



Funded by
the European Union

Grant Agreement number: 101092696
Topic: HORIZON-CL4-2022-DATA-01



CODECO

Cognitive Decentralised
Edge Cloud Orchestration

D11: CODECO Basic Operation and Open Toolkit v1.0

Work package	WP3 – CODECO Basic Operation components and Toolkit
Internal number	D3.1
Task	Tasks 3.1, 3.2, 3.3, 3.4, 3.5, 3.6
Due date	31.10.2023
Submission date	31.10.2023
Dissemination Type	PUBLIC
Deliverable leaders and editors	ICOM (Nikos Psaromanolakis, Vasileios Theodorou)
Contributing Partners	FOR (Rute C. Sofia, Dalal Ali); ICOM (Nikos Psaromanolakis, Maria Eleftheria Vlontzou, Andreas Akarepis, Vasileios Theodorou); ATH (Lefteris Mamatas, George Koukis, Ioanna Kapetanidou); SIE (Harald Müller); I2CAT (Rizkallah Touma, Alejandro Espinosa); UPRC (Efterpi Paraskevoulakou, Panagiotis Karamolegkos); TID (Alejandro Muniz da Costa, Luis Contreras Murillo); UC3M (Borja Nogales Dorado), UPM (Alberto del Rio Ponce); Almende (Andries Stam); RHT (Josh Salomon, Simone Ferlein-Reiter); IBM (Luis Garcia-Erices, Peter Urbanetz), INO (Vitor Vieira); UGOE (Xiaoming Fu); INTRA (John Soldatos, Dorine Matzakou); ATOS (Ignacio Prusiel Mariscal);
Version	1.0
Reviewer 1	FOR (Rute C. Sofia)



Funded by
the European Union



Funded by the Swiss Federal Government

Project Partners



Affiliated Entities



Executive Summary

This report is an integral part of the CODECO deliverable **D11 - CODECO Basic Operation Components and Toolkit v1.0**. D11 consists of this report and an early CODECO software release consisting of components of the CODECO open source "Basic Operation Toolkit v1.0", available in the [CODECO Eclipse GitLab repository](#). This first open source software toolkit will be fully released in June 2024 (month 18 of CODECO) and will provide the CODECO framework operational support for a cluster.

D11 and the software developed is a product of the work under development in CODECO WP3 (CODECO Basic Operation and Toolkit v1.0) and is a direct result of the work developed in T3.6 (Open Edge-Cloud Toolkit Development), the task focusing on the implementation and testing of the CODECO Toolkit based on the software components developed in Tasks 3.1 to 3.5.

Keywords: Edge-Cloud continuum; orchestration; Kubernetes; AI/ML; network; data; compute; context-awareness.

Document Revision History:

Version	Date	Description of change	List of contributors(s)
v0.1	31.07.2023	Proposal for a global ToC	Nikos Psaromanolakis
v0.2	11.08.2023	Revised Proposal for a global ToC	WP3 Task leaders
v0.3	05.09.2023	Revised Proposal for a global ToC	Nikos Psaromanolakis
v0.4	13.10.2023	First contributions by some partners	ICOM, UPRC, I2CAT, UPM
v0.5	18.10.2023	Added more contributions	ICOM, UPRC, I2CAT, UPM, FOR, SWM, ATH, RHT, TID
v0.6	19.10.2023	Updated contributions	ICOM, UPRC, I2CAT, UPM, FOR, SWM, ATH, RHT, TID, UC3M
v0.7	20.10.2023	Updated contributions	All partners
v0.8	24.10.2023	Updated contributions	All partners
v0.9	25.10.2023	Review and update contributions	All partners
v0.91	27.10.2023	Update contribution, release to internal reviewers	All partners
v0.92	29.10.2023	Updated contributions based on internal reviewers' comments	Nikos Psaromaolakis (ICOM), Rute C. Sofia (FOR), Ioanna Kapetanidou (ATH)
v0.93	30.10.2023	Internal review, updated contributions based on comments	Nikos Psaromaolakis (ICOM), Maria Eleftheria Vlontzou (ICOM), Rizk Allah Touma (I2CAT)
v1.0	31.10.2023	Final review and formatting by Coordinator	Rute C. Sofia (FOR)

Disclaimer

The information, documentation, and figures available in this deliverable have been developed by the Horizon Europe CODECO project consortium, under the European Union grant Agreement number 101092696. The content does not necessarily reflect the views of the European Commission. The European Commission is not liable for any use that may be made of the information contained herein.

Copyright notice: © 2023 - 2025 CODECO Consortium



Funded by
the European Union

Table of Contents

Executive Summary	3
1 Introduction	9
1.1 Document Scope	9
1.2 Dependencies	10
1.3 Document Structure	10
2 CODECO Architectural Design and Operational Workflow	10
2.1 CODECO Architecture and Components Summary	10
2.2 CODECO Workflow Example, Single Cluster	14
2.2.1 Creating an Application Deployment with CODECO	14
2.2.2 CODECO Support during Cluster Runtime	14
3 Specification and Implementation of CODECO Components	15
3.1 ACM: Automated Configuration Management	15
3.1.1 Component Description	15
3.1.2 Application Model description	17
3.1.3 Sub-components' Specification and Implementation	19
3.1.4 Next Cycle features	21
3.2 PDL: Privacy-preserving Decentralised Learning and Context-awareness	21
3.2.1 Component Description	21
3.2.2 Sub-components' Specification and Implementation	23
3.2.3 Next Cycle features	43
3.3 NetMA: Network Management and Adaptation	43
3.3.1 Component Description	43
3.3.2 Sub-components' Specification and Implementation	44
3.3.3 Next Cycle Features	59
3.4 MDM: Metadata Manager	59
3.4.1 Component Description	59
3.4.2 Sub-components' Specification and Implementation	60
3.4.3 Next Cycle features	68
3.5 SWM: Scheduling and Workload Migration	68
3.5.1 Component description	68
3.5.2 Sub-components features specification, Inputs & Outputs	68
3.5.3 Next Cycle features	69
4 Experimentation Assisted Developments	69
4.1 CODECO Synthetic Data Generators	69
4.1.1 Component Description	69
4.1.2 Component Specification and Implementation Aspects	70
4.2 Facility Porting Artifacts	74
4.2.1 EdgeNet Testbed	74
5 Continuous Integration, Testing, Deployment Preparation and Releasing	79
5.1 Testing Methodology	79
5.2 Deployment Preparation and Releasing	80
5.2.1 Deployment and CI/CD Methodology	80
5.2.2 Integration Testbed Environment	81
6 Summary: Current Status and Next Steps	84
7 References	86
Annex I – CODECO Cross-Layer Metrics	87
Annex II – Release Versioning	90
Annex III – Automating software builds, testing, packaging, and deployment	93



List of Figures

Figure 1. The CODECO K8s framework and its components.	11
Figure 2. CODECO and its components and interfaces. ACM and SWM reside on the control plane (master node); MDM, PDLC and NetMA reside on the worker nodes (service plane).	13
Figure 3. ACM and sub-components: High-level overview with interfaces described in Table 1.	16
Figure 4: simplified CODECO Application Model representation.	17
Figure 5. Example for a finer-grained definition of the CODECO Application Model.	18
Figure 6: Snippets of a potential structure for the CODECO Application Model.	19
Figure 7. CODECO usage scenarios (via ACM).	19
Figure 8. PDLC high-level architecture and interfaces.	22
Figure 9. PDLC sub-components and their interactions with other CODECO components.	22
Figure 10. UML Sequence diagram of PDLC-CA.	27
Figure 11. PDLC UML sequence diagram of interactions between the RL model classes. ...	32
Figure 12: Plot of k8s-worker-3 memory usage timeseries extracted from the Synthetic Data Generator.	36
Figure 13. Plot of the STGNN model is actual and forecasted memory usage values for 'k8s-worker-3' node.	37
Figure 14. PDLC UML Sequence diagram of GNN models.	37
Figure 15. PDLC A3C UML sequence diagram.	39
Figure 16: UML diagram for the PDLC MLOps sub-component.	43
Figure 17. NetMA high-level architecture.	44
Figure 18. NetMA Network Exposure UML Diagram.	47
Figure 19. NetMA, network performance probe UML sequence diagram.	49
Figure 20: NetMA MEC Enablement UML diagram.	52
Figure 21. Secure Connectivity design for the initial phase.	53
Figure 22. UML diagram of performance metrics collection by LPM module.	59
Figure 23. MDM sub-components and APIs to other components.	60
Figure 24. Interaction between MDM subcomponents.	66
Figure 25. Data Traffic Generator's flow and its interaction with the respective CODECO components and its functionality.	73
Figure 26. CODECO-component Controller sequence diagram (in this figure: ACM controller).	73

List of Tables

Table 1: ACM interfaces to other CODECO components.	16
Table 2: Status of PDLC interfaces to other CODECO components.....	23
Table 3: PDLC-CA code summary.	24
Table 4: Scripts included in I2CAT contribution.	28
Table 5: Examples on how to use RL code.	31
Table 6: NetMA interfaces to other CODECO components.	44
Table 7: NetMA Network Exposure sub-component code summary.....	45
Table 8: NetMA MEC Enablement sub-component code summary.....	50
Table 9: Status of the CODECO proposed experimental and integration framework.	81
Table 10: First proposed structure of cluster types to dimension the CODECO shared-Cloud space.	82
Table 11: Type of nodes envisioned for the CODECO Cloud-based multi-cluster environment dimensioning.....	83
Table 12: Examples for minimum system requirements, different K8s technologies.....	83
Table 13: CODECO Application Model Attributes, spec, and status.....	87
Table 14: Minimum subset of CODECO user parameters to be collected in ACM and used in the orchestration.	88
Table 15: Minimum subset of CODECO data observability parameters to be collected in MDM and used in the orchestration.....	88
Table 16: Minimum subset of CODECO networking parameters to be collected in NetMA and used in the orchestration.....	89
Table 17. CODECO subcomponents & Assisted Developments (Open Source) releasing versions.	90



List of Acronyms

Acronym	Meaning
A3C	Asynchronous Advantage Actor-Critic
A3T-GCN	Attention-Temporal Graph Convolution Network
ACM	Automated Configuration Manager
AI	Artificial Intelligence
ALTO	Application-Layer Traffic Optimization
AP	Access Point
API	Application Programming Interface
APOC	Awesome Procedures on Cypher
AR	Augmented Reality
ARM	Advanced RISC Machines
BGP	Border Gateway Protocol
CA	Context-awareness
CD	Compute Domain
CDN	Content Delivery Network
CI/CD	Continuous Integration/ Continuous Deployment
CLI	Command Line Interface
CNI	Container Network Interface
CODECO	Cognitive, Decentralised Edge-Cloud Orchestration
CP	Customer Premises
CPU	Central Processing Unit
CR	Custom Resource
CRD	Custom Resource Definition
DL	Decentralised Learning
DEV	Developer
DL	Deep Learning
DQN	Deep Queue Learning
DQN	Deep Q Network
EC	European Commission
EN	Edge Node
ETSI	European Telecommunication Standards Institute
FaaS	Function as a Service
FIDO	Fast Identity Online
FL	Federated Learning
GCN	Graph Convolution Network
GNNs	Graph Neural Networks
GRU	Gated Recurrent Unit
GUI	Graphical User Interface
HE	Horizon Europe
HTTP	Hypertext Transfer Protocol
IETF	Internet Engineering Task Force
IP	Intellectual Property
IP	Internet Protocol
JSON	Javascript Object Notation
K8s	Kubernetes
KCP	Kubernetes like Control Plane
KPI	Key Performance Indicator
L2	Layer 2
L2S-M	Link-Layer Secure connectivity for Microservice platforms
L3	Layer 3
LAN	Local Area Network
LLDP	Link Layer Discovery Protocol
LPM	L2S-M Performance Measurements
LSTM	Long-Short Term Memory



Acronym	Meaning
MARL	Multi-Agent Reinforcement Learning
MDM	Metadata Manager
MEC	Multi-access Edge computing
MGR	Infrastructure Manager
ML	Machine Learning
MLOps	Machine Learning Operations
MSE	Mean Squared Error
ND	Network Domain
NetMA	Network Management and Adaptation
NN	Neural Network
OAS	OpenAPI Specification
OCM	Open Cluster Management
OS	Operating System
PDLC	Privacy-preserving Decentralised Learning and Context-awareness
PLS	Programmable Link-layer Switches
PPO	Proximal Policy Optimization
PPO	Proximal Policy Optimisation
QoE	Quality of Experience
QoS	Quality of Service
REST	Representational State Transfer
RFC	Request for Comments
RL	Reinforced Learning
RPC	Remote Procedure Call
SCCO	Single Cluster Connectivity Orchestrator
SDK	Software Development Kit
SDN	Software Defined Network
SLICES	Scientific Large Scale Infrastructure for Computing/Communication Experimental Studies
STGNN	Spatio-Temporal Graph Neural Network
SWM	Scheduling and Workload Migration
TEE	Trusted Execution Environment
T-GCN	Temporal Graph Convolutional Network
UI	User Interface
UML	Unified Modelling Language
VXLAN	Virtual eXtensible Local Area Network
YAML	YAML Ain't Markup Language

Acknowledgements

We thank all CODECO Partners involved in WP3 for the commitment and involvement in the development of Deliverable D11.

1 Introduction

CODECO WP3 has been developing the CODECO open source framework since M1 and will continue until M18 (June 2024). The work of the WP includes the selection of appropriate open source tools (based on the architecture defined in WP2 and published in month 6 of the project, Deliverable D9 [1]), implementation and testing. The basic CODECO operation concerns the different proposed modules that can support the development of a flexible, cognitive, and self-organising edge cloud based on a single cluster environment. In WP3, tasks 3.1 to 3.5 deal with the CODECO components and their respective interfaces to other CODECO components and to the user as defined in D9. Task 3.6 deals with the overall system integration and is orthogonal to all WP3 tasks.

D11 - CODECO Basic Operation Components and Toolkit v1.0 is a deliverable consisting of this report and an [early version of the first open source CODECO toolkit \(Basic Operation Toolkit\)](#). This early version, produced after nine months of work, allows early developers to start experimenting with some components of CODECO. Thus, D11 is an intermediate deliverable of CODECO, developed in the context of CODECO WP3 (CODECO Basic Operation and Toolkit v1.0) and is a result of the work developed in T3.6 (Open Edge-Cloud Toolkit Development), the task focusing on the implementation and testing of the CODECO toolkit based on the software components developed in Tasks 3.1 to 3.5.

The work discussed in D11 has been based on an Agile methodology, where CODECO components and their sub-components were developed while the CODECO architecture was being defined (rf. to D9, released in M9).

D11 aim is twofold. Firstly, to facilitate the development, integration, and deployment of the CODECO framework following an incremental approach, which will allow the coordination of the work of the consortium members belonging to different organisations as well as external developers, as CODECO offers open source code, and reduce the complexity of the project. Secondly, to allow access by early developers to a partial release of the CODECO Basic toolkit, following the guidelines for the development of a robust open source community.

1.1 Document Scope

As explained, D11 contains an intermediate version of the open source CODECO Basic Operation Toolkit and an explanation of its software design. This deliverable will be updated to a final version in June 2024. This deliverable provides:

- An explanation of the CODECO integration framework and implementation procedures for the CODECO Basic Operation Toolkit.
- A description of the tools adopted to support the defined implementation workflow; coding guidelines that can optimise the developer's coding effort and time, installation instructions, and tools/ programming languages used as the basis for technology development in the context of the CODECO project.

D11 is composed of the following parts:

- This report (software companion report).
- The early release of parts of the CODECO Basic Operation Toolkit available via the CODECO Eclipse Gitlab¹.

¹ <https://gitlab.eclipse.org/eclipse-research-labs/codeco-project>

- A private report (CONFIDENTIAL) covering the overall specification and software for the CODECO component SWM, which is currently undergoing an open source release process².

1.2 Dependencies

D11 has the following dependencies:

- D9 - CODECO Technological Guidelines, Reference Architecture, and Initial Open source Ecosystem Design intermediate version. D9 provides the overall CODECO reference architecture and explains decisions in terms of open source tooling being used in CODECO [1].

1.3 Document Structure

This report is structured as follows:

- Section 1 introduces the overall deliverable scope, structure, and components.
- Section 2 provides a summary of the CODECO architectural design and highlights the sub-components that are being released in this deliverable.
- Section 3 provides the companion reporting for the use of the existing code, detailed per sub-component.
- Section 4 covers additional software artifacts, such as data generators, that have been developed to assist the use of CODECO.
- Section 5 describes the continuous system testing and the deployment framework that has been set in T3.6 to assist the overall project, as well as the released version of each subcomponent/ feature developed in the first implementation cycle of CODECO.
- Section 6 concludes the document.
- Annex I provides a list of CODECO defined metrics.
- Annex II provides the listing of the current CODECO sub-component release versioning, including URLs for the open source code, and URLs for the respective Docker images.
- Annex III describes the automated process implemented in CODECO for building, testing, and providing the code.

2 CODECO Architectural Design and Operational Workflow

This section provides a summary of the status of the current design of the CODECO architectural framework, to assist the reader in understanding the software developed to date. Details on the architectural design, components, and interfacing within CODECO are provided with more detail in D9 [1] and are therefore not repeated in this deliverable. The terminology used, compatible with *Kubernetes (K8s)*, has already been described in D9.

2.1 CODECO Architecture and Components Summary

An underlying assumption in CODECO is that next-generation Internet applications are now based on microservice architectures, where their components are typically containerised. The deployment of applications in the edge cloud is managed by container orchestrators such as K8s, tools that provide partially automated support for the setup and lifecycle management of containerised applications. Tasks handled by orchestrators include configuring, scheduling,

² This process is internal to partner SIE and is expected to be completed until December 2023.

and deploying containers; checking container availability; scaling the system to balance application workloads across the entire infrastructure; allocating resources to the various containers and monitoring their health; and enabling communication exchanges between containers. Figure 1 provides a high-level representation of the CODECO components. CODECO is a software-based orchestration framework interoperable with K8s that aims at providing support to a next generation of container orchestrators which can adapt and learn, developing a proper reaction and adaptation.

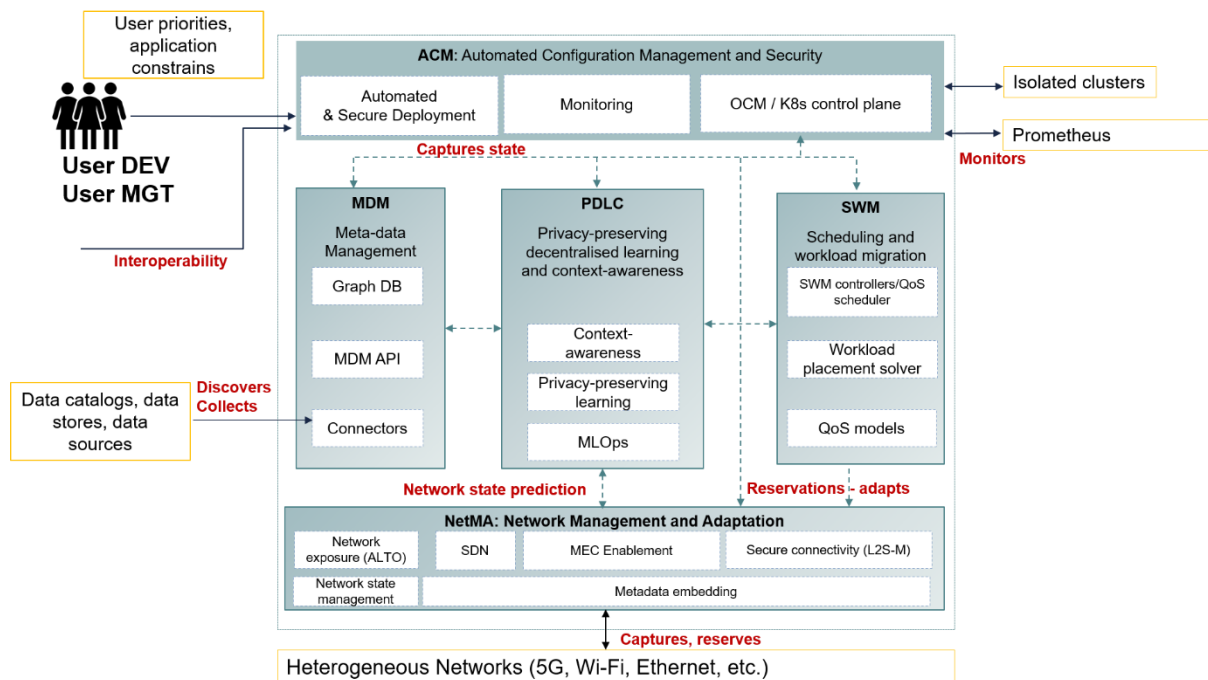


Figure 1. The CODECO K8s framework and its components³.

The only interface of CODECO to the user, as shown, is the ACM component, which focuses on supporting application setup and runtime from the far edge to the cloud, considering the input provided by the user. ACM installs the entire CODECO infrastructure and the respective integration points between users and applications, where the user in CODECO is of type DEV (application developer) or MGR (cluster manager). ACM takes care of the overall CODECO configuration, the acquisition of new nodes, and the interaction with non-K8s systems. Furthermore, ACM relies on Prometheus and integrates a CODECO monitoring framework, currently focused on infrastructure monitoring (data, network, computation) based on application requirements and still under development. ACM is therefore co-located with the control plane of the K8s (master nodes).

The CODECO MDM component provides data workflow observability to the other CODECO components, treating data as an integral part of the application workload, and integrating data observability perspectives from different categories, e.g., application perspective, system perspective, network perspective, at different points in the CODECO operational workflow.

SWM handles the scheduling and rescheduling of the application workload, based on the CODECO Application Model (supported by ACM and provided by the user during application setup), based on the novel data-compute-network approach proposed by CODECO.

³ The NetMA sub-component “network state management” has been previously named “network state forecasting” in CODECO D9. As this component integrates a monitoring and a forecasting component, the consortium has agreed to change its name.

The currently available approach for handling placement decisions relies on a solver⁴ which in the future is expected to provide an optimal match between application requirements and available resources (computational, network, data). SWM is also a control plane component, co-located with ACM and the K8s control plane, in master nodes.

PDLC is at the heart of the CODECO orchestration. Based on the infrastructure data collected by ACM (via Prometheus), NetMA, and MDM, PDLC has two functions. Firstly, it provides an aggregated cost view of a specific target performance profile for the available infrastructure which can be used by other components and is currently being considered in SWM to further define the optimal workload placement. Secondly, it provides an estimate on the overall system stability based on privacy-preserving decentralised learning approaches. PDLC is currently envisaged to operate on both K8s master and worker nodes.

NetMA provides the network awareness to CODECO and handles also secure connectivity across Pods. For this purpose, NetMA exposes networking parameters that are relevant to do an optimal workload placement via the ALTO protocol [2]. For the connectivity, NetMA handles the SDN to K8s interaction via the L2S-M open source solution⁵. Its sub-component Network State Management handles network monitoring, and also receives network forecasting provided by PDLC.

Monitoring of the overall infrastructure from different perspectives is supported by different CODECO components:

- NetMA monitors the networking infrastructure.
- MDM monitors the data infrastructure.
- ACM monitors the system (computational nodes) infrastructure.

Each CODECO component is based on a modular approach, integrating different sub-components that are expected to be built as one or more independent (dockerized) micro-services. The overall interaction between the CODECO components is illustrated in Figure 2, considering as starting point the CODECO operation within a single cluster deployment (M1-M18 of CODECO). In the figure, the following aspects are to be considered:

- We have highlighted the sub-components and interfaces that are available in this early release of code. **Green** means that the sub-component/interface is available; **orange** means that we have released partial code or a proof-of-concept for a sub-component; **red** means that there is no code available yet.
- For the case of SWM, the available components are only accessible by the CODECO consortium; the open source release is expected until December 2023.
- I-comp1-comp2-X describes components from CODECO component 1 to CODECO component 2. For instance, I-MDM-PDLC-1 means that an interface from MDM to PDLC of type *Input*, *Output*, or *bi-directional* is being conceived.

⁴ The current context-aware scheduling approach embodied in SWM is based on background work by partner Siemens. The derivative of this background work is under an OSS release process, which is expected to be concluded in December 2023. A private report on SWM is available for the Consortium and EC.

⁵ <https://github.com/Networks-it-uc3m/L2S-M>



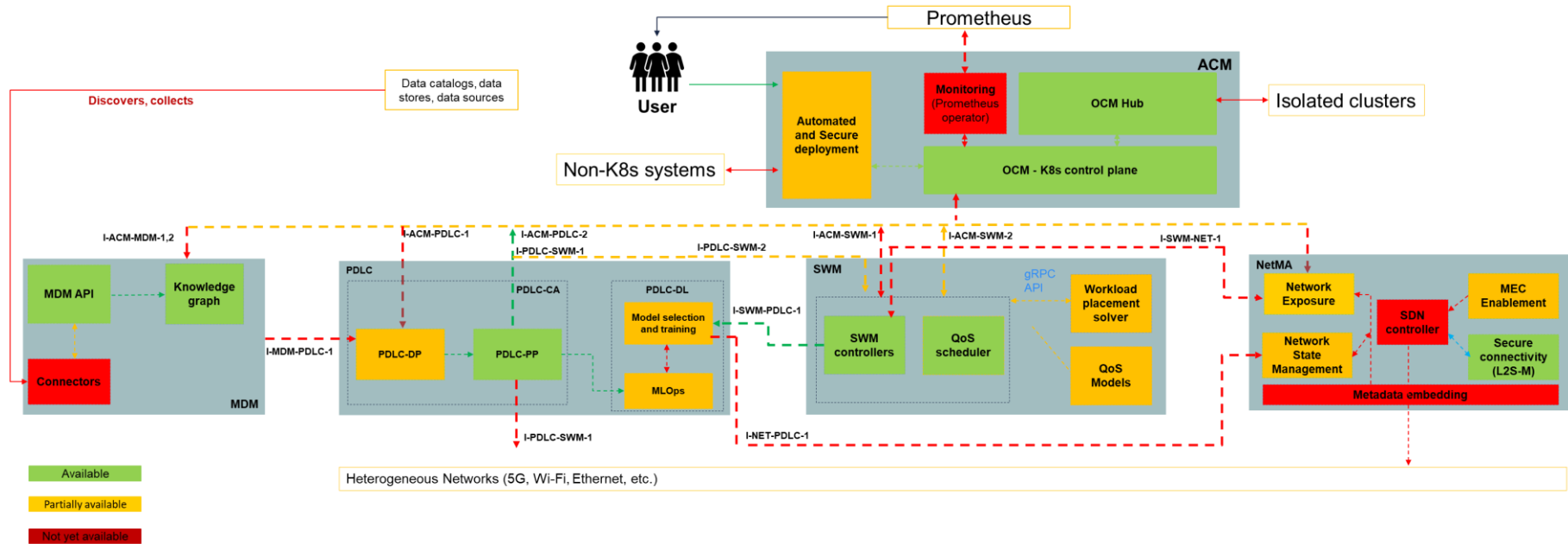


Figure 2. CODECO and its components and interfaces. ACM and SWM reside on the control plane (master node); MDM, PDLC and NetMA reside on the worker nodes (service plane).

2.2 CODECO Workflow Example, Single Cluster

This section has been adapted from D9 and aims at providing the reader with the latest updates on the CODECO operational workflow.

2.2.1 Creating an Application Deployment with CODECO

DEV is a user deploying an application consisting of multiple micro-services (multiple Docker images) across the far Edge to the Cloud. DEV downloads CODECO from the CODECO Eclipse GitLab and follows the instructions to set up ACM. ACM performs cluster sizing based on the CODECO Application Model (YAML file) provided to user DEV during application deployment setup. In this file, user DEV defines aspects such as desired *Quality of Service (QoS)*, *Quality of Experience (QoE)*, or other desired performance levels for CODECO, based on specific questions provided in a future ACM *User Interface (UI)*. The current attributes considered in the CODECO Application Model are provided in Annex I Table 14. Based on the specific parameters (representing the application requirements), ACM builds the CODECO Application Model and makes it available to all CODECO components.

ACM also handles the complete K8s setup (e.g., namespace, databases, secrets) and makes the information available to other K8s components as needed. For example, metadata information, schema, can be passed to MDM (rf. to Figure 2, I-ACM-MDM-2). Application requirements derived from the CODECO Application Model, e.g., dedicated CPU, required bandwidth, are made available to SWM (rf. to Figure 2, I-ACM-SWM-1) and to PDLC (rf. to Figure 2, I-ACM-PDLC-1), for instance.

