Deliverable D2.4

Specification of the Cloud Security Certification Language-v2

Editor(s):	Michela Fazzolari, Marinella Petrocchi
Responsible Partner:	Consiglio Nazionale delle Ricerche (CNR)
Status-Version:	Final – v1.0
Date:	31.10.2022
Distribution level (CO, PU):	PU

Project Number:	952633
Project Title:	MEDINA

Title of Deliverable:	Specification of the Cloud Security Certification Language – v2
Due Date of Delivery to the EC	31.10.2022

Work package responsible for the Deliverable:	WP2 - Certification Metrics and Specification Languages
Editor(s):	Michela Fazzolari, Marinella Petrocchi (CNR)
Contributor(s):	Michela Fazzolari (CNR), Marinella Petrocchi (CNR), Patrizia Ciampoli (HPE), Debora Benedetto (HPE)
Reviewer(s):	Juncal Alonso, Cristina Martínez (TECNALIA)
Approved by:	All Partners
Recommended/mandatory readers:	WP2, WP3, WP4, WP5, WP6

Abstract:	This is the second of the three deliverables resulting from Task 2.3, Task 2.4 and Task 2.5. This set of deliverables will present the definition and implementation of the Cloud Certification Language which encompasses three major phases: 1) the encoding of requirements of cloud certification schemas – written in natural language -- in a Controlled Natural Language (CNL), so called MEDINA CNL; 2) the editing of the requirements in MEDINA CNL through an editor tool; 3) the mapping of the CNL requirements to a domain specific language (DSL).
Keyword List:	MEDINA Cloud Certification Language, MEDINA CNL, CNL Editor, Domain Specific Language, DSL Mapper, MEDINA Ontology
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/
Disclaimer	This document reflects only the author's views and neither Agency nor the Commission are responsible for any use that may be made of the information contained therein

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	26.07.2022	Creation and tentative ToC	Michela Fazzolari CNR
V0.2	31.08.2022	ToC approved by partners	ALL
v0.3	19.09.2022	Added first version of Section 2	Michela Fazzolari CNR
v0.4	27.09.2022	Final version of Section 2 and Section 3	Michela Fazzolari CNR
v0.5	30.09.2022	Added Section 5	Michela Fazzolari CNR
v0.6	03.10.2022	Added Executive summary, Section 1 and 2	Michela Fazzolari CNR
v0.7	05.10.2022	Document review	Marinella Petrocchi CNR
v0.75	06.10.2022	Document review	Michela Fazzolari CNR
v0.8	12.10.2022	Added Section 4	Debora Benedetto HPE
v0.85	13.10.2022	Modified Section 4	Patrizia Ciampoli HPE
v0.9	19.10.2022	Changed name of the Catalogue	Michela Fazzolari CNR
v0.95	20.10.2022	QA review	Juncal Alonso TECNALIA
v0.96	24.10.2022	Modified Section 4	Patrizia Ciampoli HPE
v0.97	25.10.2022	Addressed all comments received in the internal QA review	Michela Fazzolari Marinella Petrocchi CNR
v1.0	31.10.2022	Ready for submission	Cristina Martinez TECNALIA

(cc) BY-SA

Table of contents

Terms and abbreviations 7
Executive Summary 9
1 Introduction 11
1.1 About this deliverable 11
1.2 Document structure 11
1.3 Updates from D2.3 12
2 Cloud Security Certification Language Tool High-level Architecture 13
2.1 Architecture 13
2.2 Sequence diagram 16
2.3 Coverage of requirements 18
3 NL2CNL Translator 19
3.1 Implementation 19
3.2 Delivery and usage 26
3.3 Advancements within MEDINA 28
3.4 Limitations and future work 28
4 CNL Editor 29
4.1 Implementation 29
4.2 Delivery and usage 36
4.3 Advancements within MEDINA 40
4.4 Limitations and future work 41
5 DSL Mapper 42
5.1 Implementation 42
5.2 Technical description 46
5.3 Delivery and usage 49
5.4 Advancements within MEDINA 50
5.5 Limitations and future work 50
6 Conclusions 52
7 References 53
APPENDIX A: The Cloud Certification Language 57
1. Motivation 57
2. Methodology 57
APPENDIX B: Patterns and Controlled Natural Languages for Requirements specifications 58
1. Patterns 58
2. Controlled Natural Languages 59
3. CNLs for expressing policies for secure data management 60
4. MEDINA CNL 62

CC BY-SA

APPENDIX C: From NL to CNL TOMs 65
1. Metric association 65
2. CNL translations 71
APPENDIX D: MEDINA Vocabularies and Ontologies 73
1. Background: Taxonomies and Ontologies 73
2. Editor Ontology 73
3. Cloud Resource Security Ontology 75

List of tables

TABLE 1. OVERVIEW OF DELIVERABLE UPDATES WITH RESPECT TO D2.3 12
TABLE 2. DESCRIPTION OF THE GENERAL WORKFLOW WF3, WHICH INVOLVES THE CLOUD SECURITY CERTIFICATION LANGUAGE COMPONENTS IN SOME STEPS 16
TABLE 3. EXPECTED COVERAGE OF FUNCTIONAL REQUIREMENTS FOR THE CLOUD SECURITY CERTIFICATION LANGUAGE. 18
TABLE 4. LIST OF AVAILABLE ENDPOINTS FOR THE NL2CNL TRANSLATOR COMPONENT 24
TABLE 5. MOST IMPORTANT FILES AND FOLDERS IMPLEMENTING THE NL2CNL TRANSLATOR 26
TABLE 6. LIST OF AVAILABLE ENDPOINTS FOR THE CNL EDITOR COMPONENT 35
TABLE 7. LIST OF AVAILABLE ENDPOINTS FOR THE DSL MAPPER COMPONENT 47
TABLE 8. MOST IMPORTANT FILES AND FOLDERS IMPLEMENTING THE DSL MAPPER 49

List of figures

FIGURE 1. BUILDING BLOCKS VIEW OF THE MEDINA FRAMEWORK (SOURCE: D5.2 [6]) 14
FIGURE 2. ARCHITECTURE OF THE COMPONENTS INVOLVED IN THE CLOUD SECURITY CERTIFICATION LANGUAGE 15
FIGURE 3. SEQUENCE DIAGRAM DESCRIBING THE INTERACTION AMONG THE CLOUD SECURITY CERTIFICATION LANGUAGE COMPONENTS 17
FIGURE 4. POSITION OF THE NL2CNL TRANSLATOR WITHIN THE MEDINA ARCHITECTURE (SOURCE: D5.2 [6]) 22
FIGURE 5. OVERVIEW OF THE NL2CNL TRANSLATOR ARCHITECTURE 23
FIGURE 6. POSITION OF THE CNL EDITOR WITHIN THE MEDINA ARCHITECTURE (SOURCE: D5.2 [6]) 32
FIGURE 7. OVERVIEW OF THE CNL EDITOR ARCHITECTURE 33
FIGURE 8. SCREENSHOT OF A REO OBJECT AS IT APPEARS IN THE CNL EDITOR GUI 34
FIGURE 9. CNL EDITOR FOLDERS' STRUCTURE 36
FIGURE 10. EXAMPLE: CNL STORE API KUBERNETES MANIFESTS 37
FIGURE 11. ACCESSIBILITY OF THE CNL EDITOR FROM THE MEDINA UI 38
FIGURE 12. LIST OF REOS AVAILABLE FOR THE USER POLICYEXPERT IN THE CNL EDITOR 38
FIGURE 13. LIST OF AVAILABLE OPERATIONS FOR A SELECTED REO IN THE CNL EDITOR. 38
FIGURE 14. EDITING A REO OBJECT IN THE CNL EDITOR 39
FIGURE 15. CNL EDITOR APIS 40
FIGURE 16. POSITION OF THE DSL MAPPER WITHIN THE MEDINA ARCHITECTURE (SOURCE D5.2 [6]) 45
FIGURE 17. OVERVIEW OF THE DSL MAPPER ARCHITECTURE 46
FIGURE 18. OPERATIONAL SEMANTICS FOR THE COMPOSITE AUTHORIZATION FRAGMENT, WHERE THE SYMMETRIC RULE FOR (;) IS OMITTED (SOURCE: UNPUBLISHED MANUSCRIPT, PETROCCHI M AND MATTEUCCI I.) 62
FIGURE 19. FROM NATURAL LANGUAGE TO CONTROLLED NATURAL LANGUAGE: SIMPLIFIED OVERVIEW (SOURCE: MEDINA'S OWN CONTRIBUTION) 65

