
- A boolean indicating if the user wants to manually optimize the carbon footprint.
- A previous iteration from which to perform the optimization (-1 for taking the last one).
- A boolean indicating if the user wants to automatically optimize the carbon footprint until a desired value.
- The desired carbon footprint.

- The geo-location continent in which the ML problem is going to take place.
- The geo-location region in which the ML problem is going to take place.
- A sequence of raw extra data for out-of-scope use cases.
- The task_id identifier of the ML problem to solve.

```
struct UserInput
{
    string modality;
    string problem_short_description;
    string problem_definition;
    sequence<string> inputs;
    sequence<string> outputs;
    unsigned long minimum_samples;
    unsigned long maximum_samples;
    boolean optimize_carbon_footprint_manual;
    long previous_iteration;
    boolean optimize_carbon_footprint_auto;
    double desired_carbon_footprint;
    string geo_location_continent;
    string geo_location_region;
    sequence<octet> extra_data;
    @key TaskId task_id;
};
```

Listing 3: User Input Data Type IDL description.

The *MLModelMetadata* data structure represents the output from the MLMMN. It is composed by:

- A sequence of strings identifying the *keywords* from the user input problem description.
- The *machine learning model metadata*. At the current stage of development, the metadata is generically defined as a sequence of strings.
- A sequence of raw extra data for out-of-scope use cases.
- The task_id identifier of the ML problem to solve.

```
struct MLModelMetadata
{
    sequence<string> keywords;
    sequence<string> ml_model_metadata;
    sequence<octet> extra_data;
    @key TaskId task_id;
};
```

Listing 4: Encoded Task Data Type IDL description.

The *AppRequirements* data structure depicts the output from the ALRN and consists on:

- A sequence of application-level requirements, modeled as a sequence of strings, to be considered in the selection of the machine learning model.
- A sequence of raw extra data for out-of-scope use cases.
- The task_id identifier of the ML problem to solve.

```
struct AppRequirements
{
    sequence<string> app_requirements;
    sequence<octet> extra_data;
    @key TaskId task_id;
};
```

Listing 5: Encoded Task Data Type IDL description.

The *HWConstraints* represents the group of constraints defined (or not) by the user when describing the problem. It is the output from the HWCFIN. It is composed by:

- The maximum memory footprint allowed for the ML model.
- A sequence of raw extra data for out-of-scope use cases.
- The task_id identifier of the ML problem to solve.

```
struct HWConstraints
{
    unsigned long max_memory_footprint;
    sequence<octet> extra_data;
    @key TaskId task_id;
 };
```

Listing 6: Encoded Task Data Type IDL description.

The *MLModel* data structure represents the output from the MPN. It is divided in the following fields:

- A string containing the path to the ML model.
- A string with the model name, in case of remote approach.
- A sequence of bytes with the raw model, in case of remote approach.
- A string containing the path to the properties of the model.
- A string with the model properties, in case of remote approach.
- A sequence of serialized numpy arrays with a dimension: Batch x Channels x Height x Width (each one representing a batch) conforming the input batch.
- The target latency or fps for computer vision tasks or target processing latency in seconds for other tasks, like time-series analysis.
- A sequence of raw extra data for out-of-scope use cases.
- The task_id identifier of the ML problem to solve.

The *model* and *model_properties* can be optionally filled. The reasoning for include them is to overcome situations in which the model is generated into a remote machine.

```
struct MLModel
{
    string model_path;
    string model;
    sequence<octet> raw_model;
    string model_properties_path;
    string model_properties;
```

```
    sequence<string> input_batch;
    double target_latency;
    sequence<octet> extra_data;
    @key TaskId task_id;
};
```

Listing 7: Machine Learning Model Data Type IDL description.

The HWPN selects a best-suited energy-optimized hardware according to the ML model. To represent that information, the *HWResource* data structure is defined containing the following fields:

- A string with the detailed hardware description.
- The power consumption (W).
- The estimation of latency of the given ONNX model for the given input batch.
- The maximum memory footprint that can be implemented on the target FPGA.
- A sequence of raw extra data for out-of-scope use cases.
- The task_id identifier of the ML problem to solve.

```
struct HWResource
{
    string hw_description;
    double power_consumption;
    double latency;
    double memory_footprint_of_ml_model;
    double max_hw_memory_footprint;
    sequence<octet> extra_data;
    @key TaskId task_id;
};
```

Listing 8: Hardware Resource Data Type IDL description.