The exposure of requirements and application/user information also triggers the operation of each CODECO component. PDLC defines the processes for activating the sensing and (decentralised learning) processes for the cluster. SWM makes a request to PDLC to obtain the initial weights to be considered for scheduling optimisation (I-PDLC-SWM-1). NetMA triggers the definition of the network overlay when it receives the initial deployment from SWM (I-SWM-NET-1).

After activation, PDLC regularly obtains, via available *Custom Resources (CR)/Custom Resource Definitions (CRD)* (I-ACM-PDLC-1,2), metadata provided by MDM (data metrics), ACM (user aspects and application constrains, i.e., the CODECO Application Model), NetMA (network metrics); performs an initial multi-level objective estimation based on the user selected target profiles, and stores the output on a PDLC CRD, making it available to ACM, which may trigger adjustments to the initial setup process.

In parallel, three components start monitoring different metrics. NetMA captures network metrics at node, link and path level and exposes this information via specific CRs that are accessible to all components and used by PDLC and ACM.

Similarly, MDM captures data aspects (e.g., data compliance), generates a knowledge graph and also provides the output as an MDM CR.

ACM captures user preferences and eventually behaviour, which may be useful for adapting the overall K8s infrastructure at a later stage.

2.2.2 CODECO Support during Cluster Runtime

Once the setup is complete, CODECO enters the cluster management phase (application runtime support), targeting the user MGR. During this phase, the proposed application (CODECO application workload) has been set up and is running on several containers (1 cluster), 1 or more pods per worker node. PDLC periodically receives data from MDM (I-MDM-PDLC-1); from NetMA and ACM (I-ACM-PDLC-1) and feedback from SWM regarding the



placement of the application workload (I-SWM-PDLC-1). Based on this, PDLC periodically evaluates the proposed system performance targets (e.g., greenness, service latency) provided by Bob during application setup, and provides a cost combination per infrastructure element via a CR (I-PDLC-ACM-2). If there is a need for cross-layer redistribution of the application workload, this step triggers a request from ACM to all CODECO components. In this case, PDLC passes a behaviour estimate to SWM (I-PDLC-SWM-2) via a specific CR; SWM starts the workload placement process. Once the process is complete, SWM passes feedback to PDLC (I-SWM-PDLC-1). This will not be an explicit interface; instead, feedback will be provided via specific SWM CRs (currently ApplicationGroup, Application, AssignmentPlan).

ACM handles the status to the user MGR, based on the Prometheus CODECO monitoring architecture.

3 Specification and Implementation of CODECO Components

This section provides an overview of the initial specification of the CODECO components based on deliverable D9 and derived from the initial nine months of the project.

The aim is to describe the components, their inputs and outputs, their pre-requisites, as well as their functionalities, to support the reader in understanding the current code release.

3.1 ACM: Automated Configuration Management

3.1.1 Component Description

As a quick introduction and a quick overview, the ACM is based on the *Open Cluster Management (OCM)*⁶ community-driven project which is the upstream project for Red Hat *Advanced Cluster Management*⁷. In CODECO, its operation considers three main aspects that address the integration of CODECO across the entire Edge-Cloud infrastructure:

- **Integration points between users and applications.** Mechanisms for users (e.g., user DEV) to control and change configuration of the applications.
- **The CODECO configuration.** A user request during application deployment setup or application runtime implies activation and eventual configuration of CODECO components.
- **Cluster/federated cluster configuration.** The user in this case (e.g., user MGR) handles the K8s infrastructure. A specific change to the CODECO configuration may imply the need to reinstall or reconfigure a cluster.

Figure 3 depicts ACM and its sub-components with interfaces that will need to be implemented, i.e., some technologies will need to be extended to support the planned CODECO components and APIs.

⁶ <https://open-cluster-management.io/>

⁷ <https://www.redhat.com/en/technologies/management/advanced-cluster-management>



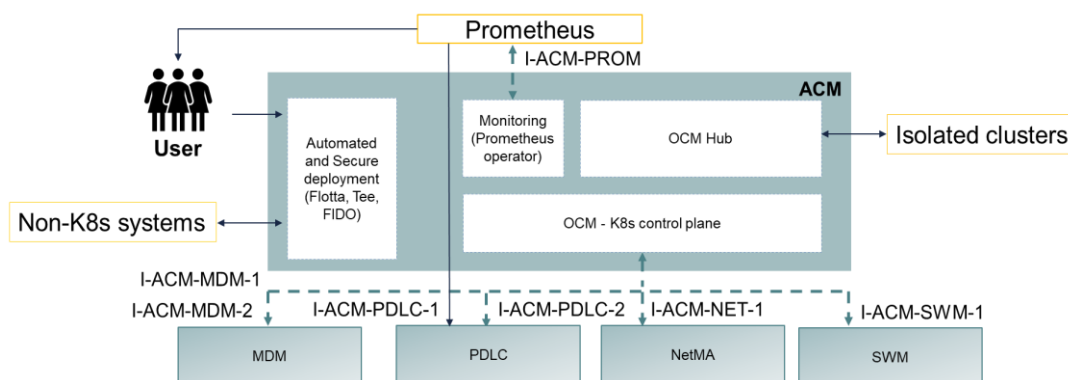


Figure 3. ACM and sub-components: High-level overview with interfaces described in Table 1.

Table 1: ACM interfaces to other CODECO components.

ID	Description	Type	Direction	Component released?
I-ACM-PROM	Integration of the CODECO metrics into Prometheus	CRD	Bi-directional	Not yet available
I-ACM-MDM-1	Data collection from CRDs of other components, to be used in the MDM integration of the CODECO MDM operator.	CRD	Bi-directional	Partially
I-ACM-MDM-2	Integration of additional data types	To be defined	Input	
I-ACM-PDLC-1	Integration of CA derived metrics into Prometheus	CRD	Input	Partially
I-ACM-PDLC-2	Use of CRDs from other components by PDLC	CRD	Output	
I-ACM-NET-1	Integration of the NetMA CRD/CR(s) and respective operator(s)	CRD	Input	Partially
I-ACM-SWM-1	Integration of the SWM controller-manager (CRD/CR(s) via K8s API)	CR via K8s API (REST)	Bi-directional	Not yet available

The ACM initial sub-components are:

- **OCM**: is used to enable end-to-end visibility and control (i.e., control-plane functionality) across K8s-based clusters. OCM will be used to provide the main ACM functionality, and it will be extended to provide support and visibility of the newly added CODECO components (shown at the bottom of Figure 3).
- **K8s API extensibility and CRD/CRs**. As a mechanism provided by K8s to extend its functionality and provide native access to new components, it is going to be used to provide native K8s APIs to the CODECO framework.
- **Monitoring**: Different CODECO components (ACM, NetMA, MDM) collect parameters that will be used to assist in a more flexible schedule, compute, network, and data awareness.
- **Automated configuration via Knative and Ansible**. We will use Knative when we need scalable stateless functions that can be easily scaled-in and -out upon load change and Ansible to support deployment management. This is usually a requirement for many stream processing and event-driven functions.
- **Control plane for independent/isolated clusters** – There are multiple open source technologies handling this problem (e.g., KCP⁸) and CODECO’s choice is yet to be decided. Here, the aim is to consider mobile environments where intermittent connectivity may prevent the registration of a cluster to the OCM Hub.

⁸ <https://github.com/kcp-dev/kcp#-kcp>

- **Far Edge integration**, via lightweight K8s distributions (e.g., K3s or Microshift). CODECO addresses the support of clusters with embedded devices at the far Edge, potentially mobile.
- **Integration towards non-K8s systems.** ACM supports connectivity towards non-k8s nodes via Flotta, to provide the connection of containerized workload to other K8s clusters.
- **Secure deployment.** The secure onboarding of new cyber-physical systems at the Edge is handled via ACM, based on tools such as FIDO and TEE.
- **Hierarchical control plane.** Via Hypershift, ACM supports a separation of the control plane into a centralized cluster (Cloud or Edge) and considering that remote worker nodes are being deployed at the Edge.

3.1.2 Application Model description

As the ACM is an important integration point for the CODECO platform and applications, one of its main tasks is to provide and manage the CODECO Application Model. The CODECO Application Model was defined in CODECO D9 [1] and is a model for the QoS/QoE requirements of an application. A simplified model for the management of the CODECO Application Model by ACM is shown in Figure 4, where it can be seen that an application has some generic application level attributes (such as **name** and **QoS**) and a composition of other supporting micro-services that, together, are used by the application. We use this structure to get input about the application requirements (QoS/QoE), and to report the status of the application to the users. For instance, and as represented in Figure 4, the ApplicationModel provides information for the sizing and locations of the pods (e.g., *avgUsedResources* and *location*) as well. However, there are some dependencies between the micro-services and consequently, between the pods that support the application deployment. Therefore, providing additional information about these relations will be relevant to ensure a successful deployment. A few aspects that may be interesting to consider are, for instance, affinity between pods (e.g., specify the location (node) for a group of pods in the cluster), application requirements (e.g., the co-location of two micro-services is important), or network performance requirements between micro-services. The more detailed the entity model (these are still without full details of all attributes) the better description and the relation details among the micro-services CODECO can have.

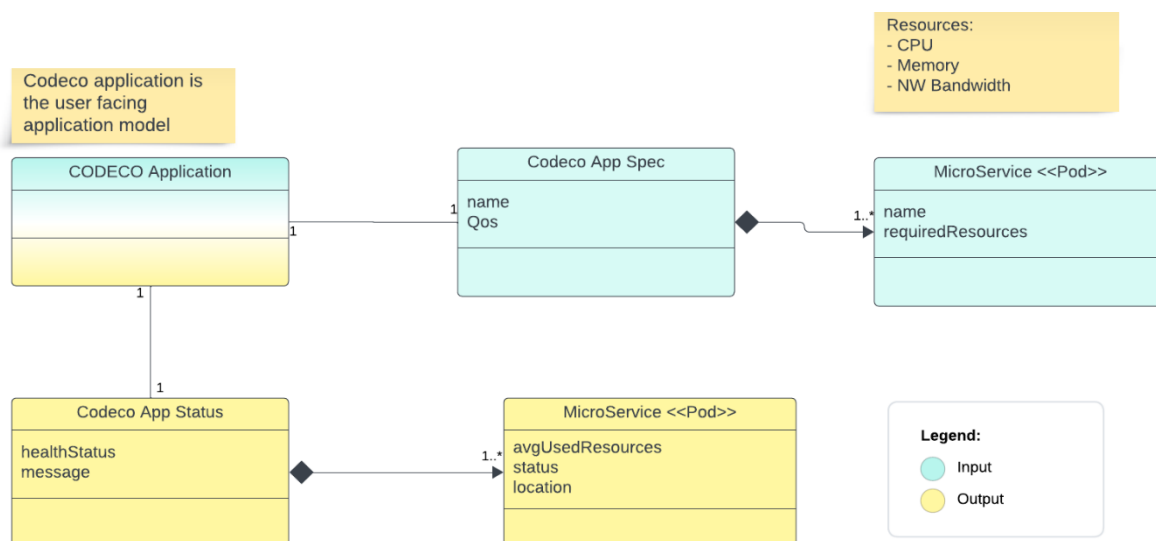


Figure 4: simplified CODECO Application Model representation.

A more detailed view of the proposed CODECO Application Model is provided in in Figure 5.

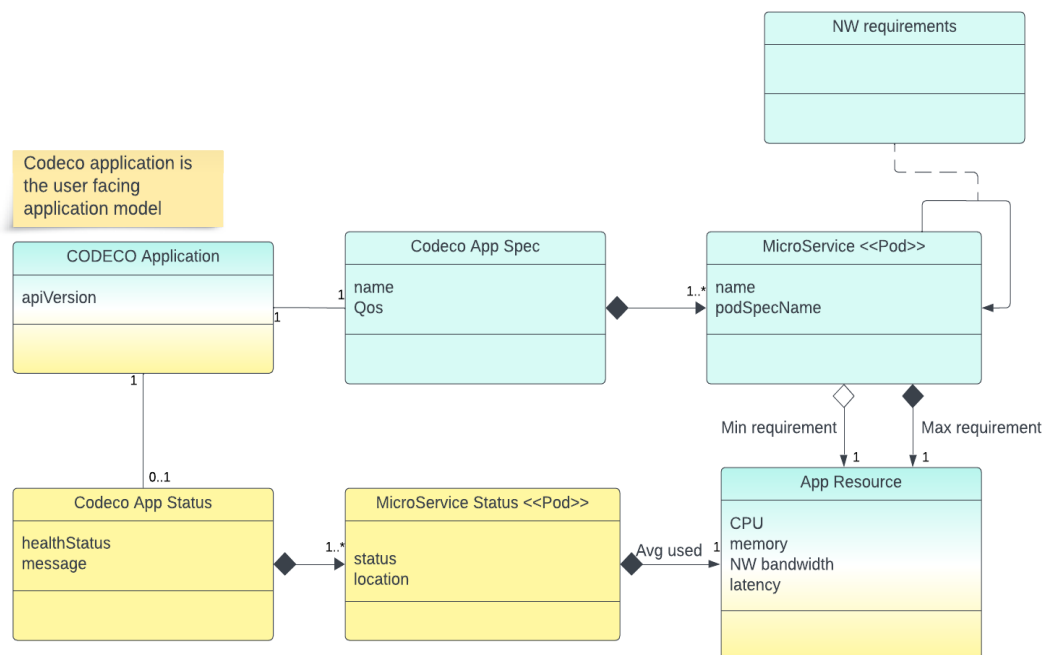


Figure 5. Example for a finer-grained definition of the CODECO Application Model.

This model with additional attributes will be used in the CODECO Application model as the API model. Figure 6 provides an example on how the respective YAML file, based on the translation of the entity model in Figure 5 (without the state, which is output of the system and without the restrictions on the network between micro-services).

```

apiVersion: codeco.he-codeco.eu/v1alpha1
kind: CodecoApp
metadata:
  labels:
    app.kubernetes.io/name: codecoapp2
    app.kubernetes.io/instance: codecoapp-sample
    app.kubernetes.io/part-of: codecoapp-operator
    app.kubernetes.io/managed-by: kustomize
    app.kubernetes.io/created-by: codecoapp-operator
  name: codecoapp-sample
spec:
  name: My CODECO App
  qosclass: Dev
  codecoapp-msspec: [
    {
      name: "CODECO micro service 1",
      podspecname: "MicroService1",
      required-resources: {
        cpu: "2",
    
```

```

    mem: "120G",
    nwbandwidth: "10M"
  }
},
{
  name: "CODECO micro service 2",
  podspecname: "MicroService2",
  required-resources: {
    cpu: "1.5",
    mem: "80G",
    nwbandwidth: "20M"
  }
},
]
    
```

Figure 6: Snippets of a potential structure for the CODECO Application Model.

3.1.3 Sub-components' Specification and Implementation

3.1.3.1 OCM - K8s control plane

3.1.3.1.1 Usage Scenario

Figure 7 shows the two common user scenarios, application deployment (user DEV) and K8s infrastructure (**platform**) configuration (user MGR). These scenarios have a lot in common since application deployment may end up with the need to perform platform configuration, and platform configuration may impact application deployment (e.g., move workloads between clusters). Both scenarios start with CRD changes that are processed by the respective ACM controller and the result is propagated to the clusters across the Edge-Cloud continuum. This diagram hides the CODECO components (PDLC, NetMA, MDM, and SWM) since these components are hidden from the user and their interface starts with ACM. For technical readers, the ACM controller component in the middle stands for the controllers of all the CODECO platform components.

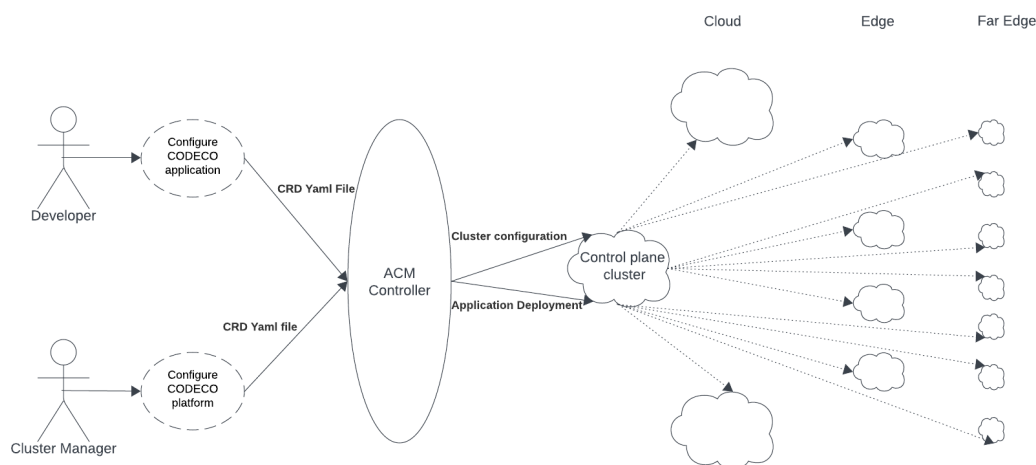


Figure 7. CODECO usage scenarios (via ACM).

3.1.3.1.2 Selected Technologies

- **Multi-cluster management.** Based on OCM (to be developed during the second phase of CODECO, M18-M36).
- **K8s API extensions** via CRD/CRs and respective operators.
- **Prometheus⁹** and exporters such as *Kubernetes-based Efficient Power Level Exporter*, *Kepler¹⁰* as well as new Prometheus operators to support new monitoring plugins from the different CODECO components, supporting the integration of new K8s metrics.
- **Knative¹¹** to support serverless deployment of containerized applications.
- **K3s¹², Microshift¹³** Lightweight K8s distributions.
- **Flotta¹⁴.** Integration with non-K8s far Edge nodes.
- **Fast Identity Online (FIDO) or Trusted Execution Environments (TEE).** Secure onboarding of devices.
- **Hypershift¹⁵.** Fast provisioning of K8s clusters on remote worker nodes.
- **Ansible¹⁶** to support the automated deployment of CODECO.

3.1.3.1.3 Pre-requisites

In terms of hardware or software, ACM will require a lab environment with at least a cluster installed, e.g., a K8s-based cluster. This cluster needs to be dimensioned to have enough resources to accommodate the ACM component installation, which will also include the whole CODECO platform with all its sub-components. There are no additional software or hardware necessary if not those already specified for CODECO, since ACM is the main integration point to enable the CODECO platform and its features.

3.1.3.1.4 Installation Guide

The ACM is the integration point towards the user, it is also in charge of the system installation - the user can install the whole CODECO framework by installing the CODECO meta-operator (which is part of the ACM), where this process triggers the installation of the whole CODECO system on the cluster to be managed and configured. See Figure 7 to visualize the interaction of different user types with CODECO. For example, user MGR can deploy (or configure, as illustrated) the CODECO platform.

Currently the installation is CLI-based only and will support the installation of all the components in one command. In the future we plan to use the Operator Hub model (a kind of app store for operators) which allows an operator to install multiple components according to the dependencies and provides a GUI for the installation.

The [installation steps available in the CODECO Gitlab](#) are summarized as follows:

- A container image of the operator should be built and pushed into an image repository (such as docker hub, quay.io and others) – this is a prerequisite and typically will be done in a different timeline than the installation. The [ACM operator is available in Docker Hub](#).
- Add the namespace “**codecoapp-operator-system**” to the cluster.
- Install the CRD “**codecoapps.codeco.he-codeco.eu**” on the cluster.
- Install all the roles and role bindings required for the operator to work – two important roles are:

⁹ <https://prometheus.io/>

¹⁰ <https://sustainable-computing.io/>

¹¹ <https://knative.dev>

¹² <https://k3s.io/>

¹³ <https://github.com/openshift/microshift>

¹⁴ <https://project-flotta.io/>

¹⁵ <https://github.com/openshift/hypershift>

¹⁶ <https://www.ansible.com/>



- codecoapp-editor-role – Used for users that can read and edit the spec part of the CRD and read the status part.
- codecoapp-operator-role – Used by the operator itself and can read and edit both the spec part and the status part.
- Deploy/modify the controller manager (a generic k8s component) with information about the CODECO operator.
- Deploy the controller image (the container) on the cluster.
- Installs the other CODECO components (SWM, NetMA, MDM, PDLC) according to their installation scripts.

3.1.3.1.5 Inputs & Outputs

ACM gets as input:

- The CODECO Application Model.
- Input from other CODECO components (rf. to Figure 6).

ACM has as output:

- The ACM UI, where state concerning the “status” section of the Application Model will be provided.
- Platform resource status (via Prometheus).
- Information about application status (e.g., runtime failures derived from the K8s selected infrastructure)

The ACM exposed APIs (the CODECO APIs) are declarative APIs following the K8s methodology. The CODECO Application Model described above is used for both input and for output of the application APIs – for input it represents the desired state of the application, and for the output it will also include some information about resource usage (e.g., average CPU usage, memory usage and energy consumption in a specific time window), as well as the current application configuration and deployment details.

3.1.4 Next Cycle features

The following expected features are presented in order of priority, from most to least:

- A full definition of the CODECO Application Model, its input and output, the overall logic of using the model in CODECO in interaction with other components.
- The completion of all sub-components missing, namely, the integration of ACM with the CODECO monitoring architecture (Prometheus) and the completion of the sub-component automated and secure deployment.
- The integration of the CODECO monitoring architecture (Prometheus operator)
- Interfaces to the other components. We expect to start with the interfacing to-from MDM and to-from SWM.

3.2 PDLC: Privacy-preserving Decentralised Learning and Context-awareness

3.2.1 Component Description

PDLC as the heart of the CODECO cognitive orchestration is shown in Figure 8, and consists of two sub-components: PDLC-CA (Context-awareness) and PDLC-DL (privacy preserving Decentralised Learning).



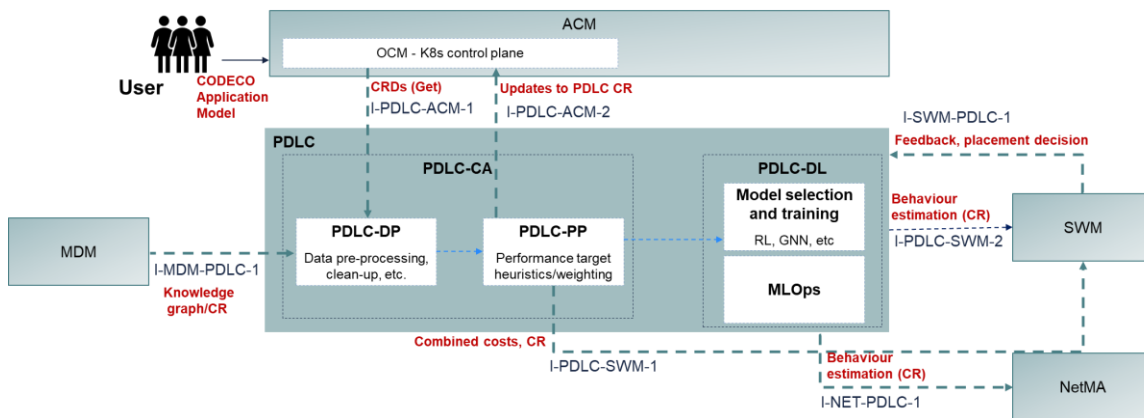


Figure 8. PDLC high-level architecture and interfaces.

The **PDLC-CA** sub-component is responsible for generating actionable context information and performance profiling based on the performance goal(s) selected by the user DEV during application deployment setup, based on cross-layer data provided by other components (NetMA, MDM, ACM). Its output is currently provided in the form of a node ranking associated with the desired performance optimisation (e.g. desired greenness or resilience). The generated output is provided in a CRD format that can be used as input to other PDLC subcomponents and other CODECO components. It is currently envisaged that the output of this sub-component will be used by the SWM.

The subcomponent **PDLC-DL** goes a step further in adding intelligence to the CODECO framework, using the raw data collected by PDLC-CA to train decentralised privacy preserving learning models to provide forecasts and predictions about the future behaviour of infrastructure nodes and deployed applications. In the future, it will supplement this raw data with the performance profiles generated by PDLC-CA as input to the models it provides. PDLC-DL will also develop MLOps pipelines that select the most appropriate model for the different objectives highlighted by the DEV user in the deployment setup.

The PDLC sub-components provided in the CODECO GitLab public repository and their interactions with other PDLC components are shown in Figure 9. Given that the subcomponents are not based on pre-existing background software, their full integration will be finalized in future releases of the CODECO framework. This will include communication between PDLC-CA and PDLC-DL as well as the integration of the PDLC-DL models with the developed MLOps pipeline, which is provided as an independent sub-component with the current release.

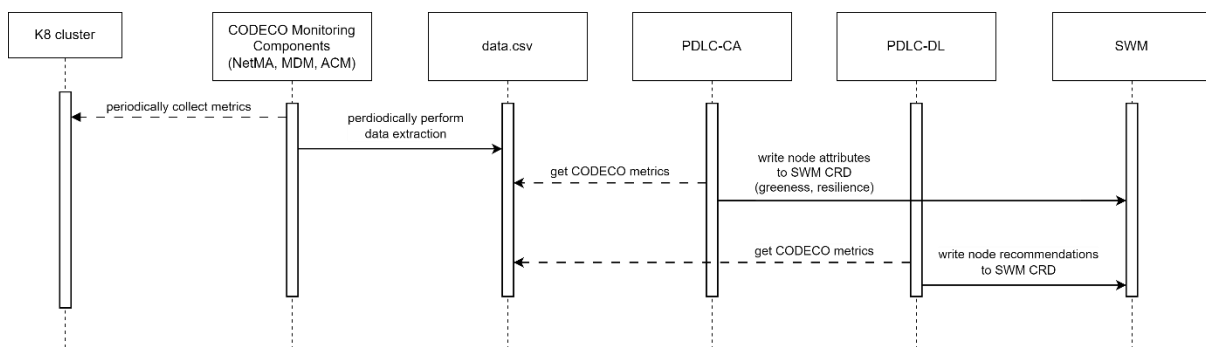


Figure 9. PDLC sub-components and their interactions with other CODECO components.

On the other hand, PDLC interfaces to other components are shown in Table 2, where we also detail the status of implementation in this early release of the CODECO open source Basic Operation toolkit.

Table 2: Status of PDLC interfaces to other CODECO components.

ID	Description	Type	Direction	Status
I-PDLC-ACM-1	Obtaining data collected by other CODECO components and made available via CRs of those components.	CRs	Input	Partially implemented
I-PDLC-ACM-2	Output of the combined infrastructure view, via a PDLC CR	CR	Output	Available
I-MDM-PDLC-1	Data collected by CODECO MDM	Knowledge graph, e.g., graph DB	Input	Not available
I-PDLC-SWM-1,2	PDLC will set attributes of custom resources of the infrastructure model, which are used by SWM to influence the placement decision for workloads. This can relate to combined costs (I-PDLC-SWM-1) or to behaviour estimation (I-PDLC-SWM-2). Currently, the relevant custom resources are <i>Endpoint</i> , <i>NetworkLink</i> , <i>NetworkNode</i> , <i>NetworkPath</i> . PDLC can also make use of the QoS Scheduler extensions.	CR	Output	Available
I-SWM-PDLC-1	PDLC will be able to know about scheduling/placement decisions of SWM by accessing the <i>AssignmentPlan</i> CR	CR	Input	Available
I-NET-PDLC-1	NetMA asks infrastructure element estimation from PDLC-DL	CR	Output	Not available

3.2.2 Sub-components' Specification and Implementation

3.2.2.1 PDLC-CA

The current version of PDLC-CA is focused on single cluster operation. It relies on inputs from other CODECO components (e.g., computational parameters such as CPU, memory; network parameters such as bandwidth, latency, link energy; data metrics such as data compliance, data size) and from Prometheus. The existing parameters are combined to serve a selected performance profile, such as optimising the Kubernetes infrastructure for greenness or resilience. The performance profile, which is currently defined statically in the code, will be an attribute of the CODECO application model selected by the user DEV during the deployment of an application.

Currently, PDLC-CA relies on simple heuristics to rank existing nodes based on the selected target profile and considering a combination of available data. The output of PDLC-CA is made available in the form of a CRD to the distributed learning subcomponent of PDLC, and also to other components of CODECO, such as SWM. PDLC-CA interacts with PDLC-DL and provides the performance profile node costs to SWM.