Figure 20. Feature computation workflow (source: MEDINA's own contribution) 67
Figure 21. Recommender system workflow (source: MEDINA's own contribution) 67
Figure 22. Plot of requirements and metrics using the first two components of the feature vectors, down-projected using TSNE, PCA and Truncated SVD respectively (source: MEDINA's own contribution) 69
Figure 23. Prototypical results for EUCS requirement AM-01.6, optimal results on rank 1 and 2 70
Figure 24. Prototypical results for EUCS requirement AM-03.6, results on rank 7 and 8 70
Figure 25. Prototypical results for EUCS requirement IM-03.4, no results 71
Figure 26. CNL Editor Vocabulary Structure Example 74
Figure 27. CNL Editor Vocabulary Metric "BackupEncriptionEnabled" 74
Figure 28. CNL Editor Vocabulary TargetValueType "Boolean" 75
Figure 29. The cloud resource taxonomy which classifies cloud resources according to their functional purpose, like compute, storage, and networking 77
Figure 30. An excerpt from the CRSO 78
Figure 31. The security property taxonomy which classifies security properties according to their targeted STRIDE-based goal 79

© MEDINA Consortium
(cc) BY-SA

Terms and abbreviations

API	Application Programming Interface
BNF	Backus-Naur Form
CCL	Cloud Certification Language
CNL	Controlled Natural Language
CNL4DSA	Controlled Natural Language for Data Sharing Agreement
CRSO	Cloud Resource Security Ontology
CRUD	Create, Read, Update, Delete
CSA or EU CSA	Cybersecurity Act
CSP	Cloud Service Provider
DB	Data Base
DCG	Discounted Cumulative Gain
DoA	Description of Action
DSL	Domain Specific Language
EC	European Commission
EUCS	European Cybersecurity Certification Scheme for Cloud Services
GA	Grant Agreement to the project
GUI	Graphical User Interface
GWT	Google Web Toolkit
IaaS	Infrastructure as a Service
ICT	Information Communications Technology
IDCG	Ideal Discounted Cumulative Gain
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MTS	Modal Transition System
nDCG	Normalized Discounted Cumulative Gain
NL	Natural Language
NLP	Natural Language Processing
NL2CNL	Natural Language To Controlled Natural Language
OPA	Open Policy Engine
P@k	Precision at k
PaaS	Platform as a Service
PCA	Principal Component Analysis
RDF	Resource Description Framework
REO	Requirement & Obligations
REST	Representational State Transfer
RS	Requirements Specifications
SaaS	Software as a Service
SVD	Singular Value Decomposition
STRIDE	Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege
SW	Software
ToC	Target of Certification
TOM	Technical and Organizational Measure
TSNE	T-distributed Stochastic Neighbourhood Embedding
TSVD	Truncated Singular Value Decomposition
UI	User Interface
UML	Unified Modelling Language
UUI	Unified User Interface

(cc) BY-SA

WF	Workflow
XACML	eXtensible Access Control Markup Language
XML	Extensible Markup Language

Executive Summary

The MEDINA project's final aim is to develop a framework for achieving a continuous audit-based certification for Cloud Service Providers based on cybersecurity certification schemes. Within this framework, the purposes of Work Package 2 "Certification Metrics and Languages" are:

- The elicitation of security controls and measures that are useful for automatic and continuous monitoring (Task 2.1).
- The definition of Technical and Organizational Measures (TOMs) and Security Metrics relevant for the continuous certification of Cloud Service Providers (CSP) (Task 2.2).
- The definition and the implementation of a Specification for Cloud Security Certification in a Controlled Natural Language (CNL) (Task 2.3).
- The definition and the implementation of a tool that allows users to modify part of the policies elicited for security controls (Task 2.4).
- The definition and the implementation of a tool that can translate the generated CNL to an enforceable machine-oriented Domain Specification Language (Task 2.5).
- The development of a framework for the definition of risks of the controls identified in Task 2.1 (Task 2.6).

A preliminary comparative analysis of four certification schemes has been carried out in deliverable D2.1 [1], together with a mapping of the controls of the schemes. The goal of this mapping is to provide guidance in the transitioning toward the EUCS candidate scheme (European Cybersecurity Certification Scheme for Cloud Services) [2]. For this reason, from now on, Work Package 2 "Certification Metrics and Languages" will focus on the security controls and requirements included in the EUCS.

This deliverable (D2.4) is a public report that describes the progress made and the results obtained in the second year of the MEDINA project, regarding WP2, and in particular Tasks 2.3**,** 2.4, and 2.5, so-called "Cloud Certification Language" tasks. This is the second deliverable of the series, the first version being the deliverable D2.3 [3], namely "Specification of the Cloud Security Certification Language – v1". The next and last deliverable will be D2.5 "Specification of the Cloud Security Certification Language Final", due at month M30 of the project.

The main aim of this deliverable is to provide a tool to render the security requirements of the chosen certification schema, which are expressed in Natural Language (NL), into a language that can be automatically "executed" by a machine. To achieve this objective, two sequential translations are made: the first from NL to Controlled Natural Language (CNL), the second from CNL to Domain Specific Language (DSL).

The first translation from NL to CNL allows a security requirement to be represented with security policies expressed in a formal intermediate language, still understandable for a human user. The translation of a requirement is the aim of Task 2.3, and it is realized through a component called NL2CNL Translator, described in Section 3.

The output of the previous tool can be checked and modified by an experienced user, and this is performed in Task 2.4, where the output of the previous task can be inspected through a dedicated tool, named CNL Editor, described in Section 4.

Once the user is satisfied with the CNL policies, the second translation can take place. This second translation is called 'mapping' and aims at transforming policies from CNL into DSL. The mapping phase is the main output of Task 2.5, and it is implemented by the DSL Mapper, described in Section 5.

The structure of this document is quite straightforward, it includes an initial section containing the general architecture of the Cloud Security Certification Language system, followed by three sections to detail each tool belonging to this system.

For each tool, this document describes its purpose and scope, the current coverage of MEDINA requirements, the component's internal architecture and its subcomponents, the relation to other components, the implementation state at time of writing this deliverable, and technical details of the component, including the programming languages, and used libraries, information about the packaging and installation of the component, and licensing. Moreover, the advancements within the MEDINA framework are highlighted and the planned steps for the future are described.

Currently, the tools presented hereafter have their initial prototypes implemented and integrated, to some degree, with other components of the MEDINA framework. Some requirements of the components are already fully satisfied by the prototypes in their current state, while the remaining ones are in the status 'partially satisfied'.

1 Introduction

1.1 About this deliverable

This document describes the result of Task 2.3, 2.4, and 2.5, during the second year of the MEDINA project. It reports on the internal design and, as well as on the current implementation state of the tools involved in the "Cloud Security Certification Language". The main aim is to translate the EUCS draft candidate cloud certification scheme [2] from Natural Language to a machine-readable language. The EUCS requirements, together with the associated metrics, can be seen as policies, i.e., rules that a Cloud Service Provider may/must fulfil in order to obtain the certification. The expression of these rules in the MEDINA DSL will allow the MEDINA assessment tools (presented in deliverable D3.5 [4]) to automatically process them.

This is the second deliverable on this topic, being the previous one the deliverable D2.3 [3]. The next deliverable will be D2.5 "Specification of the Cloud Security Certification Language Final" (M30). The original structure of deliverable D2.3 has been thoroughly revised and modified. This was necessary because the state of implementation at M12 was limited, so deliverable D2.3 contained mainly theoretical guidance on how tasks would be implemented. Nevertheless, the content of the current deliverable is closely related to that described in the previous one, so, for the reader's convenience, we have reported in the appendices those part of deliverable D2.3 necessary for understanding some of the concepts referred here, to provide a self-contained deliverable that facilitates the reader's understanding.