Finally, in order to model the output from the CN, the *CO2Footprint* data structure consisting in the following fields:

- The CO2 footprint (kgCO2e).
- The energy consumption (Wh).
- The carbon intensity.
- A sequence of raw extra data for out-of-scope use cases.
- The task_id identifier of the ML problem to solve.

```
struct CO2Footprint
{
    double carbon_footprint;
    double energy_consumption;
    double carbon_intensity;
    sequence<octet> extra_data;
    @key TaskId task_id;
};
```

Listing 9: CO2 Footprint Data Type IDL description.

B Data Model (M12 - Deprecated)

Following are the Data Structures definitions using the IDL. Every node in the Framework needs to provide a continuous feedback status to the Orchestrator Node. This is modeled with the *NodeStatus* data structure.

```
enum Status
{
        INACTIVE,
        INITIALIZING,
        IDLE,
        RUNNING,
        ERROR,
        TERMINATING
};

enum TaskStatus
{
        WAITING,
        RUNNING,
        ERROR,
        SUCCEEDED
};

enum ErrorCode
{
        NO_ERROR,
        INTERNAL_ERROR
};

struct NodeStatus
{
    Status node_status;
    TaskStatus task_status;
    ErrorCode error_code;
    @Key long task_id;
    string error_description;
    @Key string node_name;
}
```

Listing 10: Node Status Data Type IDL description.

In order to correctly manage the lifecycle of nodes and tasks, a *NodeControl* data structure is defined. This data structure is internally used in the communication library.

```
enum CmdNode
{
    NO_CMD,
    CMD_START_NODE,
    CMD_STOP_NODE,
    CMD_RESET_NODE,
    CMD_TERMINATE_NODE
};
```

```
9
10 enum CmdTask
11 {
12     NO_CMD,
13     CMD_STOP_TASK,
14     CMD_RESET_TASK,
15     CMD_PREEMPT_TASK,
16     CMD_TERMINATE_TASK
17 };
18
19 struct NodeControl
20 {
21
22     CmdNode cmd_node;
23     CmdTask cmd_task;
24     string target_node;
25     long task_id;
26     @Key string source_node;
27 };
```

Listing 11: Node Control Data Type IDL description.

The *UserInput* data structure carries information about the input from the user, when describing a new task (Machine Learning problem). It is comprised of the following fields:

- A string with the raw input problem description.
- The geo-location in which the ML problem is going to take place.
- The task_id identifying the id of the ML problem to solve.

```
1 struct GeoLocation
2 {
3     string continent;
4     string region;
5 };
6
7 struct UserInput
8 {
9     string problem_description;
10     GeoLocation geo_location;
11     @Key long task_id;
12 };
```

Listing 12: User Input Data Type IDL description.

The *EncodedTask* data structure represents the output from the Task Encoder Node. It is composed by:

- A sequence of strings identifying the *keywords* from the user input problem description.
- The task_id identifying the corresponding ML problem.

```
1 struct EncodedTask
2 {
```

```
    sequence<string> keywords;
    @Key long task_id;
};
```

Listing 13: Encoded Task Data Type IDL description.

The *MLModel* data structure represents the output from the ML Model Provider Node. It is divided in the following fields:

- A string containing the path to the ML model.
- A string with the raw model.
- A string containing the path to the properties of the model.
- A string with the model properties.
- The task_id the ML problem it refers to.

The *model* and *model_properties* can be optionally filled. The reasoning for include them is to overcome situations in which the model is generated into a remote machine.

```
struct MLModel
{
    string model_path;
    string model; // Remote cases
    string model_properties_path;
    string model_properties;
    @Key long task_id;
};
```

Listing 14: Machine Learning Model Data Type IDL description.

The Hardware Resources Node selects a best-suited energy-optimized hardware according to the ML model. To represent that information, the *HWResource* data structure is defined containing the following fields:

- A string with the detailed hardware description.
- The power consumption (W).
- The task_id of the task it refers to.

```
struct HWResource
{
    string hw_description;
    double power_consumption;
    @Key long task_id;
};
```

Listing 15: Hardware Resource Data Type IDL description.

Finally, in order to model the output from the Carbon Footprint Node, the *CO2Footprint* data structure consisting in the following fields:

```
struct CO2Footprint
{
    double co2_footprint;
```

```
    double energy_consumption;
    double carbon_intensity;
    @Key long task_id;
};
```

Listing 16: CO2 Footprint Data Type IDL description.