3.2.2.1.1 Usage Scenario

For this early release, PDLC-CA [integrates the sub-components PDLC-PP \(performance profile cost computation\) and PDLC-DP \(data processing\) as a whole](#). The proposed usage



considers the application of PDLC-CA on a single cluster which can be set up with any available tool. The current code provides the possibility to experiment PDLC-CA with minikube or with KinD, with a variable number of nodes, and provides an example for two specific target profiles: greenness and resilience, providing directions on how a developer can create new heuristics for other potential performance profiles.

The main purpose of this early release version is to show how to build aggregate node costs based on node, network, data attributes obtained from accessible CODECO CRDs.

PDLC-CA has been set to consider as input the CODECO synthetic data generator (rf. to 4.1.1). The user can therefore set different clusters and test PDLC-CA in standalone mode; and then use the generated output (CRD and csv format) to interact with other CODECO components. PDLC-DL can also use the generated CRD as input. The files are presented in Table 3.

Table 3: PDLC-CA code summary.

File	Function
apply_controllers.sh	Installs PDLC-CA in the cluster
delete_controllers.sh	Removes PDLC-CA from the cluster
extractor.py	Data processor, used by combiner.py. Gets CRDs and creates a single data.csv with the input values (scalar) to PDLC-CA
combiner.py	Used to activate data processing and generate the costs (calls extractor.py and ca_component.py)
ca_component.py	has as input the CODECO Application Model performance_profile and generates as output an aggregated value for that profile, for each node, based on CODECO NetMA, MDM, ACM parameters being monitored.
ca.csv	Output of PDLC-CA, generated by crd_data_extraction.py
ca_controller/	PDLC-CA controller and respective yaml files.
crd_data_extraction/crd_data_extraction.py	Creates the output in a csv format, by reading the CA CRD.

It is currently possible to test two specific target profiles, which are provided as potential examples on how aggregated node costs based on a performance profile can be implemented. The efficiency and performance of the heuristics have not yet been validated.

Greenness Profile (greenness(n)), focuses on helping to create an infrastructure with the lowest possible energy consumption.

- **Purpose:** to select nodes that can contribute to achieve an infrastructure (for an application group) that has the least energy expenditure possible.
- **Rationale:** the lower the node energy expenditure; the higher the available bandwidth; the higher is the node greenness.
- **Example:** $greenness(n) = ((1 - node_energy(n)) / node_energy(n)) * available_bandwidth(n)$, where n is a suitable node and $available_bandwidth(n)$ is the sum of available bandwidth on egress links.

The parameters for the node are provided via ACM (Prometheus, node_energy); the available_bandwidth (ibw and ebw) will be provided by the CODECO component NetMA.

Resilience Profile (resilience(n)), is focused on the selection of nodes that will maximise the resilience of the infrastructure.

- **Purpose:** Nodes are weighted for their contribution to resilience of the selected infrastructure
- **Rationale:** the higher the number of link failures associated with the node; the smaller the node_degree; the lower is the resilience of the node
- **Example:** $resilience(n) = node_net_failure * 1 / node_degree * sum_link_failure$

The parameters are provided by ACM (node_degree) and NetMA (node_net_failure, sum_link_failure).

3.2.2.1.2 Selected Technologies

PDLC-CA has been implemented in Python v3.0. Input and output are based on YAML (K8s CRD).

3.2.2.1.3 Pre-requisites

Rf. to the [README of the](#) PDLC-CA code for a detailed list of requirements:

- A cluster needs to be set up. In our case, we provide how-tos that can be used with KinD or minikube.
- The CODECO data generator needs to be installed and run on the cluster to generate a **data.csv** (input) with the correct format.

3.2.2.1.4 Installation Guide

A [detailed installation guide is available with the code of PDLC-CA](#).

1. Get the PDLC-CA code.

```
git clone git@gitlab.eclipse.org:eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/context-awareness/pdlc-pp.git
```

2. Set up a Kubernetes cluster.

In this release, we provide examples on how to set a cluster with KinD or minikube.

With Kind:

1. Create kind config file:

```
cat > kind-config.yaml <<EOF
# three node (two workers) cluster config
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
EOF
```

2. Create cluster:

```
kind create cluster --name <cluster_name> --config kind-config.yaml
```



With minikube:

Set up a cluster with 1 node (control-plane):

```
minikube start
```

(or) Set up a cluster with x nodes:

```
minikube start --nodes x -p <cluster-name>
```

3. Get data from your cluster.

PDLC-CA will use data collected by the CODECO components NetMA (network metrics); MDM (data workflow metrics); ACM (application and user metrics), by running the CODECO synthetic data generator on your cluster (<https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/experimentation-framework-and-demonstrations/data-generators-and-datasets/synthetic-data-generator>) to be able to retrieve metrics that is used as inputs to the PDLC-CA component.

4. Deploy the PDLC-CA controllers in your cluster.

```
chmod -R 777 .  
./apply-controller.sh
```

5. Generate aggregated costs based on a given performance profile.

Based on a cluster monitored data, the PDLC-CA component creates a CRD and also a `ca.csv` file with the computed node costs. For this, as input the following options are currently available. To check the node cost metric extracted from the CA CRD, run the following commands on the '`crd ca_crd_extractor`' directory:

```
pip3 install -r requirements_e.txt  
python3 ca_crd_extractor.py
```

The python script runs continuously to fetch the most recent updates to the CRD, until it is forced to stop by a keyboard interrupt using CTRL + C. **ca.csv** then provides, for visualization, the node(s) costs for the specific performance profile(s). The `ca.yml` holds the respective costs as well.

3.2.2.1.5 Inputs & Outputs

- **Input of PDLC-CA corresponds to CRDs provided by other CODECO components.** PDLC-CA transforms that input (CRD format) into a csv format (**data.csv**). Currently, you can play with the provided `data.csv` (single cluster with one node) or with a `data.csv` obtained with the CODECO data generator for your cluster (rf. to instructions on the readme or in Section 3.2.2.1.4).
- **Output of PDLC-CA** is provided in a CRD format and also, for the purpose of testing and visualization, in the [ca.csv](#) file.



3.2.2.1.6 UML Diagram

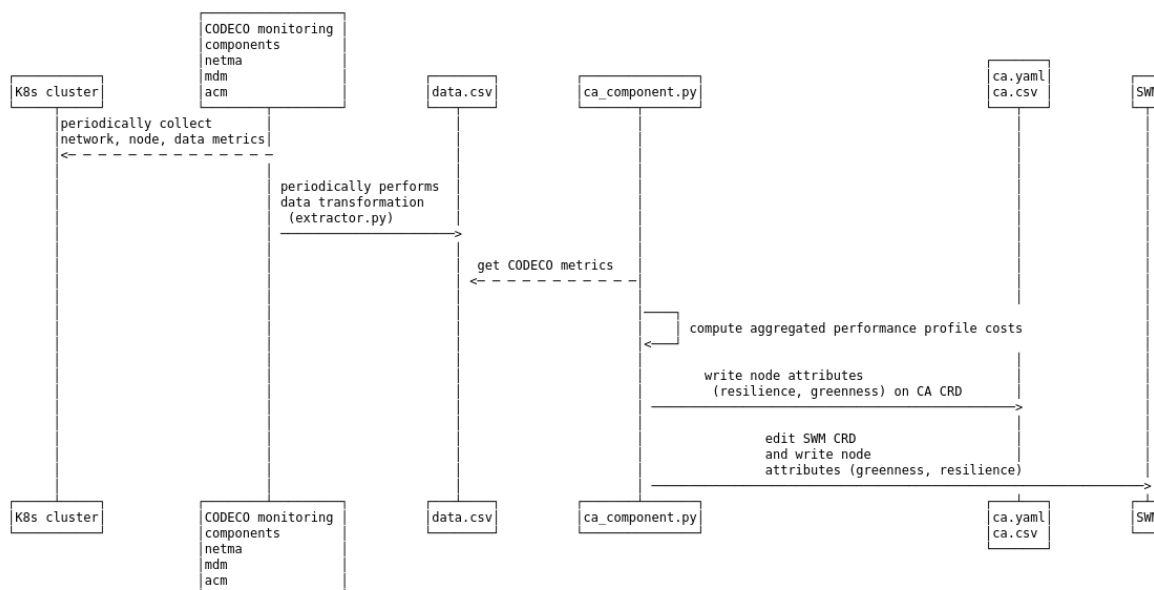


Figure 10. UML Sequence diagram of PDLC-CA.

3.2.2.2 PDLC-DL (Model Selection and Training)

In this release of the CODECO toolkit, the [PDLC-DL sub-component consists of three independent models that perform different objectives](#):

1. **Reinforcement Learning Model:** provides node recommendations to allocate applications with the objective of balancing CPU and RAM usage amongst all available cluster nodes. Detailed in Section 3.2.2.2.1
2. **Graph Neural Network Models:** provide predictions for the values of the monitored metrics of a cluster’s nodes, currently CPU and RAM. Detailed in Section 3.2.2.2.2.
3. **Model-based Multi-agent Reinforcement Learning Models:** provide a decentralized environment that enables multiple agents to interact and learn from their experiences. Detailed in Section 3.2.2.2.3.

The integration, interaction and selection of these models will be handled by the MLOps pipeline described in Section 3.2.2.4, in a future release of CODECO.

3.2.2.2.1 Reinforcement Learning Models

3.2.2.2.1.1 Usage Scenario

This usage scenario refers to a fully functional training environment that provides the user with an implementation of the CODECO infrastructure as a gymnasium environment, which includes a *Reinforcement Learning (RL)* agent that focuses on balancing CPU and RAM usage amongst all available cluster nodes and keeping the number of pods without being allocated at a minimum. Furthermore, we provide the user with the ability to train and monitor this agent with two *RL* algorithms (*Deep Queue Learning (DQN)* and *Proximal Policy Optimization (PPO)*). The model has been developed with a clear objective in mind, that being the ability to easily expand the model in the future with PDLC-CA parameters and make it use-case adaptative. Furthermore, it serves as a foundation for the future expansion to a multi-agent implementation.

Additionally, Table 4 provides a summarized description of the python files for the PDLC-DL model analysis performed by partner I2CAT ([I2CAT_RL-model](#) repository).

Table 4: Scripts included in I2CAT contribution.

File	Function
requirements.txt	Contains all the requirements for the program
codeco_env.py	Gymnasium environment used in RL algorithms as the problem representation.
data_generators.py	Script that generates the node information and prepares the data from UPRC.
data_preprocess.py	Contains useful data preprocessing functions.
K8node_config.py	Contains all the code related to the custom K8 node representation.
main.py	Main of the program, program launches from here.
training_controller.py	Contains all code related to the training and inference of the RL models.
config_train.ini	Contains all the configuration needed for the training of the RL model.

3.2.2.2.1.2 Selected Technologies

The initial PDLC-DL modelling and experimentation has been implemented in Python 3.10.9 using Gymnasium¹⁷ and Stable-baselines3 as key libraries. The full list of libraries can be found in [the requirements file](#) that can be found with the code. The selected AI technique for the first approach is a Deep RL model that focuses on distributing evenly the resource consumption among nodes and to allocate all pods possible.

The programmed components are described next. It is worth remarking that this first approach has been built with easy expansion in mind and as a foundation for the next steps in PDLC.

Creation of the Reinforcement Learning environment:

The CODECO Environment is created inheriting from the Gymnasium base Env, which has been used to model the proposed RL problem and to, after creating an agent, to make it learn to solve the modelled problem in the environment. To do this, the following aspects of the problem were modelled (as functions Inherited from the base environment):

- Init: Function called when creating an environment, it initializes the state, the observation and action space and all the auxiliar variables needed.
- Step: Function that simulates a step in the environment after using an action in it. It returns a reward and the state of the system after being affected by the previous action.
- Reset: Function that resets the environment to its original state to restart training on it.

These are the key functions that need to be reimplemented. Some functions have been created inside the environment to compute the workload of each node, update the resource allocation, update the state, or compute the reward.

Modelling the problem as a Reinforcement Learning model:

Objective:

For this proof-of-concept, the selected performance target was to balance the load of all working nodes in a K8s cluster, while minimizing the total number of pods that are not allocated due to resource limitations. The importance that is given to these two objectives can be balanced with two weights that control the computation of the workload (see reward).

¹⁷ <https://gymnasium.farama.org/index.html>

Environment (observation space):

The environment in this first approach is represented as the set of nodes and pods that form a K8s cluster, and the resources that they are consuming, also, the resource request from the pods are inside the observation space. The observation space has been proposed as the resources requested by the allocation request (currently CPU and RAM), which, in the future, will be derived from the CODECO Application Model, and the current node usage in terms of CPU, RAM.

State:

As per the state, it is represented it as a snapshot of the system status in a given timestamp where K8s can check the resource usage for all nodes and pods, in this timestamp, the CPU and RAM usage for the pod that will be allocated and the current usage of the nodes in the cluster are gathered. This can be represented as follows:

$$S_t = \{(p_i, n_j) \vee 0 \leq j < n.nodes\} \text{ where:}$$

$$p_i = \{(cpu_i, ram_i)\},$$

$$n_j = \{(cpu_j, ram_j) \vee 0 \leq j < n.nodes\},$$

$$i, j \in \mathbb{Z}$$

Action space:

Each time an allocation request is received (in this case, once per allocation request and timestamp) the pod is allocated to a new node, or it is kept the node that was previously allocated (if there was a previous allocation). The concepts of being on hold or waiting to be executed have been introduced. To represent a node that does not have an allocation, in the model a 'Fake' node that can hold potentially all the pods in the system if they cannot be allocated, the fake node is not considered when the workload is computed (see reward).

Reward:

To model the reward function, we focused on making a fully scalable, easily expandable reward function that can be adapted depending on the use case that is being used while training the model. To do so the function has been modelled by multiplying some metrics depending on the given importance for the current use case. The current proof-of-concept considers the modelling of two metrics. The idea is that metrics coming from PDLC-CA could be included in this computation depending on the use case:

- **Workload:** this custom metric for PDLC-DL measures how stressed a K8s node is comparison to other nodes. This metric works for comparisons between different K8s nodes with different parameters and works as follows:
- If the resource allocation is correct, the workload for a node i includes its usage of CPU and RAM, as it works by comparing percentages of use between parameters this can be easily expandable to any capabilities that can be measured in % of use compared to its maximum value. The formula to compute this metric is:

$$W_i = (Max_{CPU_i} / Used_{CPU_i} * 100 * w_c) +$$

$$(Max_{RAM_i} / Used_{RAM_i} * 100 * w_r)$$

$$\text{where: } W_c + W_r = 1$$

- So, the minimum value for this metric is 0 and the maximum is 100, the metric represents the % of usage of all the resources in that node, giving importance to each parameter depending on configurable weights. Currently, the function only includes weights for the CPU and RAM consumption, the idea is to expand this further in the future. Furthermore, depending on the data there are cases in which a parameter may be harder to balance than others, as allocating a pod implies reserving all the resources it requires in a node.



Once more parameters are included in the implementation, parameter weight will need to be studied and optimized thoroughly per use case.

- `Non_allocated_pods`: this is a simple percentage that represents how many pods are not currently allocated due to resource constraints.

Once the metrics are defined, the reward function can be defined as follows, where *Workload* is the list containing the workload for all nodes:

$$Reward = - (std(Workload)) + (n_pods - non_allocated_pods)$$

Some parts of the reward are put in negative value as the algorithm will focus on maximizing its value, and the objective is to minimize metrics in our system. In this case, the number of allocated pods is also added as a positive reward to encourage pods being allocated as much as possible.

3.2.2.2.1.3 Pre-requisites

- Input based on csv – the CODECO synthetic data generator needs to be installed and used for this purpose.
- The model is adapted to work both with CPU and CUDA (GPU). It is highly recommended that for larger datasets, a device compatible with CUDA is used in order to boost the training times.

Apart from the required data, the testing environment should follow the following requirements, some are mandatory, others are a recommendation:

- Having Python 3.10.2 or latest installed.
- It is recommended that the device has a GPU in order to accelerate the training performance, the program is adapted to run on CPU, but if you do so, expect high training times when using a large number of steps.
- Some software to open .pdf files (results are stored in them).
- A code editor (e.g., VSCode) is highly recommended to change the behaviour of some parts of the code if desired.
- A minimum of 8 GB of RAM and a good CPU are highly recommended if a GPU is not available.
- If the program is installed using Docker, it is mandatory to have Docker installed.

3.2.2.2.1.4 Installation Guide

To install and set up the program, [follow the steps provided in the respective git repository](#), which are summarized next.

1. Clone the repository

```
git clone git@gitlab.eclipse.org:eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/decentralised-learning/model-selection-and-training.git
```

2. Create the testing environment

For this there are two options:

- Install the program using Docker image and the [docker compose file](#):

```
docker-compose up OR
```

- Create a python virtual environment, then install the [requirements](#) with:



```
pip install -r requirements.txt
```

- Use the python program from console:

```
python main.py
```

Examples on how to use the program (if installed with docker, the command can be specified in the docker-compose file provided) are provided in Table 5.

Table 5: Examples on how to use RL code.

Use case	Training	Inference
Normal use	Python main.py --use_mode=training	Python main.py --use_mode=inference
Plot results	Python main.py --use_mode=training --plot_results=True	Python main.py --use_mode=inference --plot_results=True
Generate new data and/or node config	Python data_generators.py	Python data_generators.py
Specify where to save model (not default)	Python main.py -- use_mode =training save_models_path="example"	Does not apply, no model saved

3.2.2.2.1.5 Inputs & Outputs

This version of the implementation requires the following input data to work (already provided but can be modified):

- [.csv file](#) that contains all the node configuration information, this file can be created with the script data_generators.py.
- [.csv file](#) with the data pre-processed, you can find two examples already in the folder data.
- [config.ini](#) file that includes the configuration of some aspects of the model and training.
- [Output CRD](#), where the proposed allocations will be written.

Data pre-processing

In a future version, the data processing in PDLC will be done by the sub-component PDLC-DP. For the current version, the following pre-processing has been done:

- Convert the K8s CPU and RAM units to more concise and easier to read ones:
 - CPU has been converted to relative usage (percentage of CPU where the usage is divided by 100, expressing the number of CPU cores being used, example= 0.5 cores or 3.4 cores). The conversion is based on the official [k8s documentation](#).
 - RAM has been converted to MBs.
- The node usage monitoring data is not considered in this release, as the goal is to give predictions of pod allocations, this adapts the input data to the current use case of this first release.
- To give more flexibility, a script that generates random input node configurations (RAM, CPU capacity) is provided. By doing so, we guarantee the portability of the implemented model that can be tested with clusters of different node configurations and topologies.

Outputs:

The current release provides two outputs:

- Results visualizations with plots of the agent performance (training or inference) along with a visualization of a simple greedy algorithm for comparison. The directory of these



plots is created automatically by the program. This output is optional and can be activated using the corresponding command line argument defined in the [readme file](#).

- Allocation proposals in the CRD file provided as input. This is done through an attribute added to [the SWM CRD](#) that indicates how likely a pod should be allocated to a node. For that purpose, a vector of size number of nodes for each pod that needs allocation is used, and for each node the degree of certainty of the pod being allocated to it is given.

3.2.2.2.1.6 UML Diagram

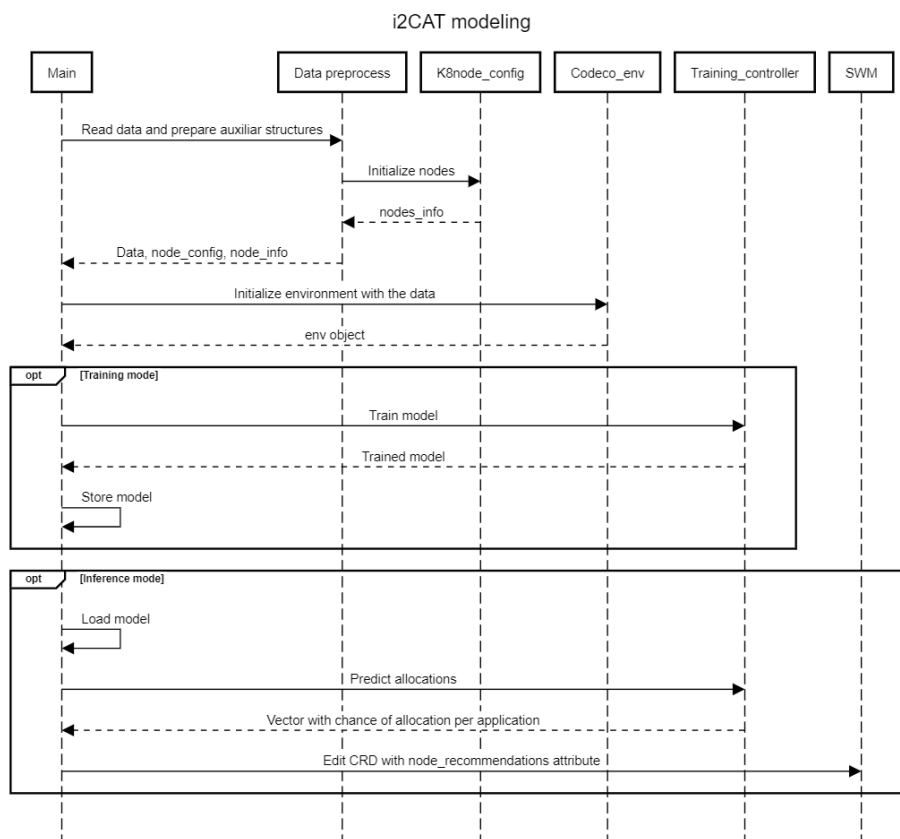


Figure 11. PDLC UML sequence diagram of interactions between the RL model classes.

3.2.2.2.2 Graph Neural Network (GNN) Models

3.2.2.2.2.1 Usage Scenario

The current proof-of-concept integrates testing done with two *Graph Neural Network (GNN)* models within the PDLC-DL subcomponent: a *Spatio-Temporal Graph Neural Network (STGNN)* [3] and an *Attention-Temporal Graph Convolution Network (A3T-GCN)* [4]. The code can be found in the [ICOM-GNN repository](#).

The two GNN models can provide predictions for the monitored metrics of a cluster’s nodes. Both models take as input historical timeseries data of each node (e.g., CPU or memory usage), as well as information about the cluster’s topology, so that they can consider the spatial, as well as the temporal dependencies of the nodes and provide predictions about the aforementioned parameters in future timesteps. It is envisioned that these predictions will be fed as input features to the RL models of PDLC-DL to enhance their performance and help them provide an improved pod allocation plan to SWM. Additionally, the forecasted parameters can be given as input to the ACM component, in order to provide insights about

the nodes' future resource usage and allow it to make more informed decisions and trigger adjustments to the initial setup.

The key purpose of this approach is therefore to provide a forecasting on the usage of the K8s infrastructure, to further assist the workload placement.

3.2.2.2.2 Selected Technologies

- **Spatio-Temporal Graph Neural Network:**
 - NumPy (Numerical Python)¹⁸: an open source, widely used Python library that provides computation functionalities and fast operations on multidimensional array objects, mostly used in the model's data preprocessing code.
 - Pandas (Python Data Analysis)¹⁹: an open source Python library that uses DataFrame and Series data structures for efficient and flexible data analysis and manipulation, mostly used in the model's data preprocessing code.
 - TensorFlow²⁰: an open source framework that provides tools and libraries for the development of machine learning models, used in the development of the model's architecture.
- **Attention-Temporal Graph Convolution Network:**
 - NumPy
 - Pandas
 - PyTorch²¹: an open source optimized tensor library for deep learning in Python, used for the development of the model's architecture.

The **Spatio-Temporal Graph Neural Network** can predict future values of nodes' metrics based on historical observations, by modelling both spatial and temporal dependencies among nodes. The nodes' topology is mapped into a Graph structure and the model consists of a Graph Convolution Layer and a Recurrent Neural Network layer. The Graph Convolution Layer applies graph convolution to the input to get the nodes' representations over time, so that for each time step, a node's representation is informed by its neighbours' representations. The Graph Convolution is calculated as:

$$h_i^{(l+1)} = \sigma \left(b^{(l)} + \sum_{j \in N(i)} \frac{1}{c_{ij}} h_j^{(l)} W^{(l)} \right)$$

where $N(i)$ is the set of neighbors of node i , c_{ij} is the product of the square root of node degrees (i.e., $c_{ij} = \sqrt{|N(i)|} \sqrt{|N(j)|}$), and σ is an activation function.

The nodes' representations are computed by multiplying the input features by the node's own weight and then each node's updated value is calculated by aggregating the neighbors' representations and then multiplying the results by the node's weight. The output of the layer is computed by combining the nodes representations. Based on the input, the graph convolution layer produces new tensor that captures the representations of nodes over time. To process the nodes' representations over time, a Recurrent Neural Network layer is utilized, in this case a *Long Short-Term Memory (LSTM)* layer²².

Regarding **A3T-GCN**, this model is an extension of the *Temporal Graph Convolutional Network (T-GCN)* model [3] and additionally uses an attention mechanism. T-GCN uses a GCN for the spatial aggregation, in order to capture the topological structure of the data and

18 <https://numpy.org/>

19 <https://pandas.pydata.org/>

20 <https://www.tensorflow.org/>

21 <https://pytorch.org/>

22 <https://docs.dgl.ai/en/0.8.x/generated/dgl.nn.tensorflow.conv.GraphConv.html> and <https://github.com/keras-team/keras-io/>



a *Gated Recurrent Unit (GRU)*, in order to capture the temporal features using the time series with spatial features. The T-GCN model takes as input n historical time series data to obtain n hidden states (h) that cover spatiotemporal information: $\{h(t-n), \dots, h(t-1), h(t)\}$.

$h(t)$ is calculated as $h_t = u_t * h_{(t-1)} + (1 - u_t) * c_t$, where u_t at time t is the update gate, meaning the factor that controls the degree to which the previous status is brought into the current status and c_t is the memory content stored at time t . u_t is defined as $u_t = \sigma(W_u[f(A, X_t), h_t - 1] + b_u)$, where $f(A, X_t)$ represents the graph convolution process, with A being the adjacency matrix (graph) and X_t being the input features at time t , and W, b represent the weights and deviations of the training process respectively. Additionally, c_t is calculated as $c_t = \tanh(W_c[f(A, X_t), (r_t * h_t - 1)] + b_c)$, where r_t corresponds to the reset gate, which is used to control the degree of ignoring the previous status, and is calculated as $r_t = \sigma(W_r[f(A, X_t), h_t - 1] + b_r)$.

Moreover, an attention mechanism is utilized, in order to re-weight the influence of historical values and to capture the data variation trends. The hidden states are given as input to the attention model and the weight $a_{(t-n)}, \dots, a_{(t-1)}, a_t$ of each hidden state is calculated by the softmax function, using a multilayer perceptron. A weighted sum is used to calculate the context vector that captures the variation information and for the output, the forecasting results go through a fully connected layer²³ [4].

3.2.2.2.3 Pre-requisites

- In terms of hardware both GNN models within PDLC-DL can support either a CPU or GPU, if available, for improved performance.
- In terms of data the [CODECO synthetic data generator](#) is required and one has to run its data extractor module.
- A cluster needs to be set up.
- In terms of software the pre-requisites are listed below for each model separately:
 - Spatio-Temporal Graph Neural Network:
 - python >= 3.7
 - numpy==1.23.5
 - pandas==2.0.3
 - tensorflow==2.12.0
 - keras==2.12.0
 - Attention-Temporal Graph Convolution Network:
 - python==3.7
 - numpy==1.18.5
 - pandas==1.1.4
 - torch==1.6.0
 - torch_geometric_temporal==0.40
 - torch_geometric==1.7.0

3.2.2.2.4 Installation Guide

To install and run the GNN models of PDLC-DL the following steps are needed, after handling all dependencies (create a cluster, generate data with the CODECO synthetic data generator).