1.2 Document structure

This document is structured as follows:

- Section 1 is the current section.
- Section 2 describes the high-level architecture of the Cloud Security Certification Language system.
- Section 3 describes the NL2CNL Translator tool. It is the main output of Task 2.3 and implements the translation of policies from Natural Language (NL) to CNL, after associating a set of policies to a TOM. The output generated by this component is a set of policies (better-said obligations) associated with a TOM, expressed in CNL.
- Section 4 details the output of Task 2.4, i.e., a tool called CNL Editor. This tool allows a user to inspect and partly modify the obligations generated within the previous step.
- Section 5 describes the DSL Mapper, which is the outcome of Task 2.5. This tool maps the obligations expressed in CNL into the MEDINA DSL, whose statements are machine-readable.
- Section 6 draws some Conclusions.
- APPENDIX A: The Cloud Certification Language, describes the motivation and the methodology followed to design and develop the Cloud Security Certification Language system.
- APPENDIX B: Patterns and Controlled Natural Languages for Requirements specifications, describes the state-of the-art on Controlled Natural Languages and the MEDINA Controlled Natural Language.
- APPENDIX C: From NL to CNL TOMs, describes the strategies explored to perform the association among metrics and TOMs.
- APPENDIX D: MEDINA Vocabularies and Ontologies, describes the MEDINA vocabularies and ontologies.

1.3 Updates from D2.3

This section summarises the main updates over the different sections of the document considering that it is an evolution of D2.3 [3].

Table 1. Overview of deliverable updates with respect to D2.3

Section	Change
2	New section that describes the high-level architecture of the Cloud Security Certification Language system. The old Section 2 has been moved to Appendix A.
3	New section that describes the prototype responsible of associating metrics to TOMs and translating them into obligations (NL2CNL Translator). Old Section 3 has been moved to Appendix B.
4	Updated version of old Section 5, describing the prototype of the CNL Editor. Old Section 4 has been moved to Appendix C.
5	Updated version of old Section 6, describing the prototype of the DSL Mapper.
Appendix A	Section including the old Section 2 of D2.3, in which the motivation and methodology of the Cloud Certification Language are described.
Appendix B	Section including the old Section 3 of D2.3, which describes the Controlled Natural Language defined for the specification of requirements.
Appendix C	Section including the old Section 4 of D2.3, which reports a detailed study about possible strategies to associate metrics to TOMs.
Appendix D	Section including the old Section 5.3 of D2.3, which reports about the MEDINA vocabularies and ontologies.

CC BY-SA

2 Cloud Security Certification Language Tool High-level Architecture

This section gives an overview of the high-level architecture of the Cloud Security Certification Language components. These components are responsible for:

- Associating metrics to security requirements.
- Translating the metric/requirement pairs from Natural Language to Controlled Natural Language.
- Viewing/revising the resulting policies/obligations.
- Finally, mapping these policies/obligations into the MEDINA Domain Specific Language.

The following sections present the technical details and the current state of the implementation of the Cloud Security Certification Language components. Each section also lists the MEDINA requirements (elicited in WP5) and their current coverage by the implemented tools. For more details regarding the motivations behind the design of these tools and the methodology followed in developing them, please refer to *APPENDIX A: The Cloud Certification Language.*

2.1 Architecture

The architecture of the MEDINA framework has been first proposed in the deliverable D5.1 [5] and updated in the deliverable D5.2 [6]. It is composed by seven building blocks, as shown in Figure 1, each of them corresponding to a well differentiated functionality. The Cloud Security Certification Language corresponds to block n. 2.

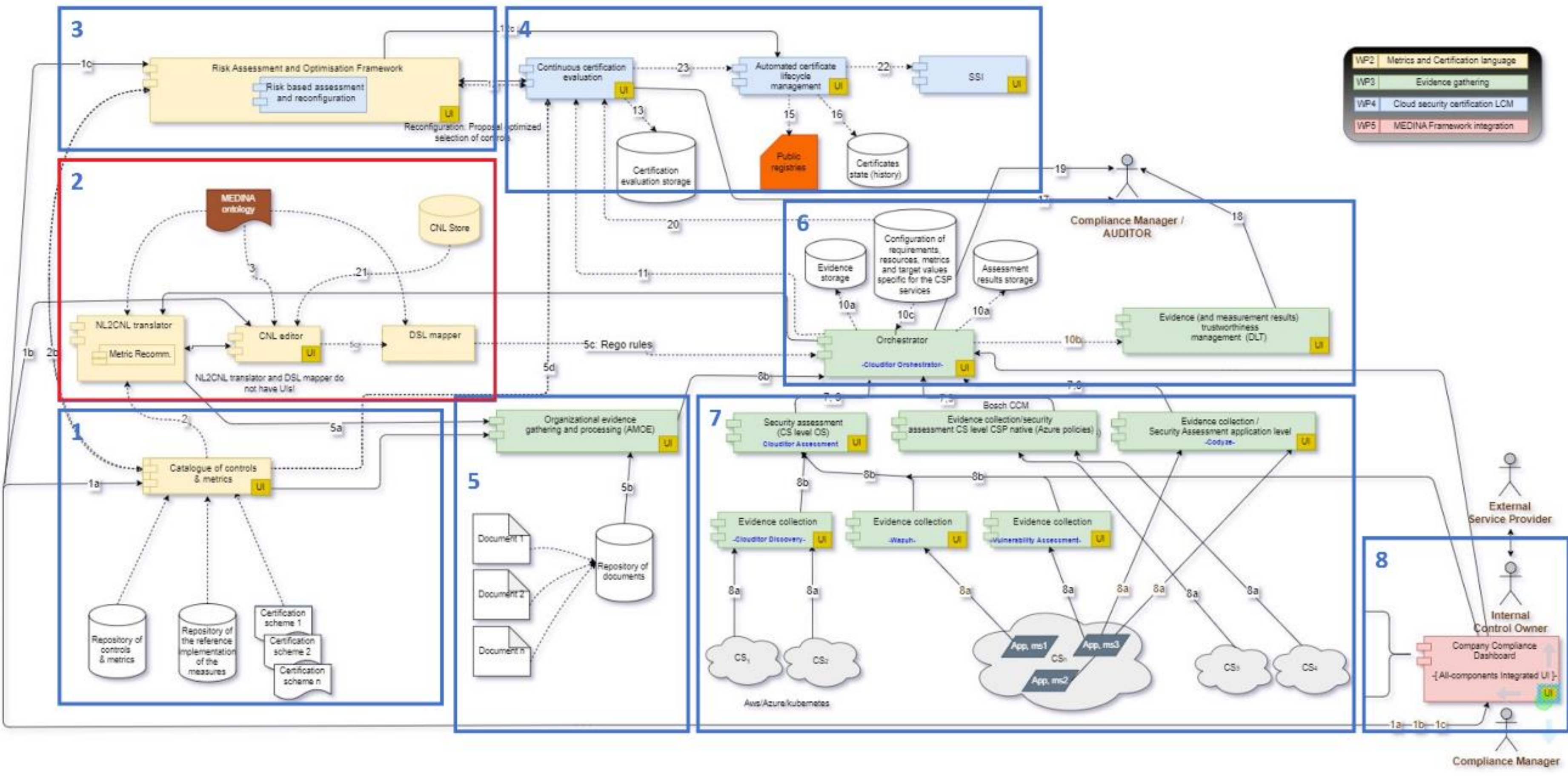


Figure 1. Building blocks view of the MEDINA framework (source: D5.2 [6])

CC BY-SA

For the sake of clarity, Figure 2 shows a more detailed schema of the components and architecture of block n.2, that will be briefly introduced afterwards.