C API usage examples for the software modules

Python implementation template for the ALRN:

```
from sustainml_py.nodes.AppRequirementsNode import AppRequirementsNode

# Manage signaling
import signal
import threading
import time

# Whether to go on spinning or interrupt
running = False

# Signal handler
def signal_handler(sig, frame):
    print("\nExiting")
    AppRequirementsNode.terminate()
    global running
    running = False

# User Callback implementation
# Inputs: user_input
# Outputs: node_status, app_requirements
def task_callback(user_input, node_status, app_requirements):
    ...                                 # set up the job
    status.update(Status.RUNNING)       # notify to the backend that job has started
    ...                                 # perform the job
    status.update(Status.SUCCEEDED) # notify to the backend that job has finished
    app_requirements.app_requirements().append("...") # record the output in the structu

# Main workflow routine
def run():
    node = AppRequirementsNode(callback=task_callback)
    global running
    running = True
    node.spin()

# Call main in program execution
if __name__ == '__main__':
    signal.signal(signal.SIGINT, signal_handler)

    """Python does not process signals async if
    the main thread is blocked (spin()) so, tun
    user work flow in another thread """
    runner = threading.Thread(target=run)
    runner.start()
    while running:
        time.sleep(1)
    runner.join()
```

Listing 17: Implementation template for the ALRN

Python implementation template for the HWCFIN:

```python
from sustainml_py.nodes.HardwareConstraintsNode import HardwareConstraintsNode

# Manage signaling
import signal
import threading
import time

# Whether to go on spinning or interrupt
running = False

# Signal handler
def signal_handler(sig, frame):
    print("\nExiting")
    HardwareConstraintsNode.terminate()
    global running
    running = False

# User Callback implementation
# Inputs: user_input
# Outputs: node_status, hw_constraints
def task_callback(user_input, node_status, hw_constraints):
    ...                                 # set up the job
    status.update(Status.RUNNING)       # notify to the backend that job has started
    ...                                 # perform the job
    status.update(Status.SUCCEEDED) # notify to the backend that job has finished
    hw_constraints.max_memory_footprint(...)  # record the output in the structure

# Main workflow routine
def run():
    node = HardwareConstraintsNode(callback=task_callback)
    global running
    running = True
    node.spin()

# Call main in program execution
if __name__ == '__main__':
    signal.signal(signal.SIGINT, signal_handler)

    """Python does not process signals async if
    the main thread is blocked (spin()) so, tun
    user work flow in another thread """
    runner = threading.Thread(target=run)
    runner.start()
    while running:
        time.sleep(1)
    runner.join()
```

Listing 18: Implementation template for the HWCFIN

Python implementation template for the MLMMN:

```python
from sustainml_py.nodes.MLModelMetadataNode import MLModelMetadataNode

# Manage signaling
import os
import signal
import threading
import time

# Whether to go on spinning or interrupt
running = False

# Signal handler
def signal_handler(sig, frame):
    print("\nExiting")
    MLModelMetadataNode.terminate()
    global running
    running = False

# User Callback implementation
# Inputs: user_input
# Outputs: node_status, ml_model_metadata
def task_callback(user_input, node_status, ml_model_metadata):
    ...                                 # set up the job
    status.update(Status.RUNNING)       # notify to the backend that job has started
    ...                                 # perform the job
    status.update(Status.SUCCEEDED) # notify to the backend that job has finished
    ml_model_metadata.keywords().append("...") # record the output in the structure
    ml_model_metadata.ml_model_metadata().append("...")

# Main workflow routine
def run():
    node = MLModelMetadataNode(callback=task_callback)
    global running
    running = True
    node.spin()

# Call main in program execution
if __name__ == '__main__':
    signal.signal(signal.SIGINT, signal_handler)

    """Python does not process signals async if
    the main thread is blocked (spin()) so, tun
    user work flow in another thread """
    runner = threading.Thread(target=run)
    runner.start()
    while running:
        time.sleep(1)
    runner.join()
```