²³ <https://www.kaggle.com/code/elmahy/a3t-gcn-for-traffic-forecasting>



- Clone the CODECO PDLC-DL [ICOM-GNN](#) repository:

```
git clone https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/decentralised-learning/model-selection-and-training
```

To run the STGNN model do the following steps:

- Navigate to the STGNN directory and install the requirements of the “requirements.txt” file, where the pre-requisites mentioned in the previous subsection are listed.

```
cd STGNN  
pip install -r requirements.txt
```

- Run the Python code:

```
python3 main.py
```

- The following CLI arguments can be used when running the model:

```
( '--metric', type=str, default='ram', help='metric to be predicted, can be `cpu` or `ram`')  
( '--train_size', type=float, default=0.5, help='train set percentage')  
( '--val_size', type=float, default=0.2, help='validation set percentage')  
( '--batch_size', type=int, default=64, help='batch_size')  
( '--input_sequence', type=int, default=12, help='number of previous timesteps given as input to the model')  
( '--forecast_horizon', type=int, default=12, help='number of timesteps that the model predicts ahead')
```

Alternatively, to run the model the following steps can be followed:

- Pull the [model's image for Docker Hub](#) and run it with the commands described below. The user should substitute the 'path/on/host/to/write/predictions/csv/files' and specify the desired path to save the model's output.

```
docker pull hecodeco/pdlc-dl-gnns-stgnn  
  
docker run -v  
/path/on/host/to/write/predictions/csv/files:/app/outputs  
hecodeco/ pdlc-dl-gnns-stgnn
```

To run the **A3T-GCN** model follow the steps:

- Navigate to the A3T-GCN directory and install the requirements of the “requirements.txt” file, where the pre-requisites mentioned in the previous subsection are listed.

```
cd A3T-GCN  
  
pip install -r requirements.txt
```

- Run the Python code:

```
python3 main.py
```

The following CLI argument can be used when running the model:

```
( '--metric', type=str, default='memory', help='metric to be predicted - currently supporting `cpu` and `memory`')
```

Alternatively, to run the model the following steps can be followed:

- Pull the model's image and run it with the commands described below. The user should substitute the 'path/on/host/to/write/predictions/csv/files' and specify the desired path to save the model's output.

```
docker pull hecodeco/pdlc-dl-gnns-a3tgcn  
  
docker run -v  
/path/on/host/to/write/predictions/csv/files:/app/outputs  
hecodeco/pdlc-dl-gnns-a3tgcn
```

3.2.2.2.5 Inputs & Outputs

The two models take as input the 'data.csv' file produced by the CODECO synthetic data generator, which includes timeseries data for a cluster's node metrics. More details about the data can be found in Subsection 4.1. Currently, for the models' input, the metrics selected are 'cpu' and 'memory'.

An example plot of a node's (k8s-worker-3) memory usage timeseries is provided in Figure 12.

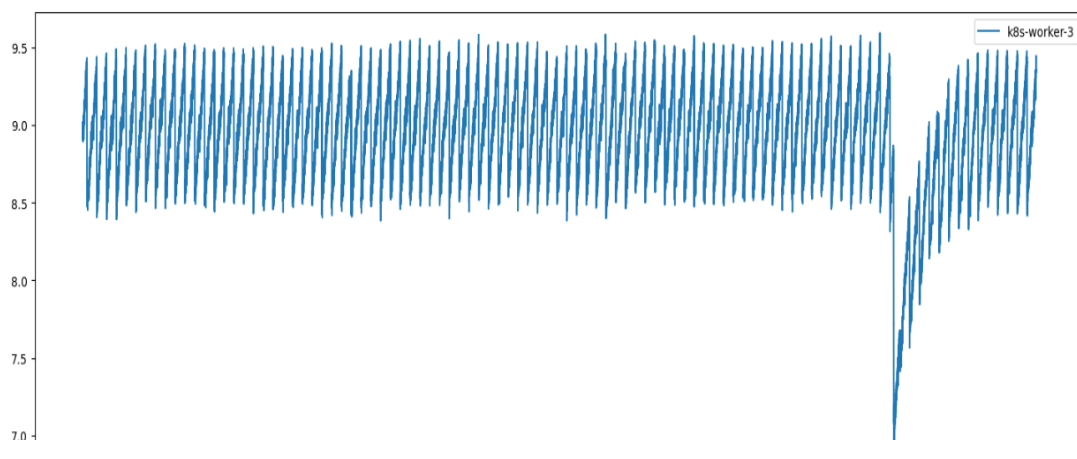


Figure 12: Plot of k8s-worker-3 memory usage timeseries extracted from the Synthetic Data Generator.



Notice that for the example described, the cluster topology considered is the one provided by default on the CODECO synthetic generator, based on its topology.json file.

Before being fed to the models, the input timeseries are processed by averaging the 3-second intervals per minute and then sampling the 1-minute intervals every 5 minutes, thus providing 5-minute frequency timeseries as input to the models.

The models make predictions for each cluster node, based on 12-timestep historical observations, for a 12 timestep forecasting horizon. The output is written in a <node_name>.csv file for the predictions of every node. A plot of the STGNN model is actual and forecasted memory usage values for 'k8s-worker-3' node is provided in Figure 13.

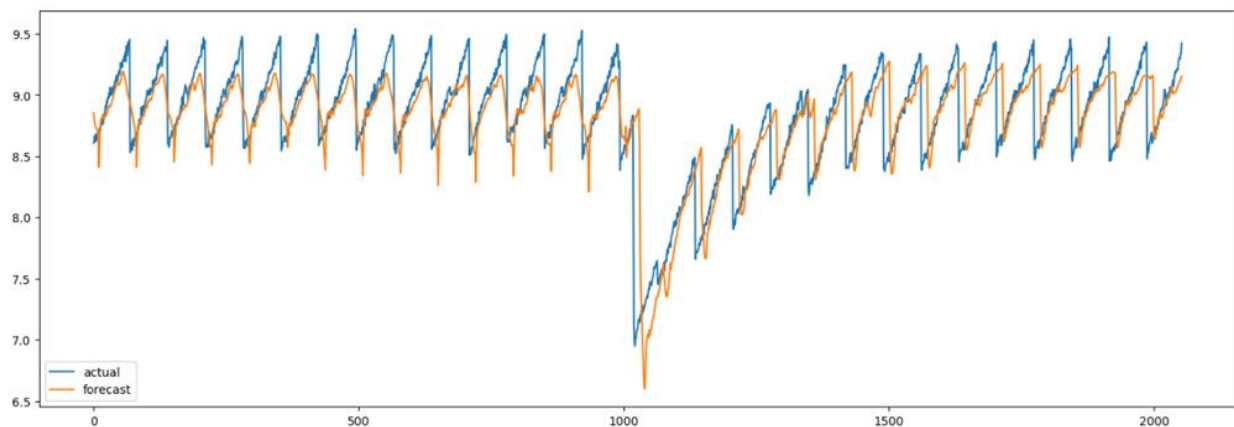


Figure 13. Plot of the STGNN model is actual and forecasted memory usage values for 'k8s-worker-3' node.

3.2.2.2.6 UML Diagram

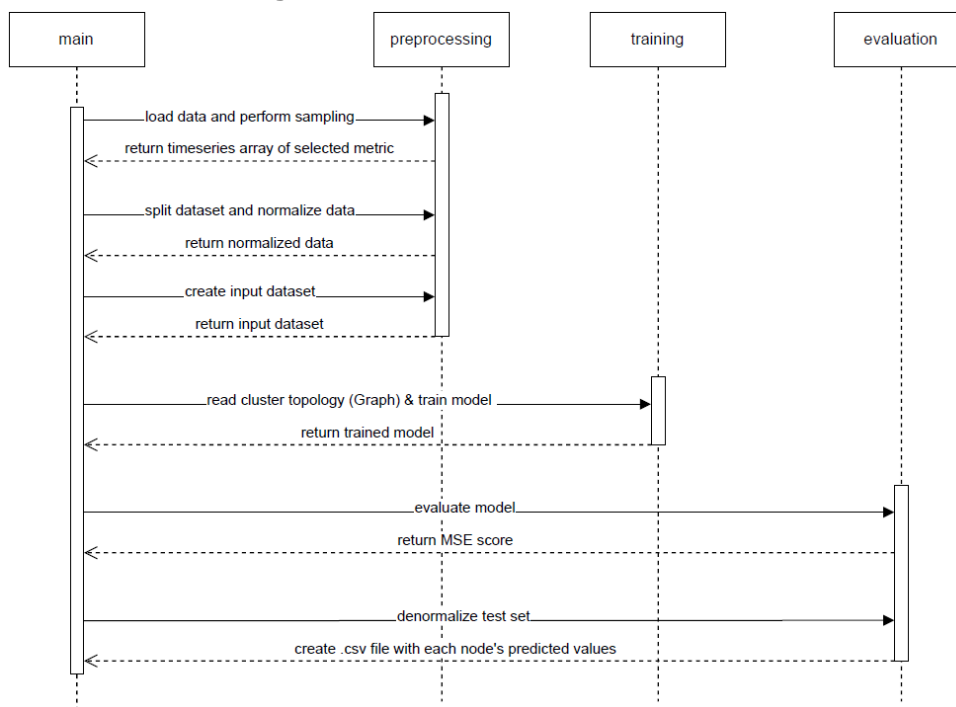


Figure 14. PDLC UML Sequence diagram of GNN models.

3.2.2.2.3 Model-based Multi-agent Reinforcement Learning (MARL) Models

3.2.2.2.3.1 Usage Scenario

The A3C algorithm within the PDLC-DL, available in the PLDC repository [MARL A3C UPM](#), is designed to facilitate *Multi-Agent Reinforcement Learning (MARL)* in various contexts. Its usage scenario includes training and deploying RL models in decentralized environments, enabling multiple agents to interact and learn from their experiences while preserving privacy and adaptability. MARL is one of the approaches in study in CODECO, to support the decentralised learning, privacy-preserving requirements [1].

A potential relation of this component relates with the network forecasting mechanism to be interfacing with NetMA, sub-component network management state.

3.2.2.2.3.2 Selected Technologies

The following technologies are utilized in the PDLC-DL component with the A3C algorithm:

- Python: The primary programming language used for implementing the A3C algorithm.
- TensorFlow: TensorFlow is employed as the deep learning framework for developing and training the neural networks within the A3C algorithm.
- NumPy: NumPy is used for numerical computations and efficient data manipulation.

3.2.2.2.3.3 Prerequisites

In terms of hardware and software, the PDLC-DL component with the A3C algorithm has the following pre-requisites :

- Hardware: The hardware requirements depend on the complexity of the reinforcement learning tasks and the size of the neural networks used in the A3C algorithm. A CPU with multiple cores or a GPU can significantly accelerate training.
- Software:
 - Python > 3.5
 - TensorFlow > 2.4

3.2.2.2.3.4 Installation Guide

[Step-by-step installation guide for setting up the component are available in GitLab](#) and summarized as follows:

- Install Python.
- Create virtual environment.

```
python -m venv venv
source venv/bin/activate
```

- Install pre-requisites.

```
pip install -r requirements.txt
```

- Execute the model.

```
python agent.py --train --num-workers 1
```

3.2.2.2.3.5 Inputs & Outputs

Inputs:

- Environment Data: The algorithm takes environment data, which may include state observations, rewards, and actions, as inputs.

- Configuration Parameters: Parameters that configure the A3C algorithm, such as learning rates, discount factors, and neural network architectures.
- Communication Data: Configuration parameters for communication with other CODECO components to retrieve information.

Outputs:

- Learned Policies: The A3C algorithm produces learned policies that determine agent actions in response to states.
- Training Progress: Training metrics and logs, including reward curves and convergence statistics.

3.2.2.2.3.6 UML Diagram

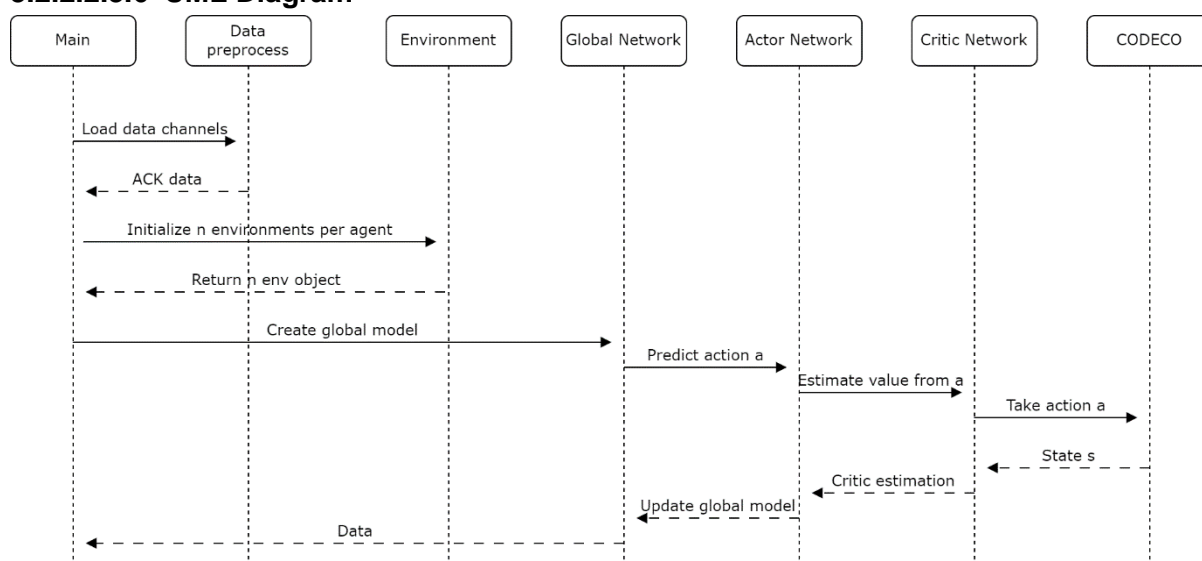


Figure 15. PDLC A3C UML sequence diagram.

3.2.2.3 PDLC-DL (MLOps)

3.2.2.3.1 Usage Scenario

The MLOps subcomponent of PDLC-DL will be responsible for the deployment and monitoring of the PDLC-DL models. The current usage scenario regards the employment of an MLOps workflow, which monitors two deployed models and ensures that the one which achieves the best performance, is used for inference.

To show-case the benefits of this process and in order to provide a proof-of-concept scenario, an MLOps pipeline has been created, where two instances of the timeseries forecasting STGNN model of PDLC-DL are deployed, one using an LSTM layer for the temporal aggregation and one using a GRU layer. The pipeline assesses the models’ performance on the test set, selects the instance with the lowest *Mean Squared Error (MSE)* score and outputs this model’s predictions. The MLOps pipeline consists of four components, one which performs the data preprocessing, two components for training the two model instances and one component that is responsible for evaluating and comparing the models’ performance and selecting the best one, in order to output its predictions.

3.2.2.3.2 Selected Technologies

The MLOps pipeline and the components for each model instance have been developed using **Python** programming language. The technologies used for the source code of the STGNN model are listed in Subsection 3.2.2.2.2 (GNNs). The code of the MLOps pipeline is available under the Privacy-preserving Decentralised Learning and Context-awareness - PDLC / Decentralised Learning / MLOps repository of the [CODECO GitLab](#) (provided by ECL). The rest of the technologies used for each model are listed next:

- **Docker:** Docker is an open platform for developing and running applications and was used to containerize the code of each component of the MLOps pipeline.
- **Kubeflow²⁴:** Kubeflow is an open source platform for machine learning and MLOps, designed to work with any Kubernetes environment, while providing tools for automating machine learning deployment, scaling, and management. It was utilized to deploy the GNN models and run the created MLOps pipeline.

3.2.2.3.3 Pre-requisites

- **Kubernetes:** To run the code of the MLOps pipeline, it is necessary to have set up a Kubernetes cluster.
- **Kubeflow:** Furthermore, Kubeflow must be installed and running. To get started with Kubeflow installation the steps described in this link: <https://charmed-kubeflow.io/docs/get-started-with-charmed-kubeflow> can be followed.
- **KFP Compiler²⁵:** In order for a pipeline to be submitted for execution, it should be compiled to YAML, using the KFP SDK compiler.

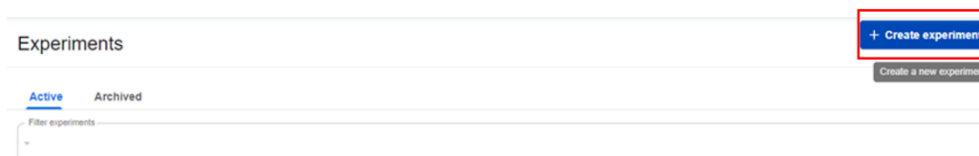
3.2.2.3.4 Installation Guide

In order to install and run the MLOps pipeline, the following steps must be followed through the Kubeflow Dashboard:

- Run a Python server to serve the input files of the pipeline, by running the command:

Create an Experiment: An experiment should be created, in order to select and run the given pipeline.

```
python -m http.server 8000
```

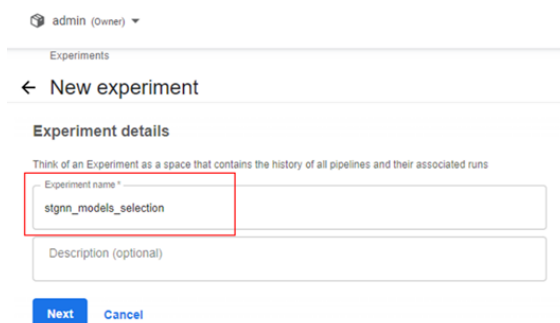


- Write experiment name: In the Experiment details section, the name and description of the created experiment can be specified.

²⁴ <https://www.kubeflow.org/>

²⁵ <https://kf-pipelines.readthedocs.io/en/latest/source/kfp.compiler.html>





admin (Owner) ▾

Experiments

← New experiment

Experiment details

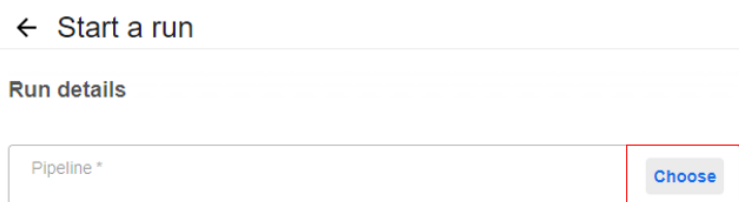
Think of an Experiment as a space that contains the history of all pipelines and their associated runs

Experiment name *
stgmn_models_selection

Description (optional)

Next Cancel

- By pressing “Next”, the “Start a run” page is displayed and the YAML file of a pipeline can



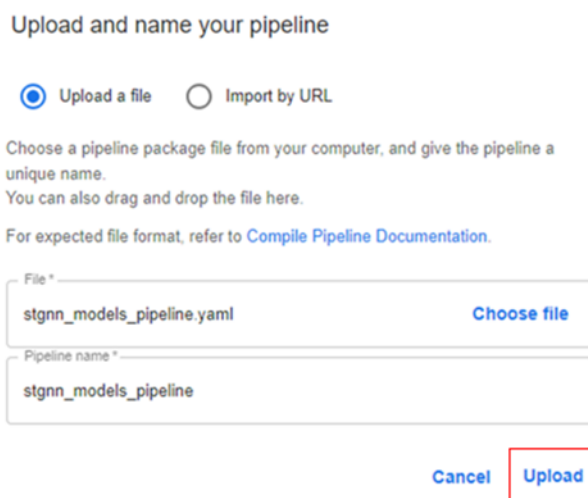
← Start a run

Run details

Pipeline * Choose

be selected.

- At this stage, the YAML file can be imported by choosing the respective file, naming it, and pressing “Upload”.



Upload and name your pipeline

Upload a file Import by URL

Choose a pipeline package file from your computer, and give the pipeline a unique name.
You can also drag and drop the file here.

For expected file format, refer to [Compile Pipeline Documentation](#).

File *
stgmn_models_pipeline.yaml Choose file

Pipeline name *
stgmn_models_pipeline

Cancel Upload

- Now that the Pipeline and the Experiment with which the Run is associated are specified, the user can start the “Run”.

← Start a run

Run details

Pipeline *
stggn_models_pipeline Choose

Pipeline Version *
stggn_models_pipeline Choose

Run name *
Run of stggn_models_pipeline (f95cb)

Description (optional)

This run will be associated with the following experiment

Experiment *
stggn_models_selection Choose

This run will use the following Kubernetes service account. ?

Service Account (Optional)

Run Type

One-off Recurring

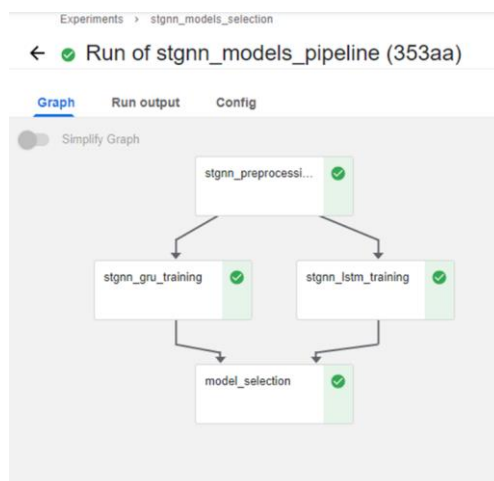
Run parameters

Specify parameters required by the pipeline

data_url
http://146.124.106.207:8000/Downloads/data_new.csv

Start Cancel

- After the components (pods) are initialized and start running, the workflow is completed, when all pods are in state “completed”.



3.2.2.3.5 Inputs & Outputs

The MLOps component of PDLC-DL, takes as input the Python code of each of the four components, their Dockerfiles and respective .yml files, as well as the Python file that specifies the pipeline by describing the input and output of each component, by referencing the components' .yml files. By compiling this file using the KFP Compiler, the final .yml file of the pipeline is produced. The pipeline's parameter specifying the input data file location is required as well and is given as input, during the “Start Run” step, in the ‘Run parameters’ field.

After the pipeline is uploaded to Kubeflow and all components are completed, the output of the pipeline is the MSE score of model instance, which achieved the best performance on the test set.

3.2.2.3.6 UML Diagram

3.2.3 Next Cycle features

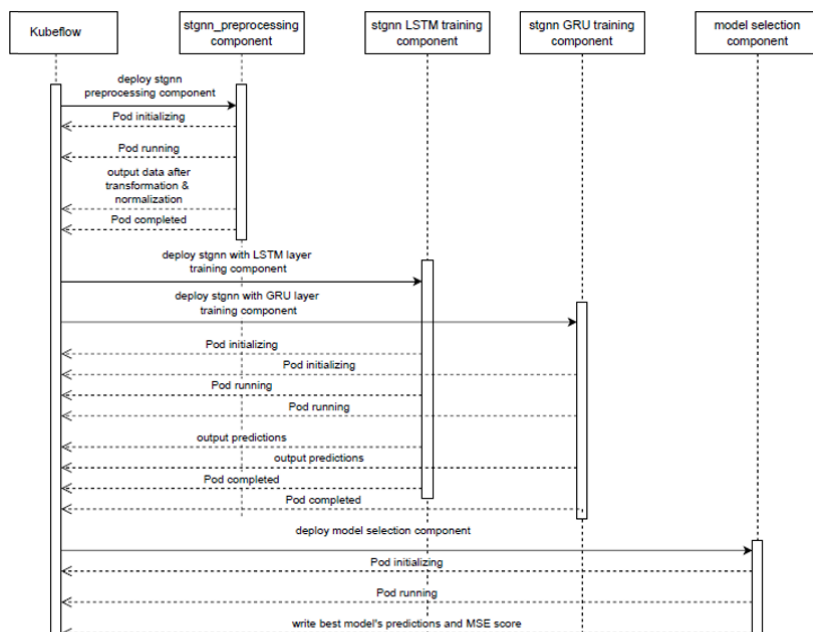


Figure 16: UML diagram for the PDLC MLOps sub-component.

The PDLC next cycle features are presented per order of priorities, starting from the most relevant one:

- Provide an end-to-end connected prototype of PDLC, fully integrating PDLC-CA and PDLC-DP, completing all internal interfaces and all external interfaces.
- In PDLC-CA, explore further metric aggregation approaches, and validate different approaches in terms of improvements that can be provided to the SWM scheduler, and to other components.
- In PDLC-DL:
 - Select the optimal model(s) to consider based on specific use-cases.
 - Customize and refine the output generated by PDLC-DL and forwarded to SWM to best suit the needs of the scheduler in that component; improve the rewarding concept.
 - Implement a preliminary MLOps pipeline that selects the best-suited model for different objectives from the ones available.

3.3 NetMA: Network Management and Adaptation

3.3.1 Component Description

NetMA, illustrated in Figure 17, is an advanced network management and adaptation solution designed to streamline the configuration of interconnections, enhancing the flexibility of Edge-Cloud operations. It effectively addresses the integration of internetworking control, catering to diverse network environments, including fixed, wireless, and cellular networks that are anticipated to be managed by CODECO. Within CODECO, NetMA handles critical aspects such as network softwarization, semantic interoperability, secure data exchange, predictive behaviour, and integrated network capability exposure through standard-based mechanisms and K8s APIs. Furthermore, AI/ML techniques are employed to glean insights from network events, facilitating closed-loop automation and adaptive control.

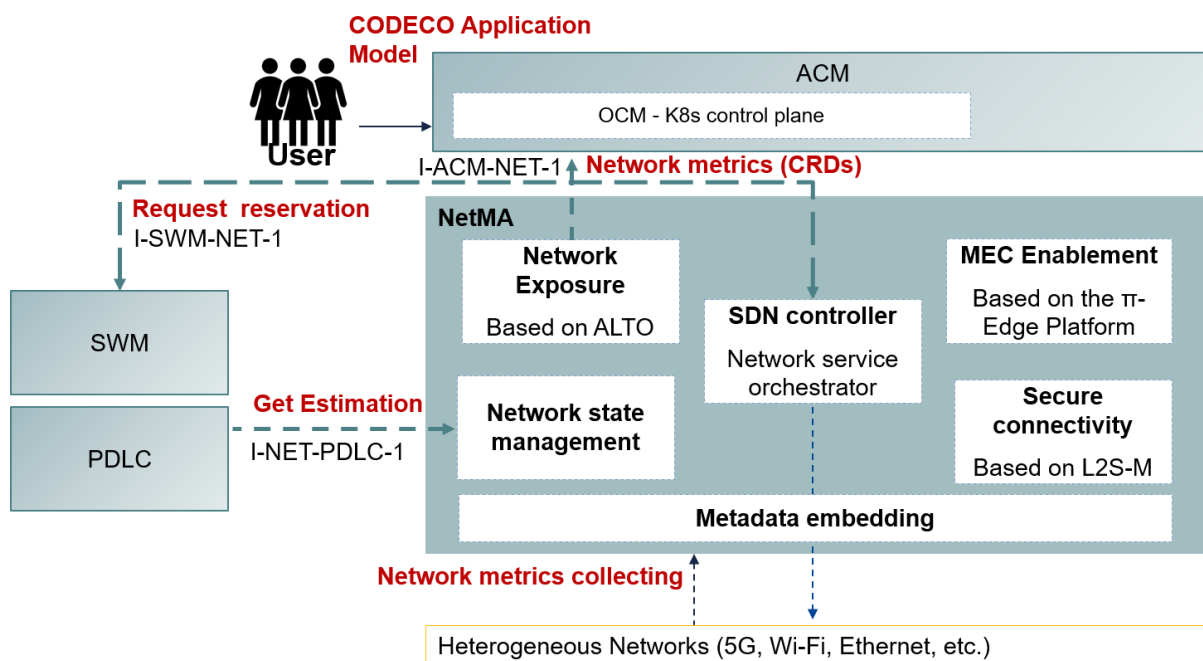


Figure 17. NetMA high-level architecture.

In the area of network softwarisation, NetMA focuses on providing *Function-as-a-Service (FaaS)* to the Edge and automating network resource management to meet specialised service requirements. It actively promotes the integration of diverse networking domains and extends the physical reach of computing facilities, ensuring seamless semantic compatibility between internetworking services. Table 6 provides a summary of the main interfaces with other CODECO components, providing status on their implementation.

Table 6: NetMA interfaces to other CODECO components.

ID	Description	Type	Direction	Status
I-ACM-NET-1	Integration of the NetMA CRD/CR(s) and respective operator(s)	CRD	Input	Partially implemented
I-SWM-NET-1	Used to request NetMA to perform QoS reservations. SWM may perform requests for reservations to NetMA, based on the scheduling/placement decisions. Furthermore, NetMA may also handle network related CRs (<i>Endpoint</i> , <i>NetworkLink</i> , <i>NetworkNode</i> , <i>NetworkPath</i>) to reflect network conditions.	CR	Bi-directional	Not yet available
I-NET-PDLC-1	Request infrastructure estimation	CR	Bi-directional	Not yet available

3.3.2 Sub-components' Specification and Implementation

3.3.2.1 Network Exposure

3.3.2.1.1 Usage Scenario

The network exposure sub-component of NetMA handles the exposure of CODECO networking metrics to other CODECO components (e.g., ACM, SWM). The exposure will be

handled periodically and may also be handled on-demand. Internal NetMA components, such as the Secure Connectivity component will also request specific data from this component.