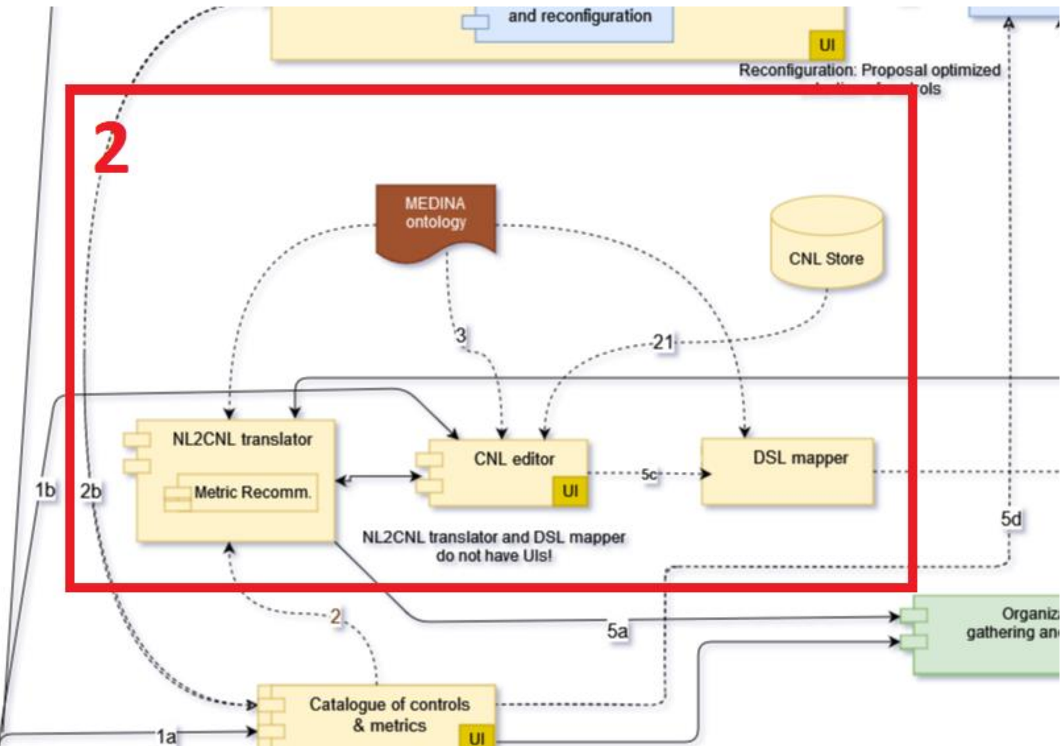


Figure 2. Architecture of the components involved in the Cloud Security Certification Language

The following are the components of the MEDINA architecture involved in block n.2 that achieve the Cloud Security Certification Language:

- The **Metric Recommender** associates a set of metrics to a requirement, by exploiting Natural Language Processing techniques. It is a sub-component of the NL2CNL Translator and is part of Task 2.4.
- The **NL2CNL Translator** translates EUCS NL requirements and metrics into their MEDINA CNL representation. The NL2CNL Translator is the main output of MEDINA Task 2.3.
- The **CNL Editor** is the user interface that allows users to visualize and possibly revise the translation of the requirements and metrics into the MEDINA CNL. The CNL Editor is the main output of Task 2.4.
- The **CNL Store** is a database that serves as output for the NL2CNL Translator and as input for the CNL Editor. It is controlled by the CNL Editor and can be accessed through its API to read and write objects in it. The objects stored in this DB are written by the NL2CNL Translator and are read by the CNL Editor and the DSL Mapper.
- The **DSL Mapper** is the MEDINA component that maps the yet not executable MEDINA CNL into the MEDINA Domain Specific Language DSL, whose statements are instead machine-readable. Therefore, the DSL is the Cloud Certification Language. The DSL Mapper is the main output of Task 2.5.
- The **MEDINA Ontology** was introduced in D2.3 [3] and recalled in this document in *APPENDIX D: MEDINA Vocabularies and Ontologies.* It is used by all the Cloud Security Certification Language components for the representation of the requirements and metrics.

The components presented so far interact with the following components in the other blocks of the MEDINA framework to perform their operations:

CC BY-SA

- The **Catalogue of controls and metrics** (a.k.a. Catalogue) belongs to block n.1. It has already been described in D2.1 [1] and will be updated in D2.2 in month M27. The interaction with this component is necessary since the NL2CNL Translator needs to retrieve the requirements and metrics descriptions from it. The two components connect to each other via their respective APIs.
- The **Orchestrator** has been described in D3.2 [7] and D3.5 [4]. The entire translation process begins and ends in the Orchestrator. In fact, the Orchestrator UI allows the user to select the requirements to be translated and to send them to the NL2CNL Translator. Finally, the result of the translation is sent to the Orchestrator itself through the DSL Mapper. Both the Orchestrator and the DSL Mapper expose an API to perform all necessary operations.

2.2 Sequence diagram

The interaction among the various components of the MEDINA framework follows predefined flows, which are described in detail in D6.3 [8] and summarized in Section 2.1.1 of D5.2 [6]. Among these workflows, the one that involves the components of the Cloud Security Certification Language is the general workflow WF3, reported in Table 2. In particular, the steps involved are step n.3 and step n.4.

Table 2. Description of the general workflow WF3, which involves the Cloud Security Certification Language components in some steps

Workflow	Workflow step no.	Workflow step comment	Linked Bosch User Story
WF3	1	Authentication to MEDINA	UC01.06 Graphical User Interface UC01.24 - User Interface Single Pane of Glass
	2	The Orchestrator UI is used to perform the following actions: a. Configure general ToC information	UC01.06 Graphical User Interface UC01.22 - Integration with Asset Management UC01.24 - User Interface Single Pane of Glass
	3	The Orchestrator UI is used to: a. Select Certification Scheme (default EUCS), EUCS Assurance level, cloud service model for the ToC to certify (any of IaaS/ PaaS / SaaS)	UC01.06 Graphical User Interface UC01.24 - User Interface Single Pane of Glass UC01.26 - Selectable Certification Schemes and Security Frameworks
	4	The UI from the CNL Editor is used to: a. Select suitable built-in Obligations as provided by the Metrics Recommender and NL2CNL Translator (or accept the ones pre-selected by default) b. Customize Target Values on the selected built-in Obligations.	UC01.06 Graphical User Interface UC01.24 - User Interface Single Pane of Glass UC01.19 – Configuration UC01.27 - Selectable Controls in Compliance Dashboard
	5	The Organizational Evidence Gathering and Processing is	UC01.06 Graphical User Interface UC01.24 - User Interface Single Pane of Glass

CC BY-SA

Workflow	Workflow step no.	Workflow step comment	Linked Bosch User Story
		used to upload the collected documentation (see WF1)	UC01.30 - Interfaces for the Provision of Organizational Evidence

These two steps n.3 and n.4 are made explicit in the sequence diagram depicted in Figure 3.

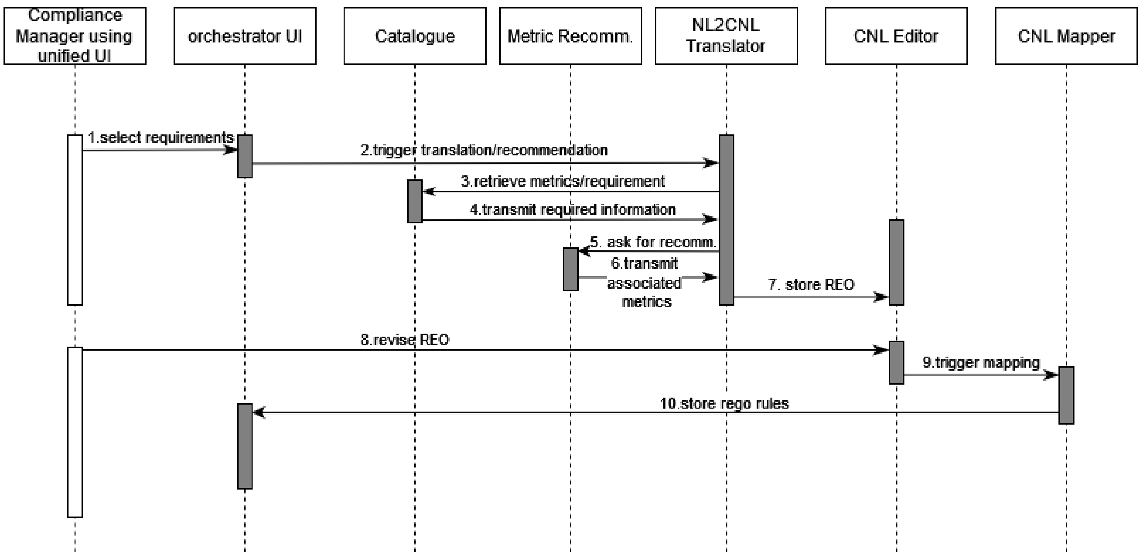


Figure 3. Sequence diagram describing the interaction among the Cloud Security Certification Language components