Listing 19: Implementation template for the MLMMN

Python implementation template for the MPN:

```
from sustainml_py.nodes.MLModelNode import MLModelNode

# Manage signaling
import os
import signal
import threading
import time

# Whether to go on spinning or interrupt
running = False

# Signal handler
def signal_handler(sig, frame):
    print("\nExiting")
    MLModelNode.terminate()
    global running
    running = False

# User Callback implementation
# Inputs: ml_model_metadata, app_requirements, hw_constraints, ml_model_baseline, hw_ba
# Outputs: node_status, ml_model
def task_callback(ml_model_metadata,
                  app_requirements,
                  hw_constraints,
                  ml_model_baseline,
                  hw_baseline,
                  carbonfootprint_baseline,
                  node_status,
                  ml_model):
    ...                                 # set up the job
    status.update(Status.RUNNING)       # notify to the backend that job has started
    ...                                 # perform the job
    status.update(Status.SUCCEEDED)     # notify to the backend that job has finished
    ml_model.model("...")               # record the output in the structure
    ml_model.model_path("...")
    ml_model.model_properties("...")
    ml_model.model_properties_path("...")
    ml_model.target_latency("...")

# Main workflow routine
def run():
    node = MLModelNode(callback=task_callback)
    global running
    running = True
    node.spin()

# Call main in program execution
if __name__ == '__main__':
    signal.signal(signal.SIGINT, signal_handler)
```

```
"""Python does not process signals async if
the main thread is blocked (spin()) so, tun
user work flow in another thread """
runner = threading.Thread(target=run)
runner.start()
while running:
    time.sleep(1)
runner.join()
```

Listing 20: Implementation template for the MPN

Python implementation template for the HWPN:

```
from sustainml_py.nodes.HardwareResourcesNode import HardwareResourcesNode

# Manage signaling
import signal
import threading
import time

# Whether to go on spinning or interrupt
running = False

# Signal handler
def signal_handler(sig, frame):
    print("\nExiting")
    HardwareResourcesNode.terminate()
    global running
    running = False

# User Callback implementation
# Inputs: ml_model, app_requirements, hw_constraints
# Outputs: node_status, hw
def task_callback(ml_model, app_requirements, hw_constraints, node_status, hw):
    ...                                 # set up the job
    status.update(Status.RUNNING)       # notify to the backend that job has started
    ...                                 # perform the job
    status.update(Status.SUCCEEDED)     # notify to the backend that job has finished
    hw.hw_description("...")            # record the output in the structure
    hw.power_consumption(...)
    hw.latency(...)
    hw.memory_footprint_of_ml_model(...)

# Main workflow routine
def run():
    node = HardwareResourcesNode(callback=task_callback)
    global running
    running = True
    node.spin()

# Call main in program execution
if __name__ == '__main__':
    signal.signal(signal.SIGINT, signal_handler)

    """Python does not process signals async if
    the main thread is blocked (spin()) so, tun
    user work flow in another thread """
    runner = threading.Thread(target=run)
    runner.start()
    while running:
        time.sleep(1)
```

```
runner.join()
```

Listing 21: Implementation template for the HWPN

© 2024 SustainML | HORIZON-CL4-2021-HUMAN-01 | 101070408

Python implementation template for the CN:

```python
from sustainml_py.nodes.CarbonFootprintNode import CarbonFootprintNode

# Manage signaling
import signal
import threading
import time

# Whether to go on spinning or interrupt
running = False

# Signal handler
def signal_handler(sig, frame):
    print("\nExiting")
    CarbonFootprintNode.terminate()
    global running
    running = False

# User Callback implementation
# Inputs: ml_model, user_input, hw
# Outputs: node_status, co2
def task_callback(ml_model, user_input, hw, node_status, co2):
    ...                                 # set up the job
    status.update(Status.RUNNING)       # notify to the backend that job has started
    ...                                 # perform the job
    status.update(Status.SUCCEEDED)     # notify to the backend that job has finished
    co2.carbon_footprint(...)           # record the output in the structure
    co2.energy_consumption(...)
    co2.carbon_intensity(...)

# Main workflow routine
def run():
    global running
    running = True
    node = CarbonFootprintNode(callback=task_callback)
    node.spin()

# Call main in program execution
if __name__ == '__main__':
    signal.signal(signal.SIGINT, signal_handler)

    """Python does not process signals async if
    the main thread is blocked (spin()) so, tun
    user work flow in another thread """
    runner = threading.Thread(target=run)
    runner.start()
    while running:
        time.sleep(1)
    runner.join()
```

Listing 22: Implementation template for the CN

D API usage examples for the software modules (M12 - Deprecated)