The network exposure module is expected to provide state information at a link level, state information at a path level. The current code provided in this early release is a proof-of-concept which serves the purpose of demonstrating how information can be exposed. The initial subset of networking parameters to be exposed in future versions of this module is provided in Annex I, Table 16. **These parameters are not yet used in the current available code of NetMA sub-components.** The proof-of-concept code files are summarized in Table 7.

Table 7: NetMA Network Exposure sub-component code summary.

File	Function
alto_core.py	Launches ALTO's module and its required plug-ins.
api/web/alto_http.py	HTTP API to receive requests.
topology_writer.py	Saves locally the maps created. Used as a debugging tool.
topology_maps_generator.py	It synchronizes with the bgp speaker to receive topology information and generates a networkx abstraction.
parsers/yang_alto.py	Provides context data and a uniformed format to the response
bgp/manage_bgp_speaker.py	It acts as a speaker, listening and exporting BGP information to us.

3.3.2.1.2 Selected Technologies

- **Python** v3.0
- **ALTO** (Application Layer Traffic Optimisation Protocol) [2].
- **ExaBGP**²⁶, an open source solution Border Gateway Protocol (BGP) speaker implementation in python, often used in SDN and network automation scenarios, to advertise, withdraw or modify BGP routes.

3.3.2.1.3 Pre-requisites

- **Hardware:**
 - The proof-of-concept provided in the CODECO GitLab can run in devices holding at least 2 CPUs and 4GiB of Memory RAM.
- **Software:**
 - Python 3
 - K8s
 - Software libraries mentioned in [requirements.txt](#)
 - exaBGP.

3.3.2.1.4 Installation Guide

- Obtain the code at: [Eclipse Research Labs / CODECO Project / Network Management and Adaptation - NetMA / Network Exposure · GitLab](#), e.g.:

```
git clone git@gitlab.eclipse.org:eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/network-exposure.git
```

²⁶ <https://github.com/Exa-Networks/exabgp>.

- In your local environment, launch a python virtual environment in order to avoid mixing dependencies that could cause potential disagreements:

```
python -m venv .venv && source ./venv/bin/activate.
```

- Install all dependencies:

```
pip3 install -r requirements.txt.
```

- Start the module with:

```
python3 alto_code.py
```

In the documentation it will be included also a list of commands during the deployment of new functionalities.

3.3.2.1.5 Inputs & Outputs

Input:

This module will receive two inputs: one from the network devices, providing topological information, and other one from its clients to request for information. The first of the inputs is right now a BGP message, as it is defined in the original ALTO IETF RFCs. In the context of CODECO, new capabilities are expected to be developed, such as the support of LLDP for building the inter-cluster network topology taking profit of the Layer-2 overlay connectivity solution provided by L2S-M. The input for this module is provided in Annex I; Table 16.

In this early proof-of-concept, an API REST for ALTO protocol with the next URIs is available, exemplifying how metrics could be exposed:

- 127.0.0.1:5000/ → Returns an index with all the services available.
- 127.0.0.1:5000/costmap → It returns a json with the costs map.
- 127.0.0.1:5000/networkmap → It returns a json with a map of the network.
- 127.0.0.1:5000/all/a/b → It returns all the disjuncts paths between A and B.
- 127.0.0.1:5000/best/a/b → It returns the optimal path to link A and B.

Output:

The output provided is currently the one defined in RFC7285. We have decided to use an API REST as the consumption API due to the easy access that it provides from any other technology and the easy translation to a future multi-cluster system.

However, in CODECO, the output is expected to also be made available via the NetMA CRs/CRDs, to other components.



3.3.2.1.6 UML Diagram

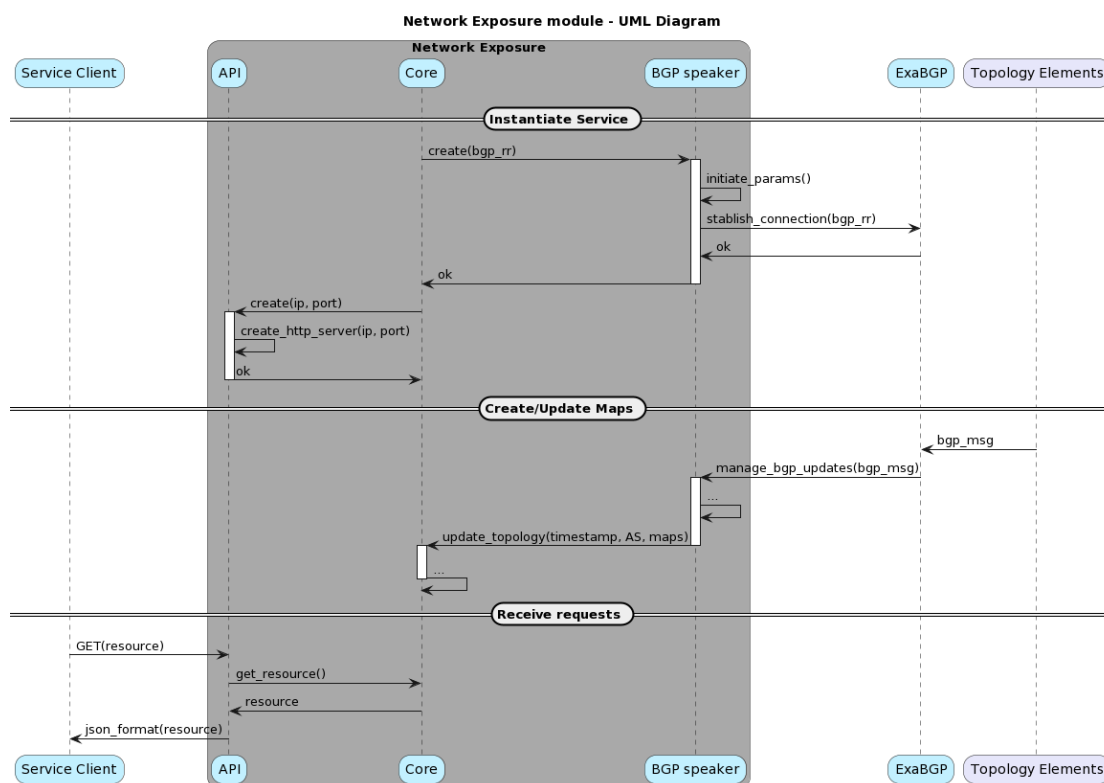


Figure 18. NetMA Network Exposure UML Diagram.

3.3.2.2 Network State Management

3.3.2.2.1 Usage Scenario

This subcomponent was previously called Network State Forecasting. All the intelligence of the CODECO architecture will be located inside PDLC component, so it was decided to swap the name to Network State Management. This component focuses on monitoring network data using a network performance probe. The probe is designed to measure various network metrics through socket communication²⁷, including bandwidth, throughput, latency, packet loss rate, jitter, retransmission rate, and network interface statistics. Its usage scenarios include:

- **Real-time Network Monitoring:** Continuously monitoring network metrics.
- **On-Demand Network Measurement:** Initiating network performance measurements based on external triggers or user requests.

3.3.2.2.2 Selected Technologies

The Network State Management component employs the following technologies:

- **Programming Language:** Python is the primary programming language used for implementing the network performance probe and forecasting algorithms.
- **Socket Communication:** Socket communication is used for gathering real-time network metrics from network devices and servers.

²⁷ Pending to be evaluated by integrating with Docker containers.

The code for the network probe has been initially developed covering the mentioned metrics. It can be found on the [CODECO GitLab](#) under Network Management and Adaptation, specifically on Network State Management, and inside the folder Monitoring. In addition, [Docker image](#) is present on project Docker hub.

3.3.2.2.3 Pre-requisites

The hardware requirements are typically minimal, and they depend on the scale and specific network monitoring needs. A standard computer or server with network connectivity is sufficient.

In terms software requirements, it must be configured or opened via a Web server like Apache or Nginx, as port 80 is the default port for HTTP traffic. By default, on new machines, this port is not open for communication.

3.3.2.2.4 Installation Guide

Step-by-step installation guide for setting up the component:

- Install Python.
- Create virtual environment.
 - `python -m venv venv`
 - `source venv/bin/activate`
- Install pre-requisites .
 - `pip install -r requirements.txt`
- Execute the script locally.
 - `python network_probe.py --host 192.168.1.291 --port 80 --live --delay 15 --bandwidth --latency --congestion --verbose`
- Execute the script in Docker.
 - `docker run --name network-probe -d network-probe --verbose --host 192.168.1.291 --live --delay 15 --bandwidth --throughput --packet-loss --latency --jitter --congestion`

3.3.2.2.5 Inputs & Outputs

Inputs:

- **Destination Host and Port:** The probe receives information about the destination host, namely, IP and port to establish communication and measure network performance metrics. This information is used to target specific network endpoints for measurement.
- **Monitored networking metrics:** The probe accepts a list of network metrics that need to be measured. This list typically includes metrics such as bandwidth, throughput, latency, packet loss rate, jitter, retransmission rate, and network interface statistics, and will be adjusted to the proposed CODECO list of parameters (rf. to Table 7).
- **Data Communication Parameters:** In addition to the networking metrics, the probe may receive data communication parameters to interact with other components in the CODECO project. These parameters can include information about the communication protocol, message format, and endpoints for sending and receiving data.

Outputs:

- **Network Performance Data:** The network performance probe produces network performance data, including the measured metrics. In this proof-of-concept, it is being



provided JSON. It will be served on the specific data communication repository, provided as input. For CODECO, the output (whenever feasible) will be provided via CRDs.

3.3.2.2.6 UML Diagram

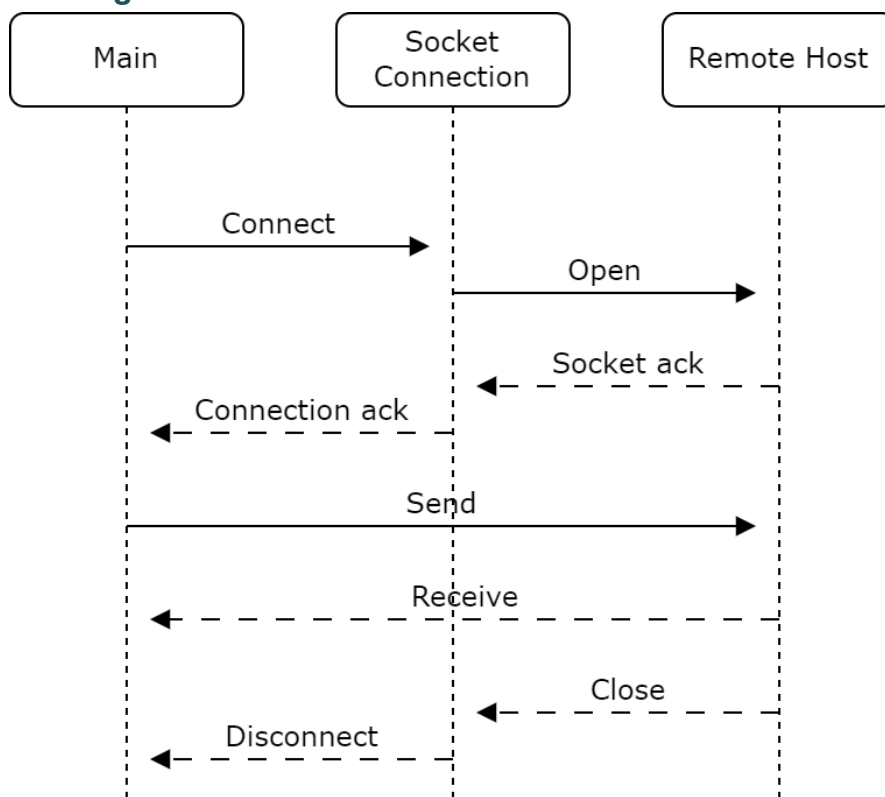


Figure 19. NetMA, network performance probe UML sequence diagram.

3.3.2.3 MEC Enablement

3.3.2.3.1 Usage Scenario

This sub-component brings to NetMA the possibility to integrate data derived from far Edge devices and non-K8s systems, by providing an integration with the *ETSI Multi-Access Edge Computing (MEC) APIs*²⁸.

An example for a usage scenario is as follows. User DEV requests via the CODECO ACM the installation of a distributed application holding several micro-services across the Edge-Cloud continuum. The request contains info regarding the MEC APIs the application wants to use. CODECO provides an optimal operational environment (cluster, multi-cluster) for the application to run, placing the different micro-services across the far Edge-near Edge-Cloud. CODECO allows the microservices that run on the far Edge to make use of the requested MEC APIs that exist on MEC Platforms on specific near Edge nodes.

One such example can be the utilization of the MEC location API by a streaming service to perform resource reassignment (e.g., channel bandwidth) to mobile users, depending on their mobility state, in order to relief an overloaded antenna. This can be done by reducing the resolution and thus the bandwidth being provided to mobile users assessing the service on a mobile node, e.g., a car.

²⁸ <https://forge.etsi.org/rep/mec>

3.3.2.3.2 Selected Technologies

- **Python 3.9.0:** The coding language in which the ETSI MEC APIs are developed.
- **Flask²⁹:** Flask is a micro Web framework written in Python. It is used for the development of the web application that will support the MEC APIs
- **MongoDB:** A nonrelational database which stores the information that is used in the MEC APIs.

The code for the location API has been partially developed and can be found on the [CODECO GitLab](#) under Network Management and Adaptation – NetMA and mec-enablement. Table 8 summarises the code.

Table 8: NetMA MEC Enablement sub-component code summary.

File	Function
Controllers/distance_controller.py	query_distance Query information about distance from a user to a location or between two users
Controllers/users_controller.py	queries_users Query location information about a specific UE or a group of Ues
Controllers/zones_controller.py	queries_zones Query the information about one or more specific zones or a list of zones.
Controllers/zones_controller.py	queries_zone Query information about a specific zone
Controllers/zones_controller.py	queries_zone_access_points Query information about a specific access point or a list of access points under a zone
Controllers/subscriptions_controller.py	create_distance_subscription Creates a subscription for distance change notification
Controllers/subscriptions_controller.py	retrieve_distance_subscriptions Retrieves all active subscriptions to distance change notifications
Controllers/subscriptions_controller.py	create_area_subscription Creates subscription to area notifications.
Controllers/subscriptions_controller.py	create_zones_subscription Creates a subscription to zone notifications
Controllers/subscriptions_controller.py	create_users_subscription Create subscription to UE location notifications.

3.3.2.3.3 Pre-requisites

Hardware Requirements

- Existence of Edge nodes

Software Requirements

- Python >= 3.9.0
- MongoDB
- Flask

²⁹ <https://flask.palletsprojects.com/en/3.0.x/>



3.3.2.3.4 Installation Guide

In order [to run the code](#), all that needs to be done is to execute the command:

```
cd mec-location  
python app.py
```

In order to build the image using docker, you have to run:

```
sudo docker build -t meclocation:1.0.0 .
```

3.3.2.3.5 Inputs & Outputs

MEC Location swagger: [Swagger UI \(etsi.org\)](#). The outputs and more information can be found here.

Inputs:

URL of the MEC API and relevant body/parameters (GET methods):

- /queries/distance
- gets the distance between 2 users/devices or between a user/device and a location:
- /queries/distance?address=sip%250_0&location=47.99%2C%2055.22
- /queries/users
- gets the info of all users in specific access points, zones, or with specific addresses(i.e., specific users). 3 lists are given, any of which can be empty.
- /queries/users?zoneId=zone05&accessPointId=AccessPoint0/queries/zones
- /queries/zones
- Query the information about one or more specific zones or a list of zones.
- /queries/zones?zoneId=zone05&zoneId=zone06
- queries/zones/{zoneId}
- Query information about a specific zone
- /queries/zones/zone01
- /queries/zones/{zoneId}/accessPoints
- Query information about a specific access point or a list of access points under a zone
- /queries/zones/zone08/accessPoints?accessPointId=AccessPoint0&accessPointId=AccessPoint3

Outputs:

- /queries/distance
- The distance between 2 users/devices or between a user/device and a location
- /queries/users
- The info of all users in specific access points, zones, or with specific addresses
- /queries/zones
- The information about one or more specific zones or a list of zones
- queries/zones/{zoneId}
- Information about a specific zone
- /queries/zones/{zoneId}/accessPoints
- Information about a specific access point or a list of access points under a zone



3.3.2.3.6 UML Diagram

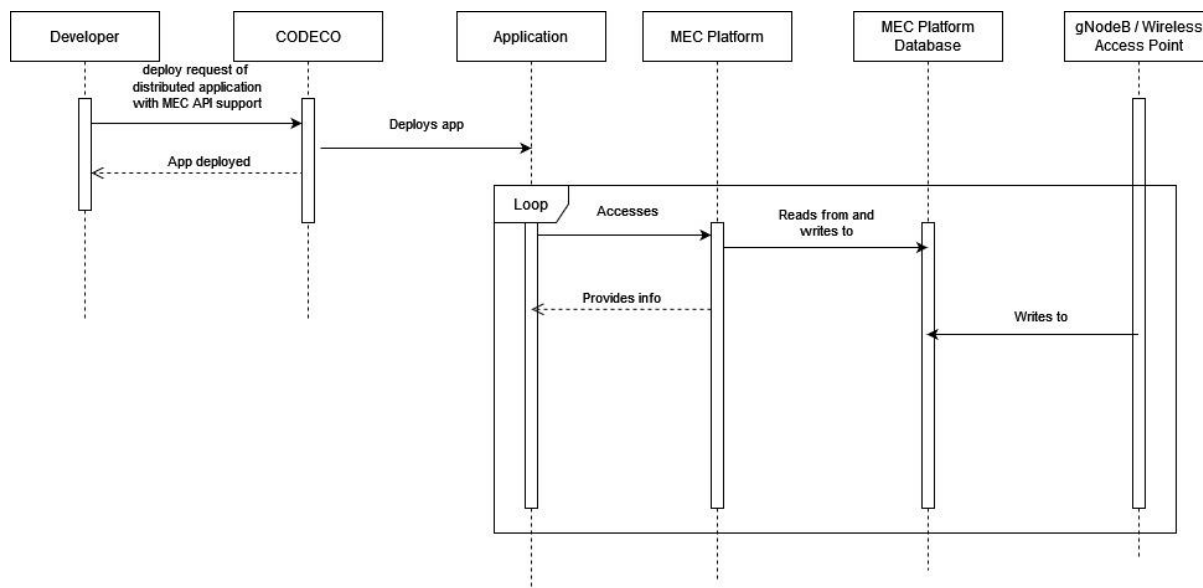


Figure 20: NetMA MEC Enablement UML diagram.

3.3.2.4 Secure Connectivity

3.3.2.4.1 Usage Scenario

As indicated in [1], this sub-component serves as the primary connectivity mechanism within the context of the project, and it is based on L2S-M³⁰ [5]. Succinctly, L2S-M enables the creation and management of virtual networks in microservices-based K8s platforms, allowing workloads (or as they are commonly referred, pods) to have secure and isolated link-layer networks. L2S-M achieves this virtual networking model through a set of programmable link-layer switches (PLS) distributed across the platform, which form an overlay network relying on IP tunnelling mechanisms (specifically, using virtual extensible LANs or VXLANs). This overlay of programmable link-layer switches serves as the basis for creating virtual networks using SDN.

To support the full programmable aspect of the overlay, L2S-M uses an SDN controller to inject the traffic rules in each one of the switches, and to facilitate the implementation of distributed traffic engineering mechanisms across the programmable data plane. For instance, priority mechanisms could be implemented in certain services that are sensitive to latency constraints.

For the first phase delineated within the context of the project, where one single cluster scenario is considered, we have identified how to progress on the basis of L2S-M in order to address the connectivity requirements of this phase. In particular, two main lines of progress are contemplated: *i)* to collect information about the performance achieved by the overlay in order to support the creation of virtual networks over it; and *ii)* to provide a CR/CRD type interface to enable the interaction with the rest of the components defined in the CODECO architecture. Figure 21 illustrates the design that has been defined to address the development proposed by both lines.

³⁰ <https://github.com/Networks-it-uc3m/L2S-M>

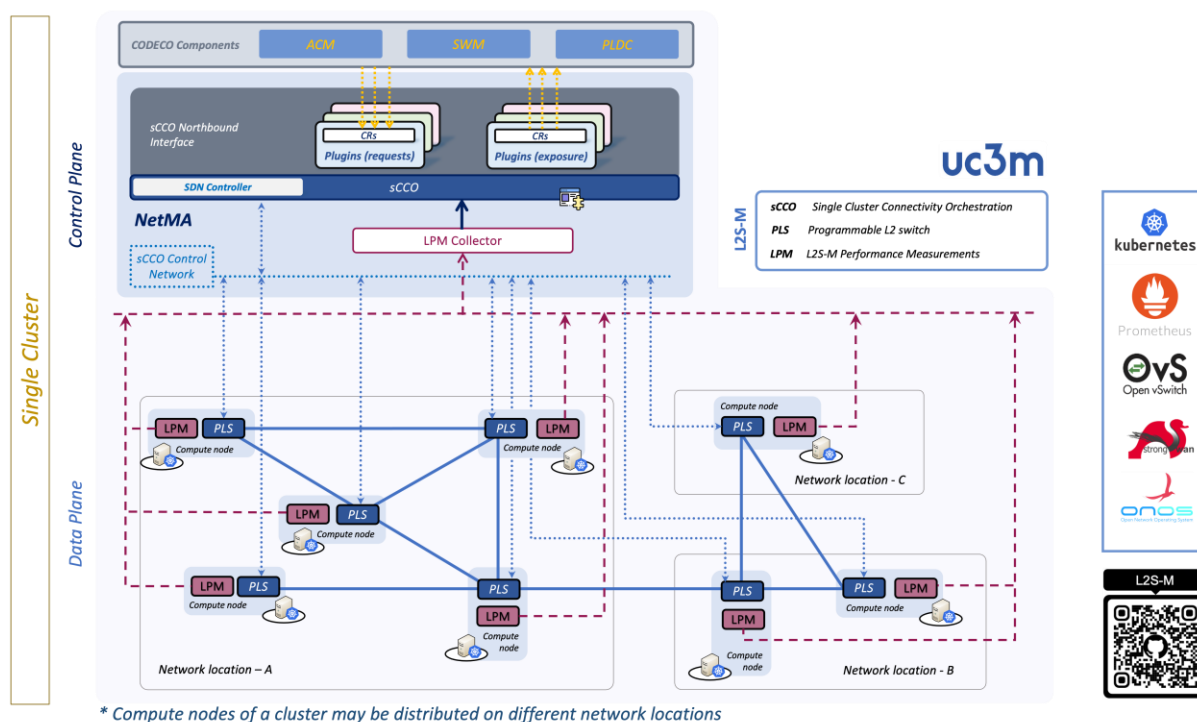


Figure 21. Secure Connectivity design for the initial phase.

This design considers a new module incorporated into L2S-M to address the first of the lines mentioned above (*i.e.*, collect performance overlay information), referred to as *L2S-M Performance Measurements (LPM)*. This module has been developed to flexibly and automatically collect performance metrics of the connectivity provided by L2S-M within a single K8s cluster. To achieve this, LPM conducts a comprehensive network performance profiling of the overlay network, considering various network performance metrics (e.g., available bandwidth, end-to-end delay, etc). Afterwards, LPM, facilitates the exposure of the collected metrics through its LPM Collector component and via a dedicated HTTP endpoint within the cluster. This allows the metrics to be visualized through any third-party application, such as Grafana. Moreover, the design of this module also encompasses the potential for the easy and agile inclusion of new, and tailored metrics. All the details concerning the development of LPM are carefully elaborated in the [CODECO repository](#).

On the other hand, it includes an additional module referred to as *Single Cluster Connectivity Orchestrator (SCCO)* to provide the CR/CRD-type interface defined in the second line. With respect to this latter, the development is currently in progress, as the consortium is evaluating the parameters/requirements that this interface should support based on how the rest of the components are expected to interact (in one or other employs the following technologies) with the Secure Connectivity sub-component of NetMA.

Next, we outline an operational workflow where we present in a concise manner the usage scenario:

1. L2S-M collects different overlay network performance metrics through the LPM module (single cluster overlay information). This information is necessary for the internal operations of L2S-M, as well as to provide it to the SWM component.
2. The sCCO discovers the overlay network topology leveraging the L2S-M SDN controller and receives the performance metrics from the LPM Collector.

3. Then, the sCCO uses its plugins to expose relevant information to other CODECO components in the form of CRs. In particular, the overlay network topology, and its performance metrics are provided to the SWM to determine the network path selections.
4. In addition, the sCCO processes and handles requests (in the form of CRs) from other CODECO components. For instance, the SWM requests the creation of a virtual network to connect two different pods. The request specifies the network path to be used and QoS demands through the appropriate CRs.
5. To create the network path, the sCCO installs at every PLS involved (through its control network) the appropriate traffic-flow rules using its SDN controller.
6. The sCCO confirms the creation of the network path and the QoS demands. Eventual QoS situations (e.g., link-congestion) are notified to SWM using the appropriate CR.

We want to note that LPM is an integral part of L2S-M that measures the performance metrics of the overlay network for internal use. As commented above, these metrics will also be provided to the SWM to aid in the network path selection. These overlay performance metrics, may be provided to any monitoring component that requires them. In this case, the interface to provide this information will need to be defined.

3.3.2.4.2 Selected Technologies

This NetMA sub-component employs the following technologies:

- K8s.
- L2S-M, which implements a K8s operator to enable link-layer virtual networking in Kubernetes clusters.
- LPM, based on Prometheus and developed using the programming language Go.

3.3.2.4.3 Pre-requisites

Since L2S-M is a consolidated open source tool, which has its official repository³¹ where its requirements as well as its installation guide are elaborated, from here on this section refers to the new L2S-M module developed in this initial phase of the project. That is, the LPM module. In this context, the pre-requisites of LPM are indicated next:

- An operational installation of L2S-M in a K8s cluster.
- The cluster must have at least two compute nodes to enable the performance measurements to be carried out through the overlay that L2S-M creates over these nodes.

3.3.2.4.4 Installation Guide

The following step-by-step guide assumes that the installation of the LPM module will be performed in a K8s cluster consisting of three nodes, *node-a*, *node-b*, and *node-c*, where the first of these nodes (i.e., *node-a*) implements the Kubernetes control plane. Within this scenario, an LPM instance will be created in each of the available nodes, and an LPM Collector instance in the *node-a*.

³¹ <http://l2sm.io>

NOTE: The content of the following guide, as well as the files referred to in the subsequent commands, is further elaborated in the [repository](#).