The sequence diagram includes the following steps:

1. The Cloud Security Certification Language functionality is activated from the Unified User Interface (UUI) of the MEDINA framework, which allows the user to access the Orchestrator User Interface (UI) and to select a (set of) requirements/TOMs to be assessed.
2. The chosen TOM is sent from the Orchestrator to the NL2CNL Translator, thus triggering the translation/recommendation.
3. The NL2CNL Translator connects to the Catalogue to retrieve information related to TOMs and metrics, in particular, it retrieves, for a certain TOM, all the metrics already associated to it in the Catalogue.
4. The Catalogue responds with the information required, if available.
5. The NL2CNL Translator queries the Metric Recommender with the TOM(s) specified by the user through the Orchestrator UI.
6. The Metric Recommender returns a set of metrics associated to a TOM. In the Catalogue, each metric is already linked to a TOM or to a set of TOMs and this association has been made by an expert when constructing the Catalogue. The metrics returned by the Metric Recommender are different from the ones already linked to it in the Catalogue and they are useful for having a larger list of metrics to choose from.
7. The NL2CNL Translator translates all the metrics/requirements pairs into obligations, then it builds an object containing all the information needed to describe a TOM and its associated metrics/obligations. This object is called REO (Requirement&Obligation) and

it is represented in XML. The REO object is sent from the NL2CNL Translator to the CNL Editor, which takes care of storing it into the CNL Store.

8. The control returns to the UUI, giving the user the possibility to open the CNL Editor and view/revise the obligations.
9. Once satisfied with the obligations, the user can directly trigger the mapping from the CNL Editor UI, by sending the REO object to be mapped to the DSL Mapper.
10. The DSL Mapper extracts from the REO object the information needed to generate the Rego rules, which are finally sent back to the Orchestrator.

2.3 Coverage of requirements

This section provides an overview of the status of the functional requirements elicited during the first two years of the project, as reported in deliverable D5.2 [6], focusing on those requirements related to the Cloud Security Certification Language. Table 3 is an excerpt of Table 5 of deliverable D5.2 [6] and indicates, for each functional requirement, its priority, and its current and expected implementation status.

Requirements have been classified as *Not Implemented* (-), *Partially implemented (P)* or *Fully implemented (F)*. The columns M15, M24 and M33 refer to the month where the status has been measured. **M15** correspond to the first version of the requirements, **M24** refers to the actual status; and **M33** is the foreseen status of the implementation of the requirements in month 33, previous to the final MEDINA integration.

Each requirement will be described in detail in the following sections.

Table 3. Expected coverage of functional requirements for the Cloud Security Certification Language.

KR	Sub-component	Req. ID	Short title	Priority	M15	M24	M33
KR3 Certification Language	NL2CNL Translator	NL2CNL.01	Translation from NL to Controlled NL	MUST	P	P	F
		NL2CNL.02	Based on NLP and ontologies	MUST	-	P	F
		NL2CNL.03	Translation of org. and technical measures	Should	P	P	F
		NL2CNL.04	Compliant with the CNL Editor language	MUST	P	P	F
	CNL Editor	CNLE.01	CNL Editor GUI	MUST	-	F	F
		CNLE.03	CNL Editor input format	Should	-	F	F
		CNLE.04	CNL Editor policies changing	MUST	P	F	F
		CNLE.05	CNL Editor vocabulary	MUST	-	P	F
		CNLE.06	CNL Editor output format	MUST	P	F	F
	DSL Mapper	DSLM.01	Translation to selected DSL	MUST	-	P	F
		DSLM.03	Mapping elements	MUST	-	P	F

(cc) BY-SA

3 NL2CNL Translator

The NL2CNL Translator is a component of the MEDINA framework that has two main purposes. The first goal is to select a set of metrics that could be useful to evaluate a certain security requirement, also called TOM (Technical and Organizational Measure). After associating a set of metrics to a requirement, the second goal is to translate those metrics into policies. Specifically, metrics are expressed in NL, while the translated policies are expressed in CNL. More information regarding CNLs in general and the specific CNL chosen for the MEDINA project can be found in the *APPENDIX B: Patterns and Controlled Natural Languages for Requirements* specifications, while *APPENDIX C: From NL to CNL TOMs* describes the strategies explored to perform the association among metrics and TOMs.

3.1 Implementation

3.1.1 Functional description

The NL2CNL Translator relies on the Catalogue of controls and metrics as a data source, in fact both the security requirements (TOMs) and metrics are stored in it. A first version of the Catalogue has already been described in D2.1 [1] and will be updated in D2.2 at month M27. Currently, the focus of the Catalogue is on the EUCS candidate scheme (European Cybersecurity Certification Scheme for Cloud Services [2].

As previously stated, in order to be evaluated, the TOMs available in the EUCS scheme must be associated with metrics, and these metrics, in turn, can be expressed by means of obligation policies[1]. The final aim of the NL2CNL Translator is to translate a set of metrics associated to a TOM into obligations, expressed in CNL.

Related requirements

In the following there is a collection of functional requirements (from deliverable D5.2 [6]) related to the component, together with a description of how and to what extent these requirements are implemented at time of writing.

Requirement id	NL2CNL.01
Short title	Translation from natural language to controlled natural language
Description	The tool shall be able to translate in a semi-automatic way the requirements selected from a security certification scheme – originally expressed in natural language (English), into a set of obligations expressed in a controlled natural language. The output of the tool will be checked manually to verify if the obligations generated by the tool are correctly linked to the selected requirement.
Implementation state	Partially implemented

The NL2CNL Translator associates a set of metrics to each selected TOM, by using the Metric Recommender, and then translates all the associated metrics into obligations. Due to the limits in the precision of the Metric Recommender, it is possible that some of the associated obligations are not suitable for the selected TOM, thus it is necessary to manually verify the output. In the current implementation, the NL2CNL Translator is able to correctly generate CNL obligations for all the requirements included in the Catalogue of controls and metrics.

[1] As reported in Appendix B, the Medina CNL language can express both authorizations and obligations. For the requirements investigated within EUCS, we have so far only considered obligations.

Requirement id	NL2CNL.02
Short title	Based on NLP and ontologies
Description	Given natural language sentences taken from the cloud certification schema, the tool will rely on NLP techniques to link these sentences to a list of recommended metrics.
Implementation state	Partially implemented

The NL2CNL Translator relies on the Metric Recommender to choose which are the metrics to be associated to a TOM. This association is done by considering the text similarity among the TOM description and the metrics descriptions, both expressed in NL. Due to the limited number of data considered up to this phase of the project, the results of the association can be inaccurate. In the next phase of the project, additional effort will be made to study how to improve the accuracy of the Metric Recommender.

Requirement id	NL2CNL.03
Short title	Translation of organizational measures and technical measures
Description	The NL2CNL Translator will be able to translate some of the organizational measures specified in the chosen EU cloud certification schemas, and some of the technical measures.
Implementation state	Partially implemented

The NL2CNL Translator is currently able to process the EUCS TOMs identified in D2.1 [1], which are the ones available in the Catalogue of controls and metrics at time of writing. In the next phase of the project, the aim is to extend the number of TOMs considered and to update this functional requirement consequently.

Requirement id	NL2CNL.04
Short title	Compliant with the CNL editor format
Description	The controlled natural language output of the NL2CNL Translator will be compliant with the format used by the CNL Editor to represent the obligations.
Implementation state	Partially implemented

The NL2CNL Translator produces an object containing the TOM's metadata, metrics metadata and CNL obligations. All this information is stored in an XML file, compliant with the format used by the CNL Editor. In the future, it could be necessary to update the format.

Requirement id	NL2CNL.05
Short title	XML Compliant
Description	The controlled natural language output of NL2CNL translator will be compliant with the XML based format supported by the CNL Editor.
Implementation state	Discarded

This requirement has been cancelled since it duplicates requirement NL2CNL.04.