Python implementation template for the TEN:

```
from SustainML import TaskEncoderNode
from SustainML import UserInput
from SustainML import NodeStatus
from SustainML import EncodedTask

def user_input_cb(user_input, module_status, output):
    ...                                 # set up the job
    status.update(Status.RUNNING)       # notify to the backend that job has
        started
    ...                                 # perform the job
    status.update(Status.SUCCEEDED) # notify to the backend that job has
        finished
    output.keywords = ...               # record the output in the structure

if __name__ == "__main__":
    node = TaskEncoderNode()                # register node
    node.register_input_cb(user_input_cb) # select callback
    node.spin()                             # start the job
```

Listing 23: Implementation template for the TEN

Python implementation template for the MLN:

```python
from SustainML import MLModelNode
from SustainML import EncodedTask
from SustainML import NodeStatus
from SustainML import MLModel

def encoded_task_cb(encoded_task, status, output):
    ...                                 # set up the job
    status.update(Status.RUNNING)       # notify to the backend that job has started
    ...                                 # perform the job
    status.update(Status.SUCCEEDED)     # notify to the backend that job has finished
    output.model_path = ...             # record the output in the structure
    output.model_properties = ...

if __name__ == "__main__":
    node = MLModelNode()                        # register node
    node.register_input_cb(encoded_task_cb)     # select callback
    node.spin()                                 # start the job
```

Listing 24: Implementation template for the MLN

Python implementation template for the HWR:

```
from SustainML import HardwareResourcesNode
from SustainML import MLModel
from SustainML import NodeStatus
from SustainML import HardwareResource

def ml_model_cb(ml_model, status, output):
    ...                                 # set up the job
    status.update(Status.RUNNING)       # notify to the backend that job has
        started
    ...                                 # perform the job
    status.update(Status.SUCCEEDED) # notify to the backend that job has
        finished
    output.hw_description = ...         # record the output in the structure
    output.power_consumption = ...

if __name__ == "__main__":
    node = HardwareResourcesNode()      # register node
    node.register_input_cb(ml_model_cb) # select callback
    node.spin()                         # start the job
```

Listing 25: Implementation template for the HWN

Python implementation template for the CFN:

```
from SustainML import CarbonFootprintNode
from SustainML import HWResource
from SustainML import UserInput
from SustainML import MLModel
from SustainML import NodeStatus
from SustainML import HardwareResource

def input_cb(hw_resource, user_input, ml_model, status, output):
    ...                                 # set up the job
    status.update(Status.RUNNING)   # notify to the backend that job has
        started
    ...                                 # perform the job
    status.update(Status.SUCCEEDED) # notify to the backend that job has
        finished
    output.co2_footprint = ...          # record the output in the structure
    output.energy_consumption = ...
    output.carbon_intensity = ...

if __name__ == "__main__":
    node = CarbonFootprintNode()      # register node
    node.register_input_cb(input_cb) # select callback
    node.spin()                         # start the job
```