1. Clone the repository where all the LPM artifacts (software and configuration files) are located:

```
git clone <url-secure-connectivity-repo>32
```

2. Deploy the LPM Collector within the cluster:

- Configure the targets in the Kubernetes *configmap* included in *LPM/lpm-c/prometheus-config.yaml* with the aim of specifying the LPM instances endpoints from which metrics should be collected. At this precise moment, the instances are not created yet, so that the endpoints are yet to exist. Nevertheless, LPM Collector can still be configured by pointing to the local DNS service. Example of the Prometheus ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: prometheus-config
  namespace: prometheus
data:
  prometheus.yml: |-
    global:
      scrape_interval: 15s
      evaluation_interval: 15s

    scrape_configs:
      - job_name: 'prometheus'
        static_configs:
          - targets:
              - node-a-lpm:8090
              - node-b-lpm:8090
              - node-c-lpm:8090
```

- **NOTE:** The configuration example provided within the repository, the scraping interval is set to 15 seconds, and three LPM instances are scrapped.
- Run the configuration file from the previous step:

```
kubectl create -f LPM/lpm-c/prometheus-config.yaml
```

- Deploy the LPM Collector:

```
kubectl create -f LPM/lpm-c/operator.yaml
```

3. Create an L2S-M virtual network where the LPM instances are going to be connected to conduct the overlay network performance analysis:

```
kubectl create -f LPM/lpm/network.yaml
```

³² <https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/secure-connectivity.git>



4. Generate a Kubernetes *configmap* file per LPM instance. In this scenario, since three nodes are considered, we need three configuration files of this type. This configuration file includes the metric to be measured with each of the nodes, and the interval of how often they will be measured. For instance, the configuration in node-a is provided below to profile the throughput, delay, and jitter metrics with the neighbor node-b every 10, 20 and 3 seconds, respectively.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: node-a-config
data:
  config.json: |
    {
      "Nodename": "node-a",
      "MetricsNeighbourNodes": [
        {
          "name": "node-b",
          "ip": "10.0.2.4",
          "jitterInterval": 10,
          "throughputInterval": 20,
          "rttInterval": 3
        },
        {
          "name": "node-c",
          "ip": "10.0.2.6",
          "rttInterval": 20,
          "throughputInterval": 20,
        }
      ]
    }
  }
```

5. Create the Kubernetes *configmap* resources related to the within the cluster from the files generated in the previous step:

```
kubectl create -f LPM/lpm/config/node-a-config.yaml
kubectl create -f LPM/lpm/config/node-b-config.yaml
kubectl create -f LPM/lpm/config/node-c-config.yaml
```

6. Deploy the LPM instances:
 - Write a yaml file with the deployment specifications for each instance (workload) that will participate in the metrics collection. The following file may be used as a reference:




```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-a-lpm
spec:
  replicas: 1
  selector:
    matchLabels:
      app: node-a-lpm
  template:
    metadata:
      labels:
        app: node-a-lpm
    spec:
      containers:
        - name: lpm-container
          image: alexdecb/net_exporter:latest
          workingDir: /usr/src/app
          securityContext:
            capabilities:
              add: ["NET_ADMIN"]
          ports:
            - containerPort: 8090
          volumeMounts:
            - name: config-volume
              mountPath: /usr/src/app/config.json
              subPath: config.json
          volumes:
            - name: config-volume
      configMap:
        name: node-a-config
---
apiVersion: v1
kind: Service
metadata:
  name: node-a-lpm
spec:
  selector:
    app: node-a-lpm
  ports:
    - protocol: TCP
      port: 8090
      targetPort: 8090
```

- Instantiate the LPM instances and their associated services:

```
kubectl create -f LPM/lpm/deploy/node-a-deploy.yaml
kubectl create -f LPM/lpm/deploy/node-b-deploy.yaml
kubectl create -f LPM/lpm/deploy/node-c-deploy.yaml
```

- After each individual instance is deployed, the IP addresses must be manually assigned through the interfaces. These should be configured according to the modules defined configuration in step 4. In the provided example scenario, the IP address are allocated as indicated next:

- IP Address of LPM instance at node-a: 10.0.2.2
 - IP Address of LPM instance at node-b: 10.0.2.4
 - IP Address of LPM instance at node-c: 10.0.2.6
- This IP configuration (e.g., the LPM instance running at node-a) can be done by accessing to each instance and executing the following:

```
kubectl exec -it [node-a-pod-name] -- /bin/bash
ip link set net1 up
ip addr add 10.0.2.2/28 dev net1
```

Once the above steps have been completed, and all instances of the LPM module are active, it is possible to access the HTTP endpoint provided by the LPM Collector component to obtain the measured performance metrics.

3.3.2.4.5 Inputs & Outputs

As inputs, the LPM module has:

- A configuration file in the form of a K8s *ConfigMap* with information related to the frequency that the LPM Collector is going to use to collect the metrics measured by each of the LPM instances.
- A file specifying the deployment options of the LPM Collector component.
- A Kubernetes *configmap* for each of the LPM instances involved in the scenario. Specifically, there will be an LPM instance for each of the nodes comprised within the cluster. Each of these configuration files includes information regarding the performance metrics to be measure, at which intervals, and the neighbours (*i.e.*, LPM instances) against which they are to be measured.
- A file specifying the deployment options for each of the LPM instances. Once again, there will be one such file for each of those instances.

On the other hand, as outputs, the LPM offers an HTTP API where it is possible to retrieve the performance values obtained for each of the configured metrics. This API is accessible from the own cluster through the URL "***http://prometheus:9090/metrics***" and offers such values following the format represented next:

```
alex@l2sm1:~$ curl http://10.0.2.5:8090/metrics
# HELP net_jitter_nodeB:nodeA
# TYPE net_jitter_nodeB:nodeA counter
net_jitter_nodeB:nodeA 0.342
# HELP net_rtt_nodeB:nodeA
# TYPE net_rtt_nodeB:nodeA counter
net_rtt_nodeB:nodeA 0.32
# HELP net_throughput_nodeB:nodeA
# TYPE net_throughput_nodeB:nodeA counter
net_throughput_nodeB:nodeA 2.58
```

3.3.2.4.6 UML Diagram

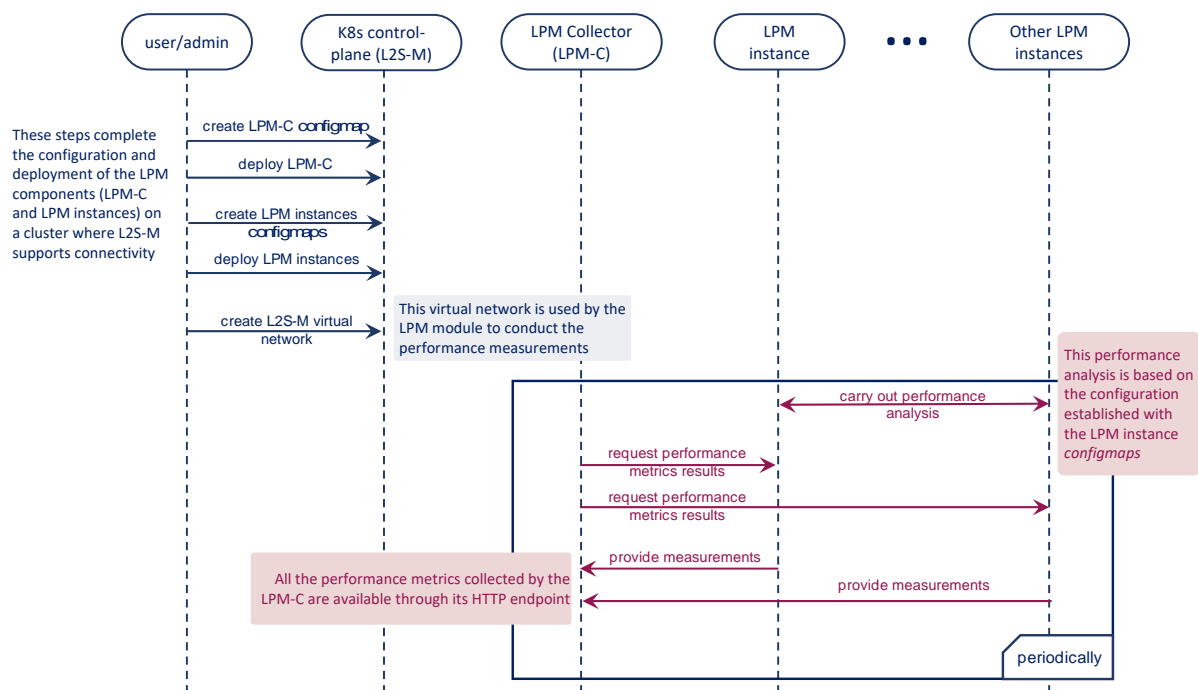


Figure 22. UML diagram of performance metrics collection by LPM module.

3.3.3 Next Cycle Features

The next cycle features are provided by order of priority, starting by the most priority features:

- Complete the different sub-components and realize their operation for a single cluster environment, contemplating also challenges to face in multi-cluster environments.
- Integrate the initial proposed monitoring metrics of CODECO (rf. to Annex I, Table 16), analyzing if they will be enough to serve the purpose of an optimal placement, in alignment with the needs of SWM.
- Complete the interfaces to other components, starting by i) ACM; ii) SWM.

3.4 MDM: Metadata Manager

3.4.1 Component Description

The CODECO MDM component collects, links, and enriches metadata related with the applications to be deployed across Edge-Cloud. This metadata assists in better characterizing the application deployment across Edge-Cloud. MDM is therefore a CODECO component that acts as a gateway between the data world (data workflow) and the K8s infrastructure (compute).

MDM and its sub-components are illustrated in Figure 23. MDM collects metadata from any “native system” that has information on the data, including data stores, catalogues, pipelines that copy and transform data, use-case specific functions that analyse the data, or any other relevant system. For this purpose, MDM relies on **connector interfaces** for each specific data type (I-MDM-E-1).

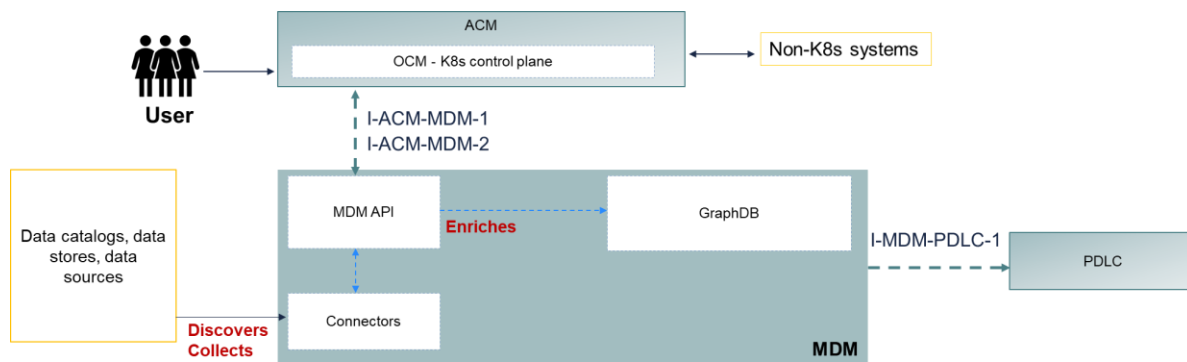


Figure 23. MDM sub-components and APIs to other components.

An MDM connector interfaces to a native data system or CRD and pushes metadata into a knowledge graph through the native (internal) MDM API. By adding new connectors or expanding the graph model, the system can be extended to collect any required metadata. MDM is event-based, relying on Apache Kafka³³. MDM relies on the Kafka event queue, a log of metadata events that occur for an Edge-Cloud setting managed with CODECO. Events describe changes that have happened and correspond to insert/update/delete of metadata entities and relationships.

MDM materializes (subsets of) the event queue in the knowledge graph to meet the needs of other CODECO components. MDM is implemented by three sub-components:

- **MDM API:** Implements the REST APIs that allow metadata to be pushed into the graph database and the graph to be queried.
- **Graph Database:** Stores the metadata graph.
- **Connectors:** They gather metadata and push it into the Graph Database using the MDM Controller APIs.

3.4.2 Sub-components' Specification and Implementation

3.4.2.1 Graph Database

3.4.2.1.1 Usage Scenario

The [Graph Database](#) is the backend storage of the MDM component. The metadata events from all MDM connectors are consolidated here, making it possible for other components to extract insights from the distributed system from a single pane of glass. It is of course possible to request information by directly querying the database using cypher, but other than during development or exploration of the metadata the MDM component is designed to offer this functionality through the MDM API.

3.4.2.1.2 Selected Technologies

The graph database is implemented using Neo4j:

- **Neo4j 4.4³⁴ or higher.**
- **Neo4j APOC³⁵:** APOC (Awesome Procedures on Cypher) is an add-on library for Neo4j that provides hundreds of procedures and functions adding a lot of useful functionality.

³³ <https://kafka.apache.org/>

³⁴ <https://neo4j.com/>

³⁵ <https://neo4j.com/labs/apoc/>

- **Cypher Graph Query language³⁶:** Cypher is a declarative graph query language that allows for expressive and efficient data querying in a property graph. It is the query language for Neo4j and the language opencypher³⁷ is based on.
- **JSON Schema:** It is used to define the entities, their properties, and relationships among them in a language-independent manner. The schemas are available online³⁸.

3.4.2.1.3 Pre-requisites

Detailed pre-requisites for running Neo4j in Kubernetes can be found from the vendor directly³⁹

For CODECO we use Neo4j's Community Edition. Commercial utilization of Neo4j's Enterprise Edition may require a license from the vendor. Please see <https://neo4j.com/licensing/> for details.

3.4.2.1.4 Installation Guide

To install the [Graph database](#) we use the Helm chart provided by Neo4j. The following steps are detailed in the [online documentation in Git](#):

1. Set the following environment variables:

```
export MDM_NAMESPACE=mdm
export MDM_CONTEXT=docker-desktop
```

2. Add Neo4j's Helm chart to the Helm repositories:

```
helm repo add neo4j https://helm.neo4j.com/neo4j
```

3. Update the yaml configuration file for your kubernetes environment (online version [neo4j-helm.yaml](#))

- set the neo4j.password for the database user neo4j
- define the volumes StorageClass if required
- adjust other parameters such as CPU and memory usage as required

4. Install the Helm chart:

```
helm --kube-context=$MDM_CONTEXT install mdm-neo4j -n $MDM_NAMESPACE
neo4j/neo4j-standalone -f ./deployment/neo4j-helm.yaml
```

3.4.2.1.5 Inputs & Outputs

The Graph database is fed metadata events from Kafka by the mdm-controller subcomponent and offers a Cypher API to the mdm-api subcomponent. These interfaces are internal to MDM and declared here only for completeness.

36 <https://neo4j.com/developer/cypher>

37 <http://opencypher.org>

38 https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/graphdb/-/tree/main/schemas/eu.codecohe?ref_type=heads

39 <https://neo4j.com/docs/operations-manual/current/kubernetes/quickstart-standalone/prerequisites/>



3.4.2.2 MDM Controller (APIs)

3.4.2.2.1 Usage Scenario

The MDM controller provides the APIs for other CODECO components to query the metadata graph and for MDM connectors to provide metadata. The MDM Controller is thus a required sub-component for all scenarios where metadata analysis is required for the CODECO use-case. A selected set of MDM connectors depending on the use-case will provide metadata that allows through this subcomponent, allowing other CODECO components like PDLC to get summarize information about the systems and data in the form of parameters to models that provide the best scheduling for a given workload.

3.4.2.2.2 Selected Technologies

The MDM metadata collection is materialized in data systems that make it convenient to manage and exploit the metadata. The key technologies used in the current MDM Controller implementation include:

- **Apache Kafka:** Distributed event store and stream processing.
- **Apache ZooKeeper**⁴⁰, for the overall Kafka management and coordination (e.g., configuration information, naming, providing distributed synchronization).
- **Python 3.9** or higher for the implementation of the APIs and the interaction with Kafka and the Graph database.
- **OpenAPI 3.0** as the standard for the REST API for both input and output
- **Swagger** as the mechanism to programmatically produce the REST API implementation designs and their documentation.

3.4.2.2.3 Pre-requisites

No specific hardware requirements are needed to run the MDM Controller. The Graph database component needs to be installed first.

3.4.2.2.4 Installation Guide

The Graph database needs to be installed first. After that, the order of installation of the MDM Controller sub-components is important.

1. **mdm-zookeeper:** This is the MDM zookeeper sub-component, installed using Helm from the Bitnami⁴¹ chart repository. This is required by the mdm-kafka sub-component.
2. **mdm-kafka:** This is the MDM Kafka sub-component, installed using Helm from the Bitnami chart repository. This is required by the mdm-ctrl sub-component.
3. **mdm-controller:** The MDM API Controller sub-component. Receives the metadata events from Kafka and pushes them into the Graph database.
4. **mdm-api:** Implements the MDM REST API to the Graph database and connector publication API.

Detailed installation information is provided in the MDM API GitLab repository:
https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/mdm-api/-/tree/main?ref_type=heads

⁴⁰ <https://zookeeper.apache.org/>

⁴¹ <https://github.com/bitnami/charts/tree/main/bitnami>



When developing the MDM subcomponents, the docker build and push description provided in the online documentation require providing the docker registry used to store the images. This as well as the tag may change when new versions are developed. Assuming that the mdm-api repository has been cloned locally and is the current directory:

- Install the MDM controller: If you need to change the tag of the docker image, or any other value, provide it in your own YAML file.

```
cd mdm-controller
helm --kube-context=$MDM_CONTEXT -n $MDM_NAMESPACE install mdm-controller ./mdm-controller/src/helm [-f my_values.yaml]
```

- Install the MDM API: If you need to change the tag of the docker image, or any other [value](#), provide it in your own YAML file.

```
cd mdm-api
helm --kube-context=$MDM_CONTEXT -n $MDM_NAMESPACE install mdm-api ./mdm-api/src/helm [-f my_values.yaml]
```

The Helm charts for the installation of the mdm-controller and mdm-api sub-components may become available in the future from an online Helm repository.

3.4.2.2.5 Inputs & Outputs

Events consist of an event envelope and a payload. While the envelope structure is given by MDM, the payload types are application specific. Payloads are either entities or relationships with minimal restrictions on their structure, i.e., they need to contain a type and unique identifier(s). Entities and relationships are defined in JSON schema for CODECO here: https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/graphdb/-/tree/main/schemas/eu.codecohe?ref_type=heads

The MDM API is based on OpenAPI REST (OAS REST API) and integrates the following endpoint groups:

- **Publish** which is the endpoint for connectors writing to the MDM.
- **Events** which is the endpoint to get events from MDM.
- **Graph** gets information by querying the data projection. Initially this would allow for Cypher queries to be performed on the knowledge graph. If because of the experience gained with partners the set of queries required for the CODECO operation can be defined in a closed set, they can be incorporated to this API at a later stage.
- **MDM** is the system information endpoint for getting status and resets or other system relevance functions.

All endpoints are protected and authorized usage enabled.

We provide here a summary of the APIs. The complete description is in swagger online documentation in the Git repository:

https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/mdm-api/-/blob/main/mdm-api/src/python/templates/swagger.yaml?ref_type=heads

Note that the following API groups are not implemented, and responses will always be HTTP status 200 (OK): *Groups, Lookup, Registration, Schemas and Versions*.



Inputs:

- **POST /publish/event:** accept in the body a JSON structured event. The event payload object could be an entity or relationship structure.

Event example:

```
{
  "connector_edf_id": "80b09f2b-5c1c-4c3d-ab90-a17093512ba0",
  "connector_type": "test connector",
  "event_type": "upsert",
  "timestamp": 1683145169,
  "payload": {}
}
```

Payload examples:

Entity:

```
"payload": {
  "json_schema_ref": "urn:hocodeco:compute:1.1.0",
  "edf_id": "2cf4c537-eedc-4995-919c-c29b9b82e9b4",
  "identifier": "364-FKE-835",
  "country_code": "US",
  "name": "TEST System",
  "city": "Cleveland, OH"
}
```

Relationship:

```
"payload": {
  "json_schema_ref": "urn:hocodeco:runs:1.1.0",
  "from_edf_id": "2cf4c537-eedc-4995-919c-c29b9b82e9b4",
  "to_edf_id": "a152ad23-ab23-1567-de23-cd34aa23ecab"
}
```

Outputs:

- **GET /graph/entity/type:** Get a JSON array of existing entity types in the metadata graph.
- **POST /graph/entity/type/{entitytype}:** Get a list of entities in the metadata graph depending on the POST request JSON filter definition.
- **GET /graph/entity/attributes/{entitytype}:** Get a JSON array of all existing attributes in the metadata graph for a given entity type.
- **GET /graph/path/{edf_id}:** Get the graph structure as a JSON object for an entity identified by edf_id. As optional query parameter one can define the number of hops (depth) around the entity that will be included in the resulting graph.

The response is currently formatted in a structure optimized for representation in a graphical user interface.

```
{
  "nodes": [
    {
      "id": 0,
      "label": "string",
      "object": {}
    }
  ],
  "relations": [
    {
      "from": 0,
      "to": 0,
      "label": "string",
      "object": {}
    }
  ]
}
```

- **POST /graph/cypher:** The POST request body accepts a plain text base CYPHER query. The response will be the default JSON structure produced by the Graph database (Neo4j). For example:

Request-body:

```
MATCH (a:compute)-[r]-(b) WHERE b.user_id = "Brandy.Neal@example.com" RETURN a as compute LIMIT 1
```

Response:

```
{
  "identity": 4,
  "labels": [
    "compute",
    "entity",
    "visible"
  ],
  "properties": {
    "_domain": "default",
    "identifier": "AF2CF2B98D7",
    "serialNumber": "none",
    "city": "Zurich",
    "ritssDataClassification": "RII unclassified (Non-Confidential)",
    "_status": [],
    "networkClassification": "General Internal Enterprise Network",
    "country_code": "CH",
    "update_time": "2023-04-24T07:54:41.596286000Z",
    "entity_type": "compute",
    "dataClassification": [
      "Public"
    ],
    "name": "srv-22",
  }
}
```



```

"connector_id": "d79df9ea-efac-360c-b7cc-2cf0de386100",
"exportClassification": "blue",
"edf_id": "523e2537-b717-34e0-b303-638b159d7fcb",
"_id": "523e2537-b717-34e0-b303-638b159d7fcb"
},
"elementId": "1"
}
    
```

- GET /mdm: Get the running version of the mdm-api sub-component.
- **GET /mdm/state**: Get the state of the different sub-components (neo4j/kafka/ctrl/etc)
- **DELETE /mdm/clean**: Delete all metadata in the MDM component. Use with caution, only meant for development and testing purposes.
- **GET /events/sse** Server-Sent Events API-based filtered events from the serialized graph queue in Kafka. This is meant for components that need to be notified of specific metadata events as they happen, as opposed to gathering the current state from querying the metadata graph at time intervals.

3.4.2.2.6 UML Diagram

The momentary interaction between the CODECO components and MDM and among the MDM subcomponents is depicted in Figure 24.

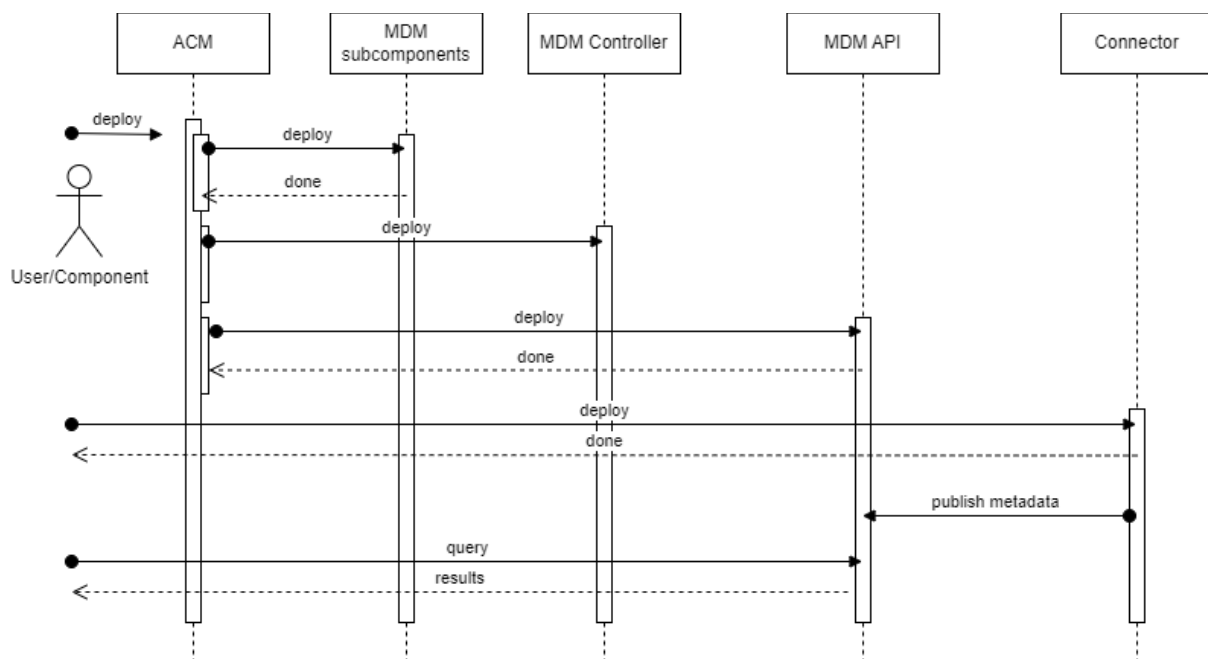


Figure 24. Interaction between MDM subcomponents.

3.4.2.3 Connectors

3.4.2.3.1 Usage Scenario

Connectors send metadata to MDM in the form of events. Events are structured JSON documents. An event contains the following elements:

- The event type, either “insert” or “delete”.
- An identifier that uniquely identifies the connector that issued the event.

- A timestamp.
- The payload, i.e., the metadata.

Metadata is sent as entities and relationships. MDM imposes a basic structure on both entities and relationships to ensure that metadata can be stored as a graph. Entities must have a globally unique identifier and a type. It is the task of the connector to assign these. In addition, entities contain arbitrary numbers of attributes. The mandatory elements of a relationship are source and target entity identifier as well as a type. Relationships do not carry attributes. MDM does not define or enforce a static data model. Instead, the graph data model is defined by the structure of the entities and relationships that are inserted into MDM.

Entities and relationships are defined in JSON schema for CODECO here: https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/graphdb/-/tree/main/schemas/eu.codecohe?ref_type=heads

3.4.2.3.2 Selected Technologies

Connectors may be implemented in any programming language if they make metadata available following the data model of the application framework (in this case CODECO's) and use the MDM Controller API conventions to make it available to the Graph database.

In the context of CODECO, though:

- **Python 3.9 or higher** is the preferred programming language, as it will allow the connector developer to leverage the open source connector SDK.
- **IBM Pathfinder SDK:** Available at <https://pypi.org/project/ibm-pathfinder-sdk> , provides YAML-based configuration and the base communication means for the connector to send metadata events to the MDM API

3.4.2.3.3 Pre-requisites

MDM Connectors do not have any hardware requirements. The MDM Controller component needs to be installed first so that connectors can send the metadata events.

3.4.2.3.4 Installation Guide

MDM Connectors are installed using Helm charts. The parameters required depend on the source of metadata to be inspected by the connector, and thus cannot be listed extensively. At a minimum, the following parameters need to be provide:

- **pathfinder.url:** The URL of the MDM API.
- **pathfinder.kubernetesUrl:** The Kube API of the Kubernetes cluster where the connector runs.

For a complete list of parameters please refer to https://github.com/IBM/pathfinder-python-sdk/blob/main/ibm_pathfinder_sdk/pathfinderconfig.py

3.4.2.3.5 Inputs & Outputs

MDM Connectors get input from the source of metadata inspected. The output is always provided in the form of metadata events sent to the MDM API using the **/publish/event** POST request. For CODECO, helper classes are provided for Python connectors here: https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadata-manager-mdm/connectors/-/blob/main/metadata-model/src/main/python/mdmmodelclasses.py?ref_type=heads

This way connector developers can use the MDM CODECO metadata model directly by instantiating these Python classes for entities and relationships.



3.4.3 Next Cycle features

These are the features we are considering for the next cycle(s):

- **Graph Database:** The metadata-model used to store the metadata in the graph is to be evolved from the initial version following the metadata needs of the use-cases and the CODECO components.
- **MDM API:** There are several aspects we are considering as the next steps for the MDM API, which are ordered following how the development would proceed:
 1. Implement the graph queries that are found to be useful in the different use-case scenarios as part of the API.
 2. Develop a component that produces labels for Kubernetes artifacts that correspond to the information gathered by the graph queries above.
 3. Optionally or alternatively, implement an operator that makes parameters computed from the metadata graph available as CRDs.
- **Connectors API:** We are to develop new connectors or assist the partners in creating connectors for the specific metadata needs of the use-cases and the CODECO components. How the user would automatically deploy the necessary (other than default) connectors for the specific workload is also a topic for development.

3.5 SWM: Scheduling and Workload Migration

3.5.1 Component description

The current implementation of the CODECO SWM component handles the initial deployment, monitoring, and potential migration of application workloads within a single cluster. This implies assisting the efficient (low latency, lower energy consumption, data sensitivity, QoE) placement of applications and their containers across the Edge-Cloud continuum, derived from the information provided by PDLC (e.g., device and node availability; container centrality; network characteristics). The SWM handles, for instance, the “best” placement (based on context-awareness indicators) for the containerized components of an application to be deployed in a cluster, dealing with dynamic properties of available infrastructure, including physical/virtual machines as well as network nodes and links. Further, this placement shall be dynamically adaptable, which implies achieving the efficient (low latency, lower energy consumption, data sensitivity, QoE) migration of containerized micro-services of an application, including their state, across Edge and Cloud, derived from the information provided by PDLC (device and node availability; container centrality, network aspects).

For a further description of the functionality and external interfaces of SWM, please refer to Deliverable D9, Subsection 2.4.3.

The SWM component of CODECO is currently undergoing an OSS release process by partner Siemens. A version of the implemented code is currently only available to the CODECO consortium and to the EC. The OSS version is expected to be released until December 2023.

3.5.2 Sub-components features specification, Inputs & Outputs

SWM consists of two subcomponents: The *QoS Scheduler* and the *Workload Placement Solver*, which are described in more detail in the separate document D11.1.