Main innovations

The main innovation introduced by the NL2CNL Translator is the use of a Metric Recommender based on NLP techniques to automatically associate a set of metrics to a TOM.

3.1.1.1 Fitting into overall MEDINA Architecture

Figure 4 shows the integration of the NL2CNL Translator within the overall MEDINA architecture. The components available are described in detail in Section 3.1.2.1 and are mapped to the MEDINA framework as follows:

- The **API Server** component coordinates all the other modules and implements the API interface towards the outside.
- The **Metric Recommender** component implements the functionalities offered by the Metric Recommender in the MEDINA framework, i.e., it associates a TOM with a set of metrics.
- The **Translation** component implements the functionalities of translating the set of metrics in obligations, from NL to CNL.

The NL2CNL Translator interacts with the Catalogue of controls and metrics, which is its main source of data since it contains the TOMs and metrics descriptions and metadata. The results of the translation are stored in the CNL Store, a database managed by the CNL Editor, accessible though the CNL Editor APIs. Moreover, the NL2CNL interacts also with the Orchestrator, which triggers the translation through its User Interface.

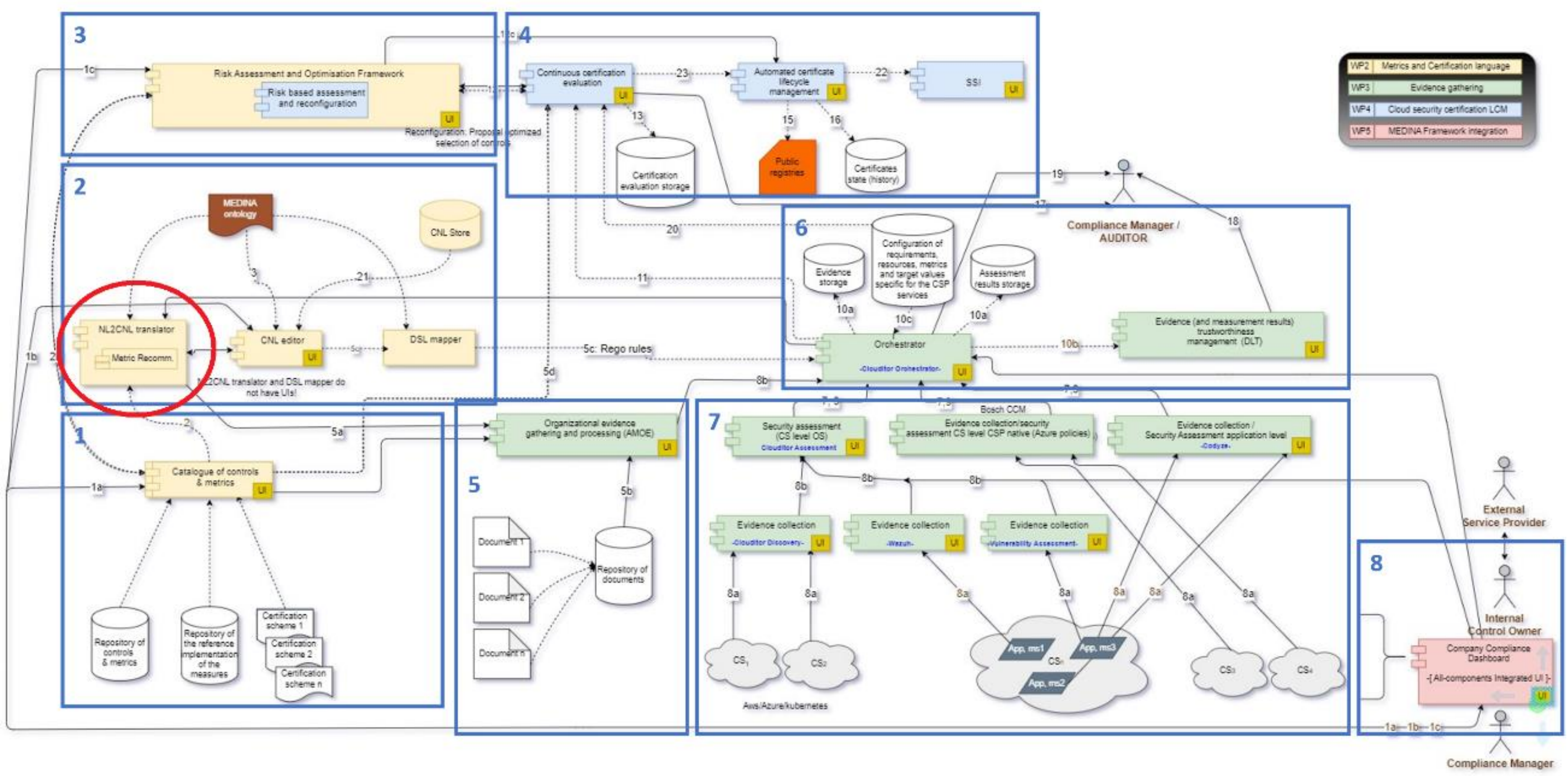


Figure 4. Position of the NL2CNL Translator within the MEDINA architecture (source: D5.2 [6])

3.1.2 Technical description

In the following, we provide the technical description of the NL2CNL Translator components. First, we present the architectural design, consisting of the architectural view and the connection between the respective components. This is followed by information on the individual components and finally by a summary of the technical description for the implementation of the prototype.

3.1.2.1 Prototype architecture

The NL2CNL translator is written in Python 3 and is organized in modules. Its functionalities are available through a REST API; thus, a Python API framework has been chosen for implementing it, i.e., FastAPI[2]. This is a very fast and high-performance Python framework, which supports a compact coding structure, resulting in fast development.

FastAPI is based on the *asyncio* capabilities of Python[3], which has been standardized as ASGI (Asynchronous Server Gateway Interface) specification[4] for building asynchronous web applications. In terms of features, FastAPI is similar to Flask[5], another widespread Python framework for writing APIs.

Figure 5 shows the architecture of the NL2CNL Translator.

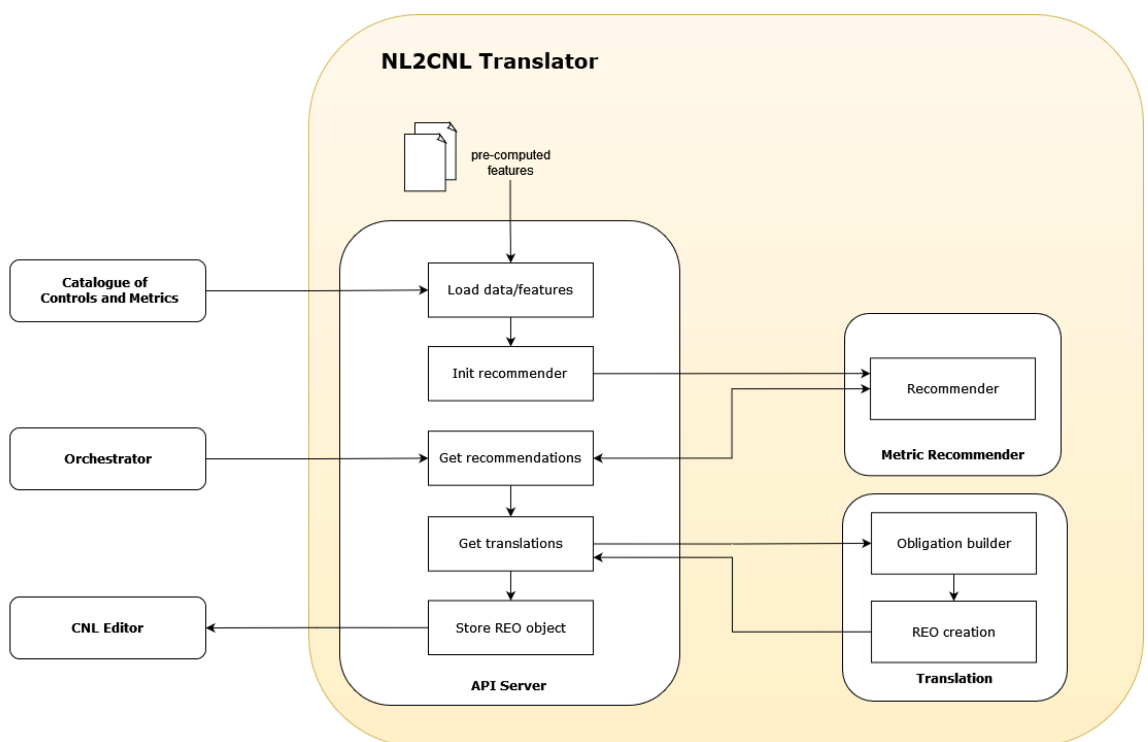


Figure 5. Overview of the NL2CNL Translator architecture

The NL2CNL Translator modules are organized into three components, according to the functionalities they offer:

[2] https://fastapi.tiangolo.com/
[3] https://docs.python.org/3/library/asyncio.html
[4] https://asgi.readthedocs.io/en/latest/specs/main.html
[5] https://flask.palletsprojects.com/en/2.2.x/

- **API Server**: it implements the API interface towards other MEDINA components. Moreover, it coordinates all the operations and the connections towards other internal components.
- **Metric Recommender**: given a TOM, it performs the association between TOM and metrics, thus it provides a set of metrics.
- **Translation**: this component takes a TOM and the associated set of metrics and translate everything into a REO object (Requirements&Obligations), which includes the TOM's metadata and the metrics translated into obligations. This object is then returned to the server.

The endpoints available to interact with the other MEDINA components are reported in Table 4.

Table 4. List of available endpoints for the NL2CNL Translator component

Function name	Parameters	Return Type	Description
create_reo_from_requirement	username, tom_code	HTTP Response	This function is designed to be called by the Orchestrator, which must transmit two parameters: the username of the user currently connected to MEDINA, and the identifier of the TOM with which the metrics and related obligations are to be associated. This is a POST function, thus if the execution is successful the final effect will be the creation of a REO object within the CNL Store of the CNL Editor. If successful, the status code returned will be of type 201, otherwise an error code will be returned depending on the problem occurred (e.g., status code 401 is returned if the specified user does not have sufficient permissions).
livez	None	HTTP Response	This function is used by the Kubernetes primary node agent to know when to restart a container. In fact, many applications running for long periods of time eventually transition to broken states and cannot recover except by being restarted.
readyz	None	HTTP Response	This function is used by the Kubernetes primary node agent to know when a container is ready to start accepting traffic.

3.1.2.2 Description of components

This section presents the components available in the NL2CNL Translator and describes how they have been developed and will continue to be updated to meet the MEDINA requirements.

API Server

The API Server is the core of the NL2CNL Translator. It connects and coordinates with all the other services/components and implements the API interfaces provided to the outside. Its logic can be divided into 5 steps:

1. Upon launching the app, the API server loads the textual descriptions of TOMs and metrics from the Catalogue of Control and Metrics. In addition, the pre-computed numerical features for these texts are loaded from local files. Using pre-computed features has pros and cons. The main advantage lies in the fact that features computation is time consuming, and thus by computing features offline the app starting is faster. The disadvantage is that when a new TOM or metric is added to the Catalogue of Control and Metrics, the corresponding feature vector for that element must be computed and the app restarted to load the changes. When the MEDINA framework is ready for production, we expect that there will be no changes to the set of EUCS requirements considered, thus we opted to use the approach of pre-computing features.
2. The Metric Recommender is then initialized with the K-d tree algorithm, computed on the feature vectors of the metrics.
3. After the preliminary steps, the API server waits for requests from the Orchestrator component. The Orchestrator starts a translation by calling the API server and passing the identifier of a TOM. The API server then calls the Metric Recommender to get the association between a TOM and a set of metrics.
4. These metrics are then translated into obligations, i.e., for each metric associated to a TOM, an obligation is constructed in XML language, using the metrics metadata previously loaded from the Catalogue of Control and Metrics. The obligations expressed in XML are then embedded into a larger object called REO, which includes TOM's metadata, metrics metadata and the obligations themselves.
5. Finally, the object thus created is saved into the CNL Store, using the APIs provided by the CNL Editor.

Metric Recommender

The Metric Recommender is initialized with a K-dimensional tree (K-d tree) algorithm. In the next stages of the MEDINA project, we plan to experiment with other algorithms to improve its performance. The Metric Recommender takes the identifier of a TOM so that it can access the correspondent textual description. Next, the correspondent feature vector is retrieved and used as input for the K-d tree algorithm to select the k closest neighbours of the query vector based on the shortest Euclidean distance. Finally, the Metric Recommender returns a set of metrics whose features are the most similar to those of the specified TOM.

Translation

The Translation component takes care of translating the metrics into obligations, moreover it generates an object compliant with the input required by the CNL Editor. This object is called REO (Requirements&Obligations), since it contains all the information needed to express a TOM and the associated obligations. The REO structure is depicted in Figure 8.

Once the object is created, the control returns to the API Server component.

3.1.2.3 Technical specifications

The prototype of the NL2CNL Translator is written in Python, version 3.8. A selection of the key libraries is shown in the following and a full list including all the used libraries can be found in the GitLab repository[6].

- fastapi
- uvicorn

[6] https://git.code.tecnalia.com/medina/public/nl2cnl-translator

- scipy
- scikit_learn
- fasttext
- numpy
- pandas
- spacy
- nltk
- python-keycloak

3.2 Delivery and usage

The following sections give a short overview of the delivery and usage of the prototype.

3.2.1 Package information

The package is delivered in a repository, containing all the needed deployment and configuration scripts for installing the NL2CNL Translator. Even if the Translator is not intended to be used as a standalone tool, it is possible to use it for testing purposes, mainly accessing the APIs through a web browser. The component is available as a Docker image. Table 5 shows the structure of the most important folders and a brief description of them.

Table 5. Most important files and folders implementing the NL2CNL Translator

Folder/file	Description
api_server.py	This file contains the code needed for implementing the APIs and for managing the communication between components.
Dockerfile	This file contains the list of commands that the Docker client calls while creating an image.
config.py	This file includes all the configurations (variables, paths, etc.) used by the prototype.
clean_sentence.py	This file belongs to the Metric Recommender. It is used to pre-process the textual description of TOMs and metrics.
fasttext_features.py	This file belongs to the Metric Recommender. It is used to pre-compute the numerical features for the textual descriptions using the fasttext library.
utils.py	This file belongs to the Metric Recommender and implements its logic.
app_utils/	This folder includes a set of utility functions.
cnl_editor_client/	This folder contains the client to interact with the CNL Editor component.
coc_backend_api_client/	This folder contains the client to interact with the Catalogue of Control and Metrics component.
data/	This folder contains local data used by the NL2CNL Translator, such as the pre-computed features for TOMs and metrics and the XML templates to build the REO objects.
openapis/	This folder contains the auto-generated OpenAPI files.

(cc) BY-SA

3.2.2 Installation instructions

The full up-to-date installation instructions can be found in the README at the NL2CNL Translator GitLab repository at this link: https://git.code.tecnalia.com/medina/public/nl2cnl-translator/-/blob/main/README.md

To setup the prototype in a local machine, it is possible to build the Docker image as follows:

```
$ docker build -t nl2cnl_translator_image .
```

3.2.3 User Manual

After building the Docker image of the prototype, it can be run as follows:

```
$ docker run -p 8000:8000 -it --name nl2cnl_translator
nl2cnl_translator_image
```

The container can be started and stopped as follows:

```
$ docker start nl2cnl_translator
$ docker stop nl2cnl_translator
```

It is possible to test the API through the command line with the CURL command, or through a browser with the interactive APIs.

For example, by typing in the command line:

```
$ curl -X 'POST' 'http://127.0.0.1:8000/
create_reo_for_requirement/{username}?tom_code={requirement_name}
```

with `nl2cnl_test` as username and `OPS-05.3` as requirement_name, a new REO object will be created for the user in the CNL Store and the id of the created object will be returned.

Alternatively, it is possible to open a browser and visit the following page: http://127.0.0.1:8000/docs#

The interactive APIs will be opened, and it will be possible to test them directly in the browser.

Alternatively, it is possible to test the interactive APIs by using the component installed in the MEDINA framework, which can be reached following this link:

https://nl2cnl-translator-test.k8s.medina.esilab.org/docs#/

3.2.4 Licensing information

The NL2CNL Translator component is open source, under the Apache License 2.0.