Listing 26: Implementation template for the CFN

E Screenshots of the Module Nodes demonstrator

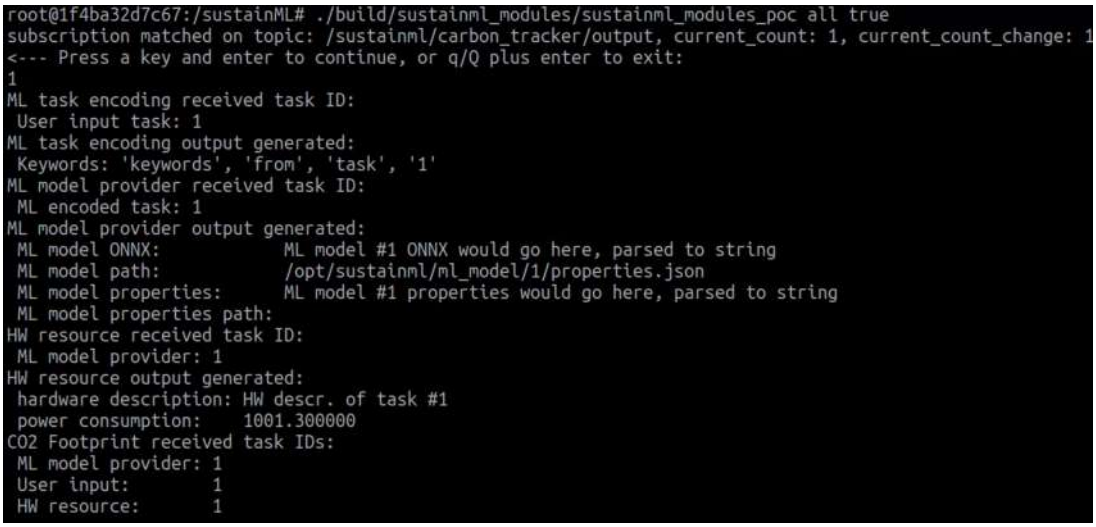

Figure 20: Module Nodes demonstrator in single process scenario

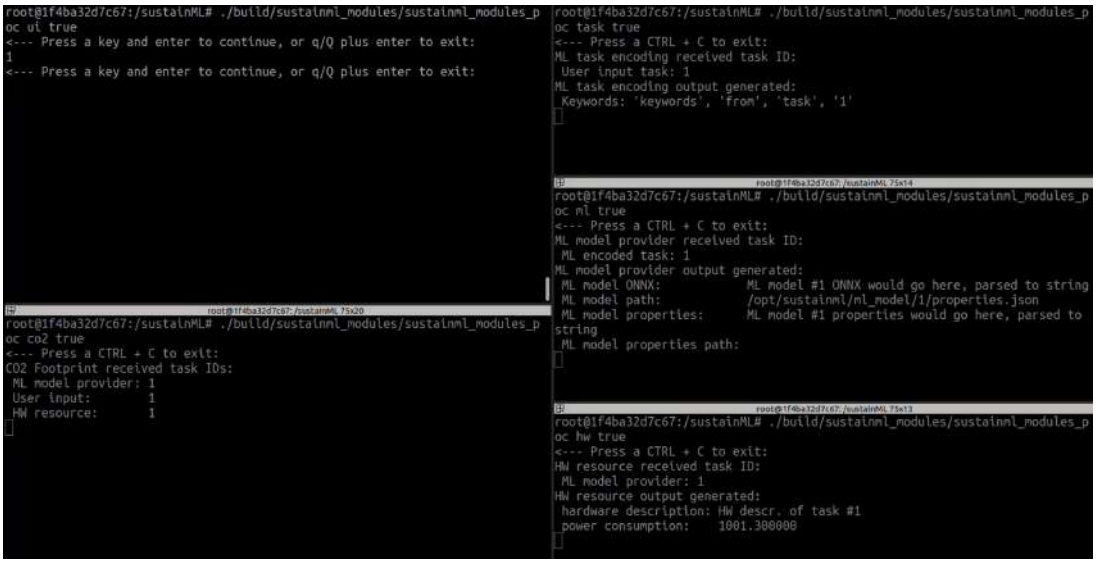

Figure 21: Module Nodes demonstrator in inter-process scenario

References

[1] *eProsima Fast DDS Documentation - What is DDS?* Version 2.11.2. Proyectos y Sistemas de Mantenimiento SL (eProsima), 2023. URL: https://fast-dds.docs.eprosima.com/en/v2.11.2/fastdds/getting_started/definitions.html.

[2] *eProsima Fast DDS Documentation*. Version 2.11.2. eProsima, 2023. URL: http://fast-dds.docs.eprosima.com/en/v2.11.2/.

[3] *eProsima Fast DDS Documentation - Transport Layer*. Version 2.11.2. Proyectos y Sistemas de Mantenimiento SL (eProsima), 2023. URL: https://fast-dds.docs.eprosima.com/en/v2.11.2/fastdds/transport/transport.html.

[4] *Data Distribution Service (DDS)*. Object Management Group, 2015. URL: http://www.omg.org/spec/DDS/1.4/About-DDS/.

[5] *eProsima Fast DDS Documentation - Quality Of Service*. Version 2.11.2. Proyectos y Sistemas de Mantenimiento SL (eProsima), 2023. URL: https://fast-dds.docs.eprosima.com/en/v2.11.2/fastdds/dds_layer/core/policy/standardQosPolicies.html.

[6] *Design Principles and Design Patterns*. Robert C. Martin, 2000.

[7] *Unified Modelling Language*. Version 2.5.1. Object Management Group, 2017. URL: https://www.omg.org/spec/UML/.

[8] Raúl Sánchez-Mateos Lizano. *Data Management Plan (M6)*. Version 1.1. SustainML: Application Aware, Life-Cycle Oriented Model-Hardware Co-Design Framework for Sustainable, Energy Efficient ML Systems. eProsima, 2023.