Interaction with these subcomponents is done via k8s custom resources. The main interface between the QoS Scheduler and the Workload Placement Solver is a gRPC interface described via Protobuf.

3.5.3 Next Cycle features

First of all, the SWM implementation will be made publicly available as open source software. The target date is end of December 2023.

Further extensions of the implemented functionality of SWM are:

- Extend and adapt current SWM custom resources (CRs) to accommodate input from other CODECO components such as PDLC to influence scheduling and workload migration decisions based on insights on system context and behaviour.
- Implement incremental scheduling (changing the scope of applications that are already running)
- Implement workload migration (re-scheduling of applications that are already running, triggered by changes in the environment, to keep the requested QoS).

4 Experimentation Assisted Developments

4.1 CODECO Synthetic Data Generators

4.1.1 Component Description

The true potential of data collection and analysis contributes to the creation of more accurate and robust ML and DL models, leading to knowledge and experience-based decision mechanisms. In the Edge-Cloud continuum area, the data collected from various heterogeneous sources and compute nodes (when running applications) feed the ML and DL models to facilitate the performance of predictive, prognostic, and clustering models, which influence the proactive adaptation decisions to dynamically optimise the entire Edge-Cloud ecosystem (e.g., reduce energy consumption, increase cluster availability, minimise latency). A major problem in the Edge-Cloud continuum resource management domain is the limited amount of data (or lack thereof) prior to application deployment and execution. To address this issue, a synthetic benchmark data generator has been implemented in CODECO, which currently provides output that is useful to other CODECO components.

In the future, it is expected that the data generator will also be able to receive input from CODECO components such as NetMA, thus creating richer, synthetic data sets that are also derived from realistic environments.

The metrics that are currently considered as input for this data generator are aligned with the minimum subset of metrics under discussion in CODECO, provided in Annex I of this deliverable, and based on the initial parameters debated in D9, Annex I.

Overall, the generator mimics the process of the cross-layer data collection from the CODECO components (ACM, MDM, NetMA), and as output results in a consistent data format for further analysis (D9 - Section 2.4.4.3.2.1 Data Availability Considerations). In order to evaluate the functionality of the Data Generator, a custom application (sample) is employed, operating within a default cross-architecture K8s cluster to generate the necessary data.

The CODECO data generator can be considered as an experimentation framework tool, to assist CODECO components like PDLC, by enriching it with relevant data.



4.1.2 Component Specification and Implementation Aspects

The Data Generator is partitioned into two vital sub-components: the collector sub-component, and the synthesizer sub-component which are working in parallel.

- **The Collector:** gathers values for already defined metrics for which no more calculations are needed, based on the cross-layer attributes provided in Annex I, like CPU and memory which belong to predefined metric categories that can be directly collected through the monitoring engine. Currently, metrics that are already available in K8s/Prometheus are fed to the data generator, as Prometheus is the basis for the CODECO monitoring aspects and it constitutes a well-suited monitoring solution for Edge-Cloud orchestration.
- **The Synthesizer:** Several of the defined provided metrics can be treated as composite metrics, and thus cannot be acquired directly by the monitoring engine (Prometheus). Features such as MDM-freshness (*description: healthiness of the node based on data freshness*), ACM-node_failure (*description failures over a time window -EMA-*), ACM-node_sec (*description: level of security guaranteed by the node*), etc., are not directly retrievable, therefore the CODECO data generator adopts a logic that relies on custom formulas adhering to specific logic to calculate these metrics. This logic will further be explored in alignment with the data aggregation aspects under development in the CODECO PDLC (PDLC-CA) component.

4.1.2.1 Communication aspects

The CODECO data generator is expected to get input of other components via CRDs. For experimental use, the Data Generator employs the above mentioned CODECO components as simulated elements, in order to provide the relevant metrics.

The Data Generator considers the Controllers, with one corresponding to each of the CODECO components. These controllers are responsible for updating the CRs (which have been defined in the CRDs). Specifically, during the Data Generator's initialization, three CRDs ([acm-crd.yaml](#), [netma-crd.yaml](#), [mdm-crd.yaml](#)) are created as well as three deployments with proper roles (<codeco-component>-role.yaml and <codeco-component>-role-binding.yaml) to view and list the respective K8s nodes. Moreover, CR objects of the created CRDs with create and patch privileges are also created. The deployments represent the controllers which are: a) ACM Controller which is responsible for delivering data related to ACM component; b) MDM Controller which is responsible for delivering data observability metrics; c) NetMA Controller which delivers the monitored networking parameters.

4.1.2.2 Selected Technologies

The Data Generator was built using the Python programming language (version: 3.10.6) and YAML file format. The latter has been chosen for its compatibility with the Kubernetes (K8s) ecosystem.

4.1.2.3 Prerequisites

Hardware Requirements

- At least a running cluster with nodes based on either AMD64 and/or ARM.

Software Requirements

- Python Libraries
 - kubernetes==27.2.0
(The "kubernetes" Python library streamlines K8s cluster management, resource control, automation, and integration for developers and operators)
 - networkx==3.1



(NetworkX Python library is used for analyzing and visualizing complex networks, including graph creation, manipulation, and algorithms)

- requests==2.31.0
(The "requests" library in Python is used for making HTTP requests, enabling interactions with web services and APIs)
- Tools:
 - Prometheus

4.1.2.4 Installation Guide

The following is a step-by-step guide to the effective installation and operation of the CODECO generator. This information is also available on the [CODECO Eclipse GitLab repository](#) (Experimentation Framework and Demonstrations - > Data Generators and Datasets -> Synthetic Data Generator). Pre-requisites are:

- **A Kubernetes Cluster** with minimum of two nodes. One of these nodes should serve as the control-plane, while the others can act as worker node (The data traffic generator has been implemented without having specific requirements related to hardware properties, meaning that it can run in a k8s cluster including cross-architecture nodes e.g., AMD, ARM etc).
- **Prometheus** Installed in the Kubernetes Cluster (with the specific manner as it is provided below)
- Assume the **administrative role** for the cluster, granting access to **both** the **default Namespace and Service Account**.

NOTE: The content of the following guide is further elaborated in the ECL public [repository](#)

1. Prometheus Installation

- To use Prometheus, it is recommended to install it via the following repository (in order to avoid potential issues):

```
https://github.com/prometheus-operator/kube-prometheus
```

- After navigating to the above GitHub repo, perform the following commands on a Kubernetes Master (control-plane) to install:

```
# Clone Repo
git clone https://github.com/prometheus-operator/kube-prometheus

# Apply
kubectl apply --server-side -f manifests/setup
kubectl wait \
  --for condition=Established \
  --all CustomResourceDefinition \
  --namespace=monitoring
kubectl apply -f manifests/
```

2. Installing the Controllers

- a) Before installing the respective codeco-component controllers, it is mandatory for the user to specify his Cluster's Topology.



Regarding the cluster's topology specification, a file namely `netma-controller/netma-controller-deployment.yaml` must be accessed and in the field `topology.json`, the user must specify his k8s cluster's topology using the Node Names and their perspective adjacency matrix (between which nodes is the connection established). A very comprehensive example is provided the file: `netma-controller/topology.json`. and a scenario is also provided in the documentation section (under the section: *Installing the Controllers/Information related to topology file*).

b) The following commands must be run to use the installation script:

```
chmod -R 777 apply-controllers.sh
./apply-controllers.sh
```

3. Execute the Extractor

The extractor is not running within pods inside the K8s cluster. It resides in the control plane and communicates with the controllers with are located within pods inside the k8s cluster. In order to achieve the communication between the extractor and the controllers, the Python library "kubernetes" (referred in the Subsection 4.1.2.3) is used to establish a proper connection with the cluster.

a) To Execute the extractor and gather results you can run the following:

```
pip install -r requirements-v3.10.6.txt
python3 extractor.py mode=<mode>
```

Note:

- Regarding the above mentioned command, the mode can be either `append` or `write`.
- For the first time of the execution, the user should use `write` mode to create a new `data.csv` file.
- This parser (`mode`) is used to append new information on the `data.csv` file.

4.1.2.5 Input and Output

The data generator does not expect any input, the user only needs to follow the steps described above. The output of the data generator is the production of synthetic data of a K8s cluster, reflecting the production of values from the parameters described in Annex I. Due to the fact that the data generator runs in a continuous loop, a file in comma separated value (.csv) format is initially created and then updated with all relevant data provided by the implemented CODECO component controllers.

4.1.2.6 Architecture Diagram & UML Diagram

Figure 25 provides a representation of the CODECO data generator workflow and interaction with the models of each CODECO component.

Figure 26 provides a communication sequence for the end-to-end data workflow of the data generator.



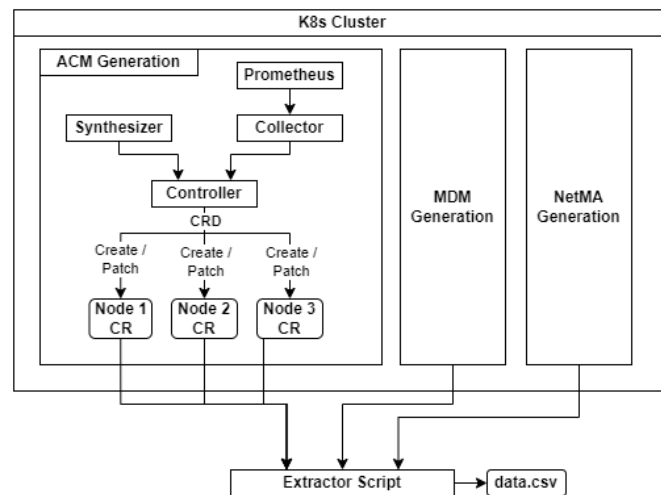


Figure 25. Data Traffic Generator's flow and its interaction with the respective CODECO components and its functionality.

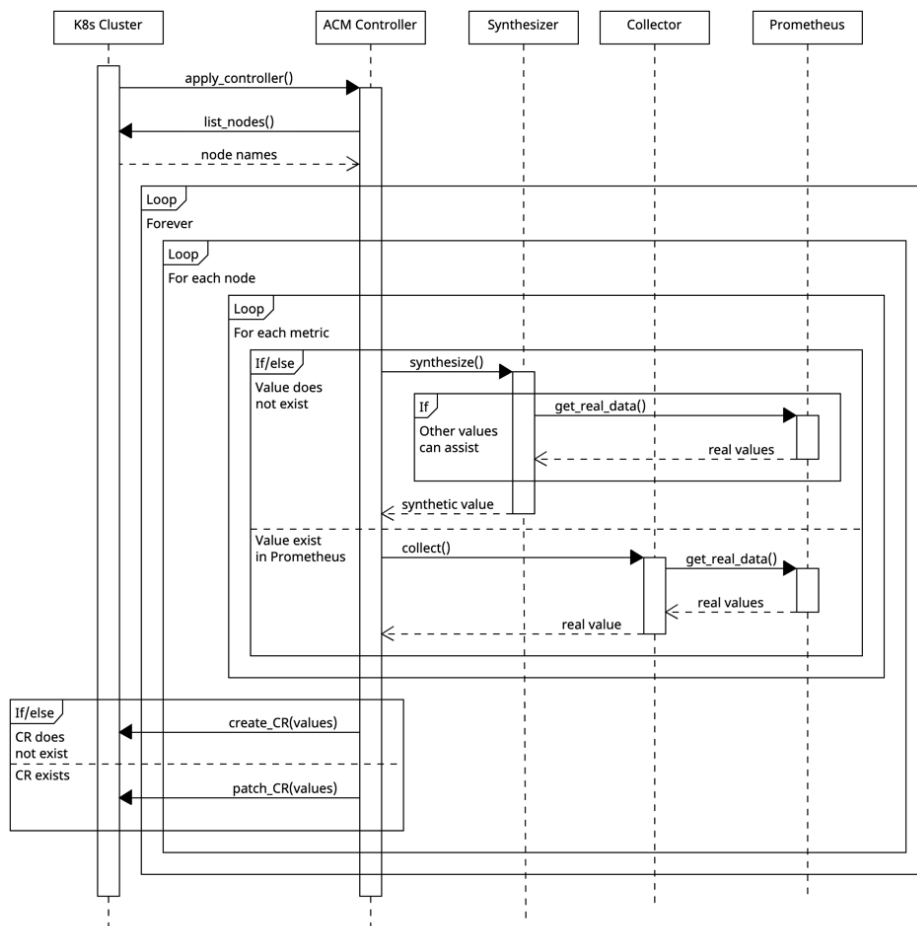


Figure 26. CODECO-component Controller sequence diagram (in this figure: ACM controller).

4.2 Facility Porting Artifacts

A key objective of CODECO is the seamless integration of the developed assets with the international EdgeNet⁴² experimental infrastructure, aiming to enable large-scale, real-world experiments. As stated in CODECO deliverable D9, EdgeNet will “assist in the building of experimentation and novel concepts by the research community utilizing CODECO technologies in an open global testbed with single or multi-cluster domains”.

This section describes the progress made so far in enabling the integration of CODECO with EdgeNet, as well as the initial efforts to explore the EdgeNet intrinsic features that are relevant to our project and how they can potentially be leveraged to accommodate external experimenters. To this end, we provide a set of preliminary examples along with stepwise instructions on how to experiment over both the worldwide EdgeNet testbed, as well as a local K8s cluster configured using the EdgeNet software.

To further facilitate EdgeNet-based experimentation in the future, CODECO envisions to support a dedicated CODECO experiment controller (operator) that will allow for automatically running experiments in CODECO’s experimental facilities (public cloud and partners’ testbeds, EdgeNet or even alternative testbeds such as SLICES-SC⁴³). This strategic approach will have several advantages, including reproducible experimentation, minimal operational support, capacity for large-scale deployment across various infrastructures etc.

4.2.1 EdgeNet Testbed

4.2.1.1 CODECO Namespace Setup

Currently, partner ATH (being the WP5 leader) has successfully registered with EdgeNet testbed⁴⁴ and established a dedicated CODECO namespace (athena-rc). Within this namespace, partners and collaborators can be authorised by the namespace manager to participate actively. Once access has been granted, each participant can download a personalised configuration file (*kubeconfig.cfg*) that is subsequently used to allow deploying and managing the available resources and the CODECO instances. This ensures streamlined collaboration and efficient experimentation within the EdgeNet testbed infrastructure. Moreover, other partners, such as FOR, are already interconnected with EdgeNet, providing the possibility to explore multi-cluster environments across EdgeNet, in cooperation with ATH and other partners.

4.2.1.2 Node contributions

EdgeNet offers to users the option to contribute their on-premises nodes while also to explicitly select these specific nodes for experiments (by declaring this in the corresponding yaml configuration file). As of now, CODECO partners have contributed with the following nodes:

- ATH, one VM node.
- FOR, one physical node with the capability to be interconnected to the CODECO demonstrator cluster(s) in the fortiss IIoT Lab⁴⁵.

4.2.1.3 EdgeNet Probing

Two designated demos have been developed and deployed over the EdgeNet testbeds, and are available via the CODECO Eclipse repository ([Experimentation Framework and](#)

⁴² <https://www.edge-net.org/>

⁴³ <https://slices-sc.eu/>

⁴⁴ <https://www.edge-net.org/>

⁴⁵ <https://www.fortiss.org/en/research/fortiss-labs/detail/iiot-lab>



[Demonstrations ->Edgenet Framework -> demos](#)). These relate with demonstrating the Selective Deployment functionality:

- In the first demo, EdgeNet selects some nodes from pre-specified geographic locations, explicitly defined by a selector in the corresponding .yaml file, to run a simple CDN service.
- In the second demo, based on the location-based node selection, as defined by the selector in the .yaml file, EdgeNet selects nodes to ping a server or IP address, aiming to generate traffic. This example highlights the potential use of EdgeNet as a benchmarking tool for the CODECO use cases, i.e., setting up clients (around the globe) to test the provided services.

The respective selector fields are illustrated in the next code snippet:

```
selector:
  - value:
    - Europe
    - Asia
    - North_America
  operator: In
  quantity: 3
  name: Continent
---
apiVersion: v1
kind: Service
metadata:
  name: cdn-service
  namespace: athena-rc
spec:
  type: ClusterIP
  ports:
    - name: http
      port: 80
      targetPort: 80
    - name: vlc
      port: 8080
      targetPort: 8080
  selector:
    app: cdn-server

selector:
  - value:
    - North_America
  operator: In
  quantity: 2
  name: Continent
- value:
  - Europe
  operator: In
  quantity: 2
  name: Continent
```

To run the demos, users should use the following guidelines:

Pre-requisites

For starters, users must:

- Create an EdgeNet account by signing up on the landing app⁴⁶.
 - In case of registering an institution to EdgeNet, an institutional email and information should be provided for authorization purposes.
- Install `kubectl`⁴⁷, the Kubernetes command-line interface.
- Register to a *namespace*.
- Acquire an up-to-date *kubeconfig file* - a configuration file - obtained from the landing app
Download my kubeconfig file tab after logging in to EdgeNet.

⁴⁶ <https://www.edge-net.org/pages/running-experiments.html>
⁴⁷ <https://kubernetes.io/docs/reference/kubectl/overview/>

- Download the `.yaml` files - i.e `cdn-service-example.yaml` and `ping-me-example.yaml` from the EdgeNet framework subgroup of the ECL repository.

Users can also use the following terminal commands to print information on the EdgeNet nodes:

Commands to get information on the EdgeNet nodes:

```
kubectl get nodes,vpnpeers,namespaces,subnamespaces,slices -o wide --  
kubecfg <mycfg.cfg> -n <athena-rc>  
kubectl describe <node_name> --kubecfg <mycfg.cfg> -n <athena-rc>
```

Finally, users can run the provided demos, like so:

Instructions to run demos:

- Apply the services:
 - `kubectl apply -f cdn-service-example.yaml --kubecfg <mycfg.cfg> -n <athena-rc>`
 - `kubectl apply -f ping-me-example.yaml --kubecfg <mycfg.cfg> -n <athena-rc>`
- Connect to running pods:
 - `kubectl exec -it <cdn-server-podname> bash --kubecfg <mycfg.cfg> -n <athena-rc>`
- Get the log output:
 - `kubectl logs <ping-source-podname> --kubecfg <mycfg.cfg> -n <athena-rc>`
- For the ping-me-example case observe the pings from the pods through tcpdump:
 - `sudo tcpdump -i <eth0> icmp`

4.2.1.4 Local EdgeNet Cluster

For the proof-of-concept provided in the CODECO GitLabour setup, ATH relied on a recent EdgeNet update which made major improvements in the documentation and described the deployment of EdgeNet features/CRDs independently within a user's K8s cluster environment. During the setup effort, ATH contributed with minor bug fixes and improvements to the code from the official EdgeNet GitHub repository, enhancing the overall functionality of the infrastructure.

Based on the forked version of EdgeNet codebase⁴⁸, ATH was able to test the EdgeNet fundamental functionalities within our own K8s cluster environment. Specifically, we focused on features/CRDs such as: i) Multi-provider: involving node contribution to the EdgeNet, and ii) Multi-tenancy: encompassing user creation, tenant request/approval, role request/approval. The Multi-tenancy functionalities are designed to serve different cluster management purposes, enabling the utilization of a shared cluster environment.

⁴⁸<https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/experimentation-framework-and-demonstrations/edgenet-framework/-/tree/main/edgenet>



A step-by-step guide on how to create your local EdgeNet node based on our forked EdgeNet version is provided next. In the current document, we will refer to the following link⁴⁹ as CODECO-EdgeNet repository and to the following link⁵⁰ as CODECO-EdgeNet-node repository.

Prerequisites

For starters, users must:

- Have a K8s cluster running with *kubectl* installed.

Then, users can create their local EdgeNet cluster based on the next steps:

Instructions to run:

- 1) **Clone the CODECO-EdgeNet repository into the master node.**

```
git clone https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/experimentation-framework-and-demonstrations/edgenet-framework.git
```

- 2) In the Installation folder, users will find .sh and .yaml files.

```
ls edgenet-framework/installation
```

- 3) **Create a vpnpeer.yaml file in both master and worker nodes, as depicted below:**

```
apiVersion: networking.edgenet.io/v1alpha1
kind: VPNPeer
metadata:
  name: <> # Replace with the node name
spec:
  addressV4: <> # Replace with the edgenetmesh0 ipv4 address
  addressV6: <> # Replace with the edgenetmesh0 ipv6 address
  endpointAddress: <> # Replace with the public ip address of the node (e.g. use https://ipinfo.io)
  endpointPort: 51820
  publicKey: <> # Replace with the result of `echo "private key generated previously" | wg pubkey`
```

- 4) **Copy the vpnpeer.yaml of the worker node(s) into the master node.**

- 5) **Apply the copied yaml(s)**

```
kubectl apply -f vpnpeer.yaml
```

- 6) **Confirm the node(s) in the cluster.**

```
kubectl get vpnpeers
```

4.2.1.4.1 EdgeNet User & Namespace Setup – Multi-tenancy

It is feasible to create a new “regular” user with limited rights in our local EdgeNet cluster and, afterwards, assign them with roles and rights for various cluster management purposes.

- 1) **Create a regular user running the create-user script** which will create a username whose name is the provided email and a respective namespace. In addition, a private

⁴⁹<https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/experimentation-framework-and-demonstrations/edgenet-framework/-/tree/main/edgenet>

⁵⁰<https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/experimentation-framework-and-demonstrations/edgenet-framework/-/tree/main/node>



key for the user will be created as well as a role-binding requests, a user certification folder in the home directory and a configuration file.

```
- cd edgenet-framework/installation
- ./create-user.sh
```

- 2) One can **identify the different rights** of the “admin” and “regular user” by **changing the configuration file** located in the `kube` directory of the master node and trying to observe the cluster resources (e.g., pods).

- Rename the configuration files, or
- Add the `--kubeconfig <config-user@gmail.com> -n <namespace>` parameters when executing the commands.

- 3) We can **change the rights of the regular user** (e.g., to observe the cluster resources) by applying the appropriate yaml – e.g., make the regular user a tenant and **approving the request as admins**, adding "approved: true" under the "spec" field.

```
- kubectl create -f tenantrequest.yaml --kubeconfig <config-user@gmail.com> -n <namespace>
- kubectl edit tenantrequest.yaml
```

Note that when creating the tenant request regular users must use `create` and not `apply` as they only have “write” and not “read” privileges.

4.2.1.4.2 Node contributions – Multi-provider

In order to contribute a node to the local EdgeNet cluster, a user should:

- 1) **Copy** the `config-public` file found in the `.kube/` directory into the CODECO-EdgeNet repository `edgenet/configs/public_ehome.cfg`:

```
- cat .kube/config-public
- pico edgenet/configs/public_ehome.cfg
```

- 2) **Find the public key** which was generated on the **master node**.

```
- cat .ssh/id_rsa.pub
```

- 3) **Copy the public key into the CODECO-EdgeNet-node repository** in `vars/edgenet-codeco.yml` and replace the last line named `edgenet_ssh_public_key` so any node can be contributed and can join the cluster. The `edgenet-codeco.yml` is located under the `vars` directory and contains the information illustrated in the following code snippet:

```
# edgenet-kubernetes
containerd_version: 1.6.12
kubernetes_version: 1.23.17
edgenet_node_version: 1.0.17
#publicIPv4: 195.251.209.231
kubeconfig_url: https://raw.githubusercontent.com/swnuom/edgenet/main/configs/public_ehome.cfg
# edgenet-ssh
edgenet_ssh_public_key: ssh-rsa <> ansible-generated on athm3
```

- 4) **Establish a connection** to the node intended to be contributed to the cluster e.g.

```
- ssh user@<IP.address>
```

- 5) **Run the `start.sh` script** which installs Ansible and runs the node ansible-playbook on the target machines and automatically deploy an EdgeNet node.

```
- cd edgenet-framework/node
- ./start.sh
```

- 6) **Confirm** that the nodes joined the cluster (expected output shown below). **Establish a connection** to the master node.

```
- kubectl get nodes -o wide
```



4.2.1.5 Source code

The EdgeNet related assets are available in the CODECO Eclipse repository ([Experimentation Framework and Demonstrations->EdgeNet Framework](#)). In this GitLab subgroup, we have included detailed instructions on how to create and experiment within the EdgeNet testbed infrastructure as well as within a local EdgeNet cluster.

More specifically, we have released the developed demo examples created for CODECO and some .yaml files defining resources to enable the utilization of a shared cluster, under the 'demo' and 'tutorials' directories, respectively. The 'edgenet', 'node' and 'installation' directories contain files necessary for the setting up a local EdgeNet cluster, as described in Subsection 4.2.1.4.

To consolidate the comprehensibility of our contributions, we also provide specific complementary videos that demonstrate the explored processes and functionalities.

5 Continuous Integration, Testing, Deployment Preparation and Releasing

This section provides an overview on the CODECO methodology, environment and structure that is being used for continuous integration, system testing and deployment preparation and release, based on OSS practices.

The official CODECO repository is provided by partner Eclipse ([CODECO Eclipse GitLab](#)). However, for testing, integration of the overall framework, the consortium agreed in addition to consider the [offered ICOM infrastructure](#), which has the capability to support continuous integration with a high degree of automation, while it can be easily connected with GitLab Runners, hosted within ICOM facilities too.

As described in this section, the process of work is as follows:

- CODECO partners upload their OSS contributions (sub-component level) to the official CODECO repository, and in parallel, push the same code to the ICOM GitLab environment in order to enable the CI/CD pipelines. In the later stage of the project, the two repositories will be synchronized and therefore the partners will be able to push their code only to the public CODECO repository.
- SIE, coordinating SWM (currently undergoing an OSS process) uploads its code directly to the ICOM GitLab environment.
- The ICOM GitLab environment is used to support a continuous integration, assisting in a faster deployment of all CODECO components and interfaces.

The overall process is described in the next subsections, where more details for the CI/ CD procedures followed, are explained in ANNEX I.

5.1 Testing Methodology

CODECO adopted a continuous integration-system testing/deployment-release methodology, to allow for an early release of results in the form of software. This methodology consists of the following:

- **Feature – Unit Testing.** Unit testing involves testing individual internal CODECO sub-components by simulating or mocking all of the other components in the framework. This practice enables early detection of failures caused by changes in specific parts of the code or functions. Developers conduct unit testing locally in their development



environments, and these tests are also automatically invoked by the Continuous Integration (CI) server. The development teams create the necessary job definitions for use by the CI/CD server.

- **Interfacing – Integration Testing.** Integration testing verifies the proper functioning of a service and its features by examining the seamless interaction of subsystems, while simulating only other services. All CODECO components and sub-components need to successfully pass the integration tests specified before releasing a new software version. Development teams provide instructions on accessing their components' interfaces and functionalities for testing purposes, as well instruction on how these components can be deployed. Such tests triggered automatically by the Continuous Integration server for components with updated dependencies.
- **System testing,** which involves deploying all components and assessing fundamental features and user interactions. The purpose of system testing is to evaluate the compliance of the integrated system with the specified requirements. CODECO deployment and testing are also expected to be viable at a global scale by integrating CODECO with EdgeNet. In this way, CODECO assets will be deployed in K8s clusters comprising geographically-distributed nodes. Experimenters can explicitly define the requirements of the K8s cluster on which they desire to deploy and test CODECO components. Using the Selective Deployment feature (explained in Subsection 4.2), EdgeNet will ensure that the nodes selection is performed in such a way that the predefined requirements are met. In a later project stage, the system testing will be enhanced by implementing a designated controller, which will allow for the automated deployment of CODECO assets across the testbed infrastructure.

5.2 Deployment Preparation and Releasing

5.2.1 Deployment and CI/CD Methodology

As described and shown in Section 2, CODECO is being designed as a modular framework with several components (currently five), and each CODECO component consists of one or more sub-components.

Each CODECO sub-component is encapsulated in a container image to be deployed independently in the form of a K8s deployment object. In case it acts as a K8s controller, the deployment of the container image also requires the application of the corresponding CRDs. Containerisation allows applications to be packaged into containers, ensuring secure and isolated execution. This facilitates portability across different computing environments, regardless of the specific hardware and operating system used.

GitLab Runners have been set up by partner ICOM in the private CODECO GitLab repository, which is used exclusively for unit and integration testing, while interacting directly with the CI cluster. The code available in this private environment is first made available in the CODECO Eclipse repository and then pushed to the private one, as the CI Cluster is only connected to the private repo and at this stage of the project the two repositories (public and private) are not synchronised.

In addition, within each CODECO subcomponent, a Dockerfile is provided to build the Docker images of the CODECO services. Each CODECO subcomponent's Dockerfile is then used to automatically build the images within Gitlab CI. Developers can configure the appropriate gitlab-ci.yml files to automate the tasks of building, testing, packaging, and deploying artefacts on the integration testbed K8s cluster. If any tests or builds fail, the CI server notifies the relevant developer via email to resolve the issues. More information on this process is



provided in Appendix III. In addition, the GitLab container registry is used to share and rebuild container images and perform automated deployment.

Container orchestration is done based on K8s. The K8s deployment YAML configuration files (including Secret, Services, Volumes etc.) are managed by ICOM as the integration leader (WP3, T3.6) with the help of each component’s development team. All partners participate in the integration task (T3.6), with the exception of partner ECL.

Currently, when the code is uploaded (in each subcomponent) and all the tests are executed successfully, a merge request is sent by the contributor to the sub-component leader (task leader), to update the main branch of each project within CODECO public repo. At the end of each implementation cycle, a new tag is created, and a new code version, utilizing the tag, is published/ released, using only the code that is available in the “main” branch of each CODECO subcomponent. Moreover, at the end of each implementation cycle, the images of subcomponent are uploaded to CODECO DockerHub⁵¹. More details regarding the releasing version of each subcomponent are offered in the table presented in Annex II.

5.2.2 Integration Testbed Environment

CODECO envisions in its working plan the integration into EdgeNet (rf. to Section 4.2) as one of the environments that could provide support for large-scale, multi-cluster, multi-tenant features. CODECO also envisions to create, in intersection with EdgeNet as far as possible, a shared experimental infrastructure composed of different clusters provided by some partners; eventually some connections to EdgeNet and: a Cloud-based shared environment to assist in faster testing and performance validation. The experimental environments envisioned in CODECO are summarised in Table 9. This section provides information on the current status of the different integration environments.

Table 9: Status of the CODECO proposed experimental and integration framework.

Testbed type	Target	Status
Individual clusters, partner facilities	To allow individual partners to test CODECO in alignment with the CODECO use-cases	Achieved. Individual partners have provided their plans for setting up single cluster/multi-cluster environments, and which could be interconnected via a public infrastructure (e.g., EdgeNet, shared Cloud space).
ICOM private K8s cluster	To allow for integration and system tests of the overall CODECO framework since an early stage of the project	Achieved ICOM has provided the cluster and performed the required testing for the release of the CODECO sub-components (early release CODECO Basic Operation Toolkit v0.1)
Shared Cloud-space	Create a common environment for integration and system deployment, envisioning the need to support multi-cluster, multi-tenant environments.	Under development. Identification of the specific deployment environment to consider (nodes and respective features required) has been done; consultation to services is ongoing, coordinated by partner IBM
EdgeNet	Assist in the exploitation of CODECO, by integrating the framework into a worldwide testbed; take advantage of a realistic multi-cluster multi-tenant environment, interconnected over the Internet	Under development. Partners can already interconnect to EdgeNet, based on the experiments performed by ATH. Next step is to test interconnections via EdgeNet between CODECO partners that have plans to use EdgeNet (currently, ATH and FOR).

⁵¹ <https://hub.docker.com/repositories/hecodeco>



5.2.2.1 ICOM Private K8s Cluster

At the current stage of the project, ICOM K8s cluster is used for integration tests. As mentioned in the previous subsection, GitLab Runners running at integration testbed K8s cluster provided by ICOM, take charge of executing the CI/CD pipelines, as they described in the “gitlab-ci.yml” file, to automate the procedures of testing, building, packaging and deployment. For the resource monitoring of the cluster, K8s Dashboard, Prometheus⁵² and Grafana⁵³, are configured and utilized, providing valuable insights on the health status of each CODECO component.

In detail, ICOM K8s cluster is consist of a single K8s node with the following characteristics, ensuring that the hardware requirements of CODECO components can be fulfilled:

- **Operating System:** Ubuntu 20.04.3 LTS
- **CPU:** 2 x Intel(R) Xeon(R) Silver 4214 CPU @ 2.20GHz
- **Memory (RAM):** 128GB
- **Hard disks:** Total Drives: 3 x 960GB, RAID 5

5.2.2.2 CODECO Cloud-based Shared Environment

Envisioning large-scale testing and also federated environments, the CODECO consortium is building a Cloud-based shared space to assist further development in multi-cluster environments. This environment will be used for integration testing and also for demonstrations (third year of CODECO). The basic requirements for this shared space are as follows:

- Multi-cluster environment (minimum of three types of clusters), with OCM/Openshift.
- Each cluster envisioned to have 1-3 control plane nodes; at least 2 worker nodes.
- Cloud-based service must ensure that nodes are in Europe.
- Possibility to interconnect local clusters supported by CODECO partners.
- Possibility to interconnect with EdgeNet.

The three types of clusters envisioned in this environment and summarised in Table 10 are as follows:

- Cluster A (**core** cluster), based on single extremely powerful node.
- Cluster B (**heterogeneous** cluster), based on a mix of nodes.
- Cluster C (**constrained** cluster), based on many small nodes.

Table 10: First proposed structure of cluster types to dimension the CODECO shared-Cloud space.

Cluster type	Nodes	Total Cores	Expected	Total Memory
Core	One extremely large, capable of fitting the OCM Hub and one managed cluster	64		512
Heterogeneous	One large (master nodes); four medium nodes (worker nodes with K0s, K3s, etc)	32		96
Constrained	One medium (master nodes); 14 small nodes (workers with K0s, K3s, etc).	18		44

The type of node is being dimensioned to ensure heterogeneity. For this, CODECO considers a definition of small, medium, and large nodes as provided in Table 11.

⁵² <https://prometheus.io/>

⁵³ <https://grafana.com/>



Table 11: Type of nodes envisioned for the CODECO Cloud-based multi-cluster environment dimensioning.

Node Type	CPU cores	RAM (GB)
Small	1-2	2-8
Medium	4-8	16-32
Large	16	32

Furthermore, as CODECO defined the use of heterogeneous K8s technologies, respecting the principles of OSS use, Table 12 summarizes minimum system requirements that allow to better dimension the multi-cluster environment.

Table 12: Examples for minimum system requirements, different K8s technologies.

K8s technology	K8s node role	Minimum System Requirements				Node Type
		OSS	CPU/vCPU	RAM	Disk (GB)	
OCM	Master	Red Hat Enterprise Linux (RHEL) 7.5, or RHEL Atomic Host 7.4.5 or later	4 CPU; 4 vCPU	16 GB	42	Medium
	Worker		1 vCPU	8 GB	32	Medium
	External etcd				20	Medium
	Ansible controller			75 MB		Small
Microshift		Red Hat Enterprise Linux (RHEL) Extended Update Support (EUS) (9.2 or later)	- x86_64 or aarch64 CPU architecture - 2 CPU cores	2 GB	2	Small
K3s		- RHEL9, Ubuntu, openSUSE Leap, Oracle Linux, Rocky Linux and SLES - RedHat/CentOS Enterprise Linux and Raspberry Pi OS with additional setup	- x86_64 or armhf or arm64/aarch64 or s390x CPU architecture - 1 CPU core	512 MB		Small

5.2.2.3 CODECO – EdgeNet Environment

As already presented earlier, CODECO is expected to support also large-scale tests through its integration with EdgeNet. In this context, an ATH-owned node that can be selected for deployment has been contributed to the EdgeNet testbed and has the following characteristics:

- **Operating System:** Ubuntu 22.04.3 LTS
- **CPUs:** 4
- **Memory (RAM):** 4 GB
- **Ephemeral storage:** 35.8 GB



6 Summary: Current Status and Next Steps

The CODECO Deliverable D11 presents the initial release of the CODECO toolkit's Basic Operation Toolkit, which offers developers the opportunity to begin experimenting with certain CODECO components.

In detail, D11 is the outcome of the efforts carried out in T3.6 (Open Edge-Cloud Toolkit Development), which is focused on implementing and testing the CODECO toolkit, using software components that are being developed across Tasks 3.1 to 3.5, based on the system and CODECO components architecture presented in D9 [1], released in M9. Therefore, D11 is offered as an interim milestone within the CODECO project, developed as part of CODECO WP3 (CODECO Basic Operation and Toolkit v1.0).

The work presented in D11 aimed to facilitate the development, integration, and deployment of the CODECO framework following an incremental approach, by presenting tools adopted to support the defined implementation workflow and integration framework, coding guidelines, installation instructions, as well as inputs and outputs of each CODECO sub-component, ensuring the coordination of the work between consortium members belonging to different organizations and external developers.

Within D11, the implementation details of each CODECO subcomponent that is released in the initial implementation cycle, to assist the reader in understanding the software developed so far, are presented.

The next steps are as follows:

- concerning the CODECO OSS framework:
 - An early release of the ACM component has been provided, along with the current version of the CODECO Application Model, still under discussion.
 - MDM APIs and their interconnection with a knowledge graph have been provided.
 - PDLC sub-components have been released. For the specific case of PDLC-CA, the current early release allows the user DEV to play with different performance targets, and to explore the possibility to consider aggregated costs for nodes, For PDLC-DL, three independent Decl models that perform different objectives are introduced, accompanied by the initial version of MLOps procedures that will be followed within PDLC-DL sub-component.
 - Within NetMA component, the first version of Secure Connectivity (L2S-M) subcomponent is described, while Network Exposure, MEC Enablement and Network State Forecasting subcomponents offered their initial implementation details.
 - For SWM, a confidential document is released, detailing the complete specifications and software associated with the CODECO component SWM, which is presently undergoing a process to become open source.
- The definition of a first sub-set of metrics (network, computational, data observability) that can define a cluster and assist in scalable, privacy preserving, and efficient data aggregation have been provided at an operational level, and integrated into the CODECO synthetic data generator.
- In regard to CODECO experimentation:
 - Improvements to EdgeNet, to allow the integration of CODECO in multi-cluster, multi-tenant environments have been made available.
 - The initial design of the CODECO shared Cloud experimental environment have been provided.



- Concerning the automated and agile setup, deployment, testing, and releasing of CODECO, the current methodology has been discussed.

The open source release parts of the CODECO Basic Operation Toolkit can be found in the CODECO Eclipse Gitlab⁵⁴, while the container images of the CODECO sub-components are available in the CODECO DockerHub⁵⁵.

Finally, upon the completion of WP3 in M18, a new updated deliverable introducing the second (final) release of the CODECO toolkit for a single cluster environment will be released.

⁵⁴ <https://gitlab.eclipse.org/eclipse-research-labs/codeco-project>
⁵⁵ <https://hub.docker.com/repositories/hecodeco>



7 References

- [1] R. C. Sofia (Ed.), H. Müller, J. Solomon, R. Touma, L. Garces Erice, L. Contreras Murillo, D. Remon, A. Espinosa, J. Soldatos, N. Psaromanolakis, L. MAmathas, I. Kapetanidou, V. Tsaoussidis, J. Martrat, I. Mariscal, P. Urbanetz, D. Dykemann, V. Theodorou, S. Ferlin-Reiter, E. araskevoulakou, P. Karamolegkos. *CODECO Deliverable D9 -Technological Guidelines, Reference Architecture, and Initial Open source Ecosystem Design v1.0*. DOI 10.5281/zenodo.8143859. June 2023.
- [2] S. Shalunov, W. Room, R. Woundy, S. Previdi, S. Kiesel, R. Alimi, R. Penno, R. Y. Yang. Application-Layer Traffic Optimization (ALTO) Protocol, IETF RFC 7285, Sep. 2014. doi: 10.17487/RFC7285.
- [3] Zhao, L., Song, Y., Zhang, C., Liu, Y., Wang, P., Lin, T., ... & Li, H. (2019). T-gcn: A temporal graph convolutional network for traffic prediction. *IEEE transactions on intelligent transportation systems*, 21(9), 3848-3858.
- [4] Bai, J., Zhu, J., Song, Y., Zhao, L., Hou, Z., Du, R., & Li, H. (2021). A3t-gcn: Attention temporal graph convolutional network for traffic forecasting. *ISPRS International Journal of Geo-Information*, 10(7). <https://doi.org/10.3390/ijgi10070485>.
- [5] Gonzalez, L. F., Vidal, I., Valera, F., Martin, R., & Artalejo, D. (2023). A Link-Layer Virtual Networking Solution for Cloud-Native Network Function Virtualisation Ecosystems: L2S-M. *Future Internet*, 15(8), 274.



Annex I – CODECO Cross-Layer Metrics

This annex provides an aggregated perspective of the minimum sub-set of metrics for:

- The definition of a CODECO Application Model (QoS, QoE preferences from user DEV concerning the application to be deployed across far Edge-Cloud), as
- The minimum subset of metrics currently expected to be collected by different components (NetMA, MDM, ACM).

These metrics are currently the basis for the CODECO data generator and also in use as input to other CODECO components, such as PDLC.

Table 13: CODECO Application Model Attributes, spec, and status.

ID (attribute)	Description	Type	Min-Max	How it is handled
spec				
userid	Identification of the user DEV setting up an application.	Integer	-	Available in K8s, but should be provided by other CODECO components by ACM
usergroup		Integer	-	Available in K8s, but should be provided by other CODECO components by ACM
QoS	Preferred QoS level. A scale 1-5 is provided to the user and converted to a desired QoS model, e.g., Diffserv codepoints.	Integer	1,5	New
cpu	Desired CPU usage (%).	Integer	0,100	Taken from K8s
mem	Desired memory	Integer		Taken from K8s
failure_tolerance	Desired tolerance to infrastructure failures. The user selects a %	String	0,100	To be provided by CODECO. Failure_tolerance is currently computed in PDLC (sub-component PDLC-CA) as an aggregate metric that considers parameters from the computational infrastructure and from the networking infrastructure (NetMA)
energy_expenditure	Maximum desired level of energy expenditure for the overall K8s infrastructure serving an application group.	String	0,100	Energy expenditure of the infrastructure is provided by PDLC (PDLC_CA) based on node expenditure (ACM, Prometheus) and link/path expenditure
security_level	Desired level of security based on a scale of 1-5	Integer	1,5	To be provided by CODECO, ACM.
compliance_level	expected level of compliance, based on a scale of 1-5	Integer	1,5	Provided by MDM
Input_data_size	Expected volume of input data for the application. 0 implies no input data. Expressed in the same units as the mem.	Integer	0,	Provided by MDM
Output_data_size	Expected volume of output data for the application. 0 implies no output data. Expressed in the same units as the mem.	Integer	0,	Provided by MDM
status				
cpu	status on CPU usage for the application/applicationgroup			Provided by ACM Provided by ACM
mem	status on mem usage for the application/applicationgroup			Provided by PDLC to ACM, based on data from NetMA, ACM, MDM
failure_tolerance	status on current infrastructure failures			Provided by PDLC to ACM, based on data from NetMA, ACM, MDM
energy_expenditure	status on energy used to serve the application			Provided by PDLC to ACM, based on data from NetMA, ACM, MDM
portability	status on the portability			Provided by PDLC to ACM,

ID (attribute)	Description	Type	Min-Max	How it is handled
	effectiveness			based on data from NetMA, ACM, MDM

Table 14: Minimum subset of CODECO user parameters to be collected in ACM and used in the orchestration.

ID (attribute)	Description	Type	CODECO components relying on the metrics
cpu	CPU represents compute processing available on nodes and is specified in units of K8s CPUs. Limits and requests for CPU resources are measured in cpu units. In K8s, 1 CPU unit is equivalent to 1 physical CPU core, or 1 virtual core, depending on whether the node is a physical host or a virtual machine running inside a physical machine. The Linux Kernel provides the CPU status	integer	All, from K8s/Prometheus
mem	MEM represents compute memory available on nodes. Limits and requests for memory are measured in bytes. You can express memory as a plain integer or as a fixed-point number using one of these quantity suffixes: E, P, T, G, M, k. Note: Mebibyte (Mi): 1 MiB = 1.048 MB	integer	All, from K8s/Prometheus
node_failure	node failures averaged over a specific time window (Exponential moving average)	integer	PDLC
node_energy	Energy consumed by a node	string	PDLC

Table 15: Minimum subset of CODECO data observability parameters to be collected in MDM and used in the orchestration.

ID (attribute)	Description	Type	CODECO components relying on the metrics
node_name	identifier of the node in K8s	string	All
freshness	Measuring the healthiness of a node, to whether it is able to ensure that new or updated data of its existing workload is processed in a timely fashion and that the dataset grows in stable specified size.	string	PDLC, SWM
compliance	Data Compliance refers to the desired adherence to regulations and policies derived from geographical zones regulation, etc	string	PDLC, NetMA, ACM
portability	How effective the application workload performs in the current node in comparison to a former node.	string	PDLC, ACM, SWM
input_data_size	The initial size of an application	string	PDLC



ID (attribute)	Description	Type	CODECO components relying on the metrics
	data set (transient, non-transient)		
output_data_size	The current size of an application data set (transient, non-transient)	string	PDLC

Table 16: Minimum subset of CODECO networking parameters to be collected in NetMA and used in the orchestration.

ID (attribute)	Description	Type	CODECO components relying on the metrics
node_name	Identifier of the node in K8s	string	All
link_id	Identifier of the link of a node	string	SWM, PDLC
link_failure	Number of failures of a link	integer	PDLC, used in PDLC to compute a resilience factor
node_net_failure	Sum of elink_failure and link_failure	integer	PDLC-CA used in PDLC to compute a resilience factor
ebw	Egress bandwidth for a node - sum of bw for all Egress links on that node	string	PDLC, expected to be used in target profile factors such as greenness
ibw	Ingress bandwidth for a node - sum of bw for all ingress links on that node	string	SWM
uid_visits	Number of visits from a UE (or UID) to the node. May be relevant in case of mobile environments, UID visits to a node, for instance. Can be based on nr of visits x duration of visit.	string	PDLC, e.g., for computing node betweenness
geolocation	location of the UE (geo-coordinates translated to string)	string	far Edge information, may be useful for Edge selection (e.g., CDN use-case); PDLC and MDM may use this.
zone	Associated to an inter or intra-routing approach; inter, AS	string	SWM may use; PDLC, MDM
node_degree	In a network, the degree of a node is defined as the sum of all edges connected to it (later, ingress and egress links)	integer	PDLC requires for betweenness, congestion and eventually resilience factor computation
latency	average latency derived from elink latency; can be based on RTT	string	The network awareness needs to consider also link metrics, to allow for an adequate deployment and network adaptation
path_length	Number of hops traversed; Average Shortest Path Length	integer	PDLC may use to compute latency target; SWM
link_energy	The average expenditure of energy over a time window (EMA) for a specific link	string	PDLC, greenness

Annex II – Release Versioning

Table 17 provides a list of the current CODECO sub-components, URLs for open source code, and images in Docker. The notation in the Table is as follows:

- C: CODECO component acronym.
- SC: Sub-component abbreviation.
- OSS: GitLab Repo URL for the open-source code.
- DH URL: URL for the respective Docker Hub image.
- Tag: Image tag.

Table 17. CODECO subcomponents & Assisted Developments (Open Source) releasing versions.

C	SC	URL	OSS	DH URL	Tag
ACM	ACM Operator	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/acm	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/acm/-/releases/v1.0.0	https://hub.docker.com/repository/docker/hecodeco/acm-appoperator	hecodeco/acm-appoperator:1.0
PDLC	PDL C-CA	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/context-awareness/pdlc-pp	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/context-awareness/pdlc-pp/-/releases/v1.0.0	https://hub.docker.com/repository/docker/hecodeco/pdlc-ca-pp/general	hecodeco/pdlc-ca-pp:1.0
		PDL C-DL	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/decentralised-learning/model-selection-and-training/-/tree/main/MARL_A3C-UPM?ref_type=heads	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/decentralised-learning/model-selection-and-training/-/releases/v1.0.0	https://hub.docker.com/repository/docker/hecodeco/pdlc-dl-gnns-stgcn/general
	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/decentralised-learning/model-selection-and-training/-/tree/main/MARL_A3C-UPM?ref_type=heads		https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/decentralised-learning/model-selection-and-training/-/releases/v1.0.0	https://hub.docker.com/repository/docker/hecodeco/pdlc-dl-gnns-a3tgcn/general	hecodeco/pdlc-dl-gnns-a3tgcn:1.0
	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/decentralised-learning/model-selection-and-training/-/tree/main/MARL_A3C-UPM?ref_type=heads		https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/decentralised-learning/model-selection-and-training/-/releases/v1.0.0	https://hub.docker.com/repository/docker/hecodeco/pdlc-i2cat-rlmodel/general	hecodeco/pdlc-i2cat-rlmodel:1.0
	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/decentralised-learning/model-selection-and-training/-/tree/main/MARL_A3C-UPM?ref_type=heads	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/privacy-preserving-decentralised-learning-and-context-awareness-pdlc/decentralised-learning/model-selection-and-training/-/releases/v1.0.0	https://hub.docker.com/repository/docker/hecodeco/pdlc-marl-a3c	hecodeco/pdlc-marl-a3c:1.0	

C	SC	URL	OSS	DH URL	Tag
	MLOPs	https://colab-repo.intracom-telecom.com/colab-projects/hecodeco/pdlc/pdlc-dl/mlops	https://colab-repo.intracom-telecom.com/colab-projects/hecodeco/pdlc/pdlc-dl/mlops/-/releases/v1.0.0		
NetMA	ME	https://gitlab.eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/mec-enablement	https://gitlab.eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/mec-enablement/-/releases/v1.0.0	https://hub.docker.com/repository/docker/hecodeco/netma-mec-enablement	hecodeco/netma-mec-enablement:1.0
	NE	https://gitlab.eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/network-exposure	https://gitlab.eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/network-exposure/-/releases/v1.0.0		
	L2S-M	https://gitlab.eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/secure-connectivity	https://gitlab.eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/secure-connectivity/-/releases/v1.0.0	https://hub.docker.com/repository/docker/hecodeco/netma-seconn-l2sm-operator/general	hecodeco/netma-seconn-l2sm-operator:1.0
				https://hub.docker.com/repository/docker/hecodeco/netma-seconn-l2sm-ovs/general	hecodeco/netma-seconn-l2sm-ovs:1.0
https://hub.docker.com/repository/docker/hecodeco/netma-seconn-l2sm-lpm/general				hecodeco/netma-seconn-l2sm-lpm:1.0	
NSM	https://gitlab.eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/network-state-management	https://gitlab.eclipse-research-labs/codeco-project/network-management-and-adaptation-netma/network-state-management/-/releases/v1.0.0	https://hub.docker.com/repository/docker/hecodeco/netma-netmanagement/general	hecodeco/netma-netmanagement:1.0	



C	SC	URL	OSS	DH URL	Tag
MDM	MDM APIs	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadatan-manager-mdm/mdm-api	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadatan-manager-mdm/mdm-api/-/releases/v1.0.0	https://hub.docker.com/repository/docker/hecodeco/mdm-neo4j-ctrl/general	hecodeco/mdm-api:1.0
				https://hub.docker.com/repository/docker/hecodeco/mdm-neo4j-ctrl/general	hecodeco/mdm-neo4j-ctrl:1.0
	Graph DB	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadatan-manager-mdm/graphdb	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/metadatan-manager-mdm/graphdb/-/releases/v1.0.0		
Assisted Development	Synthetic Data Generator	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/experimentation-framework-and-demonstrations/data-generators-and-datasets/synthetic-data-generator	https://gitlab.eclipse.org/eclipse-research-labs/codeco-project/experimentation-framework-and-demonstrations/data-generators-and-datasets/synthetic-data-generator/-/releases/v1.0.0	https://hub.docker.com/repository/docker/hecodeco/synthetic-data-generator-acm/general	hecodeco/synthetic-data-generator-acm:1.0
				https://hub.docker.com/repository/docker/hecodeco/synthetic-data-generator-mdm/general	hecodeco/synthetic-data-generator-mdm:1.0
				https://hub.docker.com/repository/docker/hecodeco/synthetic-data-generator-netma/general	hecodeco/synthetic-data-generator-netma:1.0

Annex III – Automating software builds, testing, packaging, and deployment

To enable this level of automation, specific pre-requisites must be met, detailed as follows:

- The presence of a `.gitlab-ci.yml` file at the project's root.
- Unit tests located within a valid project path for the testing stage.
- A Dockerfile for appropriate artifact packaging in the form of an image.
- A kubeconfig file as an Environmental Variable, referring to the integration testbed k8s cluster along with a k8s deployment configuration file, to facilitate image deployment on the Kubernetes cluster.

The next code snippet presents an example of the `gitlab-ci.yml` file designed for a Python codebase, specifically named "**ca_controller**", which acts as a Kubernetes controller for some defined CRs. This file, demonstrating the necessary stage configurations, can serve as a useful starting point for developers working on their own codebases.

CI/CD Configuration file

```
# .gitlab-ci.yml
variables:
  IMAGE_NAME: $CI_REGISTRY_IMAGE/ca
  IMAGE_TAG: $CI_COMMIT_BRANCH.$CI_COMMIT_SHA
  CONTAINER_REGISTRY: colab-repo.intracom-telecom.com:5050

stages:
- test
- build
- prepare_deployment
- deploy

default:
  image: python:3.8
  before_script:
  - apt-get update && apt-get install gettext-base

test:
  stage: test
  only:
  - test
  tags:
  - internal-cluster
  script:
  - cd ca_controller
  - python3 -m pytest
```



```
build:
  stage: build
  tags:
  - internal-cluster
  image: docker:20.10.24
  only:
  - main
  services:
  - name: docker:20.10.24-dind
  variables:
  DOCKER_DRIVER: overlay2
  DOCKER_TLS_CERTDIR: ""
  DOCKER_HOST: tcp://docker:2375

before_script:
- docker login $CONTAINER_REGISTRY -u $REGISTRY_USER -p $CR_TOKEN

script:
- docker build -t $IMAGE_NAME:$IMAGE_TAG .
- docker push $IMAGE_NAME:$IMAGE_TAG

artifacts:
  paths:
  - $CI_PROJECT_DIR
  expire_in: 1 week

prepare_deployment_files:
  stage: prepare_deployment
  tags:
  - internal-cluster
  only:
  - main
  script:
  - cd ca_controller
  - export IMAGE_CON=$IMAGE_NAME:$IMAGE_TAG
  - envsubst < ca-deployment-ci.yaml > ../ca-deployment-ci.yaml
  artifacts:
  paths:
  - ca-deployment-ci.yaml
  expire_in: 1 day
```



```
deploy:
stage: deploy
tags:
- internal-cluster
only:
- main
image:
name: bitnami/kubectl:latest
entrypoint: ['']
before_script:
- echo $KUBECONFIG64 | base64 --decode > /.kube/config
script:
- kubectl apply -f ca_controller/ca-crd.yaml
- kubectl apply -f ca_controller/ca-role.yaml -n codeco-pdlc
- kubectl apply -f ca_controller/ca-role-binding.yaml -n codeco-pdlc
- kubectl apply -f ca-deployment-ci.yaml -n codeco-pdlc
```

At the top of the file, three global variables are defined and made available to all the jobs, while also the four desired stages are defined, namely: “test”, “build”, “prepare_deployment” and “deploy”. The rest of the file defines jobs for each stage.

Each job has an intuitive non-standard name (e.g., build) and defines multiple configurations required to complete successfully. In most cases, jobs define a pipeline stage during which they will be invoked for execution, a base image to use for the job fulfilment, as well as a “scripts” tag with the actual execution logic.

For the case of the test job, the stage is set to “test”, the base image to python:3.8 and the scripts tag executes the “python3 -m pytest” command to run all project unit tests.

Once the test stage is completed, jobs that are part of the “build” stage will be executed. That is, build job, which uses a ‘docker:20.10.24’ as the base image to containerize the application. Any artifacts built during the “build” stage are available in the next stages of the pipeline. Then, the build job pushes the image to the project’s private image registry. A set of global variables created using environmental variables - defined in Gitlab’s project settings, under the CI/CD settings - belonging to the codeco-k8s-cluster environment are passed to the docker executor to successfully complete the job.

The next job of the file is the “prepare_deployment_files”, part of the “prepare_deploy” stage, utilizes a python:3.8 base image to overwrite the deployment file with the proper tag referring to the new image created in the previous build job, and passes the updated file in the next stages of the pipeline.

The last job of the file deploy, part of the “deploy” stage, utilizes a kubectl base image to connect through the Kubernetes API to the integration testbed cluster and deploy or update the resource defined in the “deployment.yaml” file created in the previous stages, as well as the CRDs, Roles and Roles Bindings files.