3.2.5 Download

The code is currently available on MEDINA's git repository, in the GitLab hosted by TECNALIA:

https://git.code.tecnalia.com/medina/public/nl2cnl-translator

3.3 Advancements within MEDINA

This section reports the advancements implemented in the NL2CNL Translator component within the second year of the MEDINA project. In particular, the following list includes the improvements introduced between month 12 and month 24:

- The API interface has been redesigned: endpoints not intended to be accessed from the outside have been hidden, while they remain in the code for testing purposes.
- In addition to the Catalogue client, a second client has been implemented to interact with the CNL Editor. Moreover, the Catalogue client has been updated to reflect changes in the Catalogue of Control and Metrics.
- The Translation component has been implemented and finalized. This implied the definition of the REO structure, the definition of the interface towards the CNL Editor and the implementation of the REO generation.
- Several utility functions have been added to support the functionalities implemented by the Metric Recommender and the Translation components.
- Regarding the Metric Recommender, the data source was initially taken from some excel files containing the TOMs and metrics definitions. This functionality has been updated and the NL2CNL Translator is currently able to interact with the Catalogue of Control and Metrics to take the information it needs.
- The secure authenticated access to the DSL Mapper endpoints is guaranteed through the use of Keycloak[7], an open-source tool that enables single sign-on with Identity and Access Management to all MEDINA components.

3.4 Limitations and future work

The main limitation in the current development of the NL2CNL Translator prototype regards the results obtained by the associations provided by the Metric Recommender. We are aware that the obtained results may sometimes be inaccurate. This is due to the lack of data to work with. In fact, at this stage of the project, the focus is on a small number of EUCS requirements. This was also due to the fact that this part of the project was dedicated to designing and implementing a working framework in which all components are able to talk to each other. In the near future, we plan to focus on refining the Metric Recommender and the results it generates.

An initial improvement will be achieved by increasing the number of metrics considered, by updating the Catalogue of Control and Metrics. Secondly, we plan to carry out a fine-tuning of the *fasttext* model, by training it on texts that relate specifically to cybersecurity. A further attempt will also be made by trying different classification algorithms that can take advantage of the associations between requirements and metrics already available in the Catalogue of Control and Metrics. Finally, a further possibility is the introduction of a different dataset derived from other security domains, e.g., datasets relating to the specification of security requirements in the context of software programs[8].

Finally, as plan for the rest of the project, we will update the NL2CNL translator to deal with the August 2022 draft candidate version of the EUCS scheme [9].

[7] https://www.keycloak.org
[8] https://zenodo.org/record/4530183#.YtFLxOxBxDM

4 CNL Editor

The CNL Editor is a tool that has two main functionalities for an authorize account: showing the REO (Requirement&Obligation) objects that are created from Recommender/NL2CNL Translator components and allowing some changes to these objects before passing to the next step of DSL Mapping. The tool has a Web interface and REST API to carry out its functionalities.

This section describes CNL Editor tool in the context of the MEDINA framework with its implementation and use at the actual stage of the prototype. Eventual next changes will be reported in future deliverable.

4.1 Implementation

4.1.1 Functional description

The CNL Editor is a component of the MEDINA framework that allows a user, with a web interface, to refine the obligations (policies) associated with a specific TOM by the Recommender/NL2CNL Translator components (see 3.1.1). The associations are contained in an object called REO (Requirement&Obligation) which is an XML file whose structure is explained in 4.1.2.2. The obligations are showed as CNL statements and the CNL Editor gives the user the possibility to change the *operator* and/or the *target value* of a specific obligation or to delete obligations not valid or appropriate.

The CNL Editor uses a vocabulary that contains terms coming from the Catalogue of controls and metrics to guide the user with operator and target value selection. The CNL Editor works on REO objects stored in an internal database accessible from the CNL Editor itself with an internal API. Some basic operations on REOs can be invoked from other components using the CNL Editor API.

A user can connect to the CNL Editor, from the MEDINA UI and through the Editor Web Interface in order to:

- Visualize the list of associated REOs.
- Select a specific REO to:
  - *show* the REO
  - *update* the REO by deleting obligations or by modifying the operator and/or the target Value
  - *delete* the REO

When a REO is considered completed, the user can invoke the *map* function in the CNL Editor that calls the DSL Mapper to translate CNL statements into Rego code (see Section 5.1).

Related requirements

In the following there is a collection of functional requirements (from deliverable D5.2 [6]) related to the component, together with a description of how and to what extent these requirements are implemented at time of writing.

Requirement id	CNLE.01
Short title	CNL Editor GUI
Description	The controlled natural language Editor will have an interface accessible by web browser.
Implementation state	Fully implemented

The CNL Editor is available through a Web GUI, access to which is managed by Keycloak[7]. The web GUI shows the REOs associated to a user and allows her/him to perform the aforementioned operations (show, edit, delete, map) on them.

Requirement id	CNLE.02
Short title	CNL Editor policies authoring
Description	The CNL Editor will allow creating statements for security controls.
Implementation state	Discarded

This requirement was related to initial idea of policy fulfilment. In earlier stages of the MEDINA project, it was decided to create policies with a different component (Orchestrator and NL2CNL Translator). So, this functionality is no longer requested to the CNL Editor in the interactions with the other tools. Users can visualize obligations and can change the *targetValue* or select obligations that must remain associated with the REO for the next mapping phase, but they cannot create new obligations.

Requirement id	CNLE.03
Short title	CNL Editor input format
Description	The CNL Editor will accept as input NL2CNL Translator format (XML based).
Implementation state	Fully implemented

The CNL Editor takes in input a CNL file containing a REO object in XML format, as generated by the NL2CNL Translator and stored in an internal CNL Editor database. When the CNL Editor shows the list of REOs to the user, it reads them from this database. Updates can be made on these files through the CNL Editor internal API.

Requirement id	CNLE.04
Short title	CNL Editor policies changing
Description	The CNL Editor will allow changing input (policies) from NL2CNL Translator.
Implementation state	Fully implemented

When a user visualizes, through the CNL Editor GUI, the policies inside the REO, she/he can change some values for the operator and the target values, or she/he can delete policies. The REO is generated by the NL2CNL Translator, and the user can, in this way, refine the policies, according to the CSP specific requirements. After the edit phase, the CNL Editor writes the updated REO file on the internal store.

Changes can also be made in the CNL Editor by using the Editor API (see Section 4.1.2.2).

Requirement id	CNLE.05
Short title	CNL Editor vocabulary
Description	The CNL Editor will use an ontology-based vocabulary to model security controls. Ontology will be the same used by NL2CNL Translator and based on W3C Web Ontology Language (OWL) standard format.
Implementation state	Partially implemented

The CNL Editor is strictly related to a vocabulary, that is an internal file containing the ontology that the CNL Editor uses for CNL purposes. The Ontology comes from The Catalogue of controls and metrics, and it contains terms related to *MetricName*, *Operator*, *TargetValue*, and *ResourceTypes*, which are the terms used inside the policy statements. This vocabulary must be

aligned with the Catalogue of controls and metrics and will therefore be updated throughout the life of the project.

Requirement id	CNLE.06
Short title	CNL Editor output format
Description	The CNL Editor will generate security controls with an XML format suitable for the DSL Mapper.
Implementation state	Fully implemented

The CNL Editor writes the updated REOs in XML format and this is the input for the DSL Mapper. The mapping function, which converts CNL policies into Rego code, is invoked by the CNL Editor through the *Map* button on the GUI.

Main innovations

The main innovation of the CNL Editor is the possibility to change, with constraints, the policies associated with a TOM. The user can change the operator and the target values for an obligation, or she/he can delete obligations within a REO.

4.1.1.1 Fitting into overall MEDINA Architecture

Figure 6 highlights the CNL Editor within the MEDINA architecture and shows its relations with the other components.

The CNL Editor takes as input the files stored by the NL2CNL Translator within the CNL Store and updates these files which in turn will be used as input by the DSL Mapper. The two components mentioned, i.e., the NL2CNL Translator and the DSL Mapper, interact directly with the CNL Editor through its API.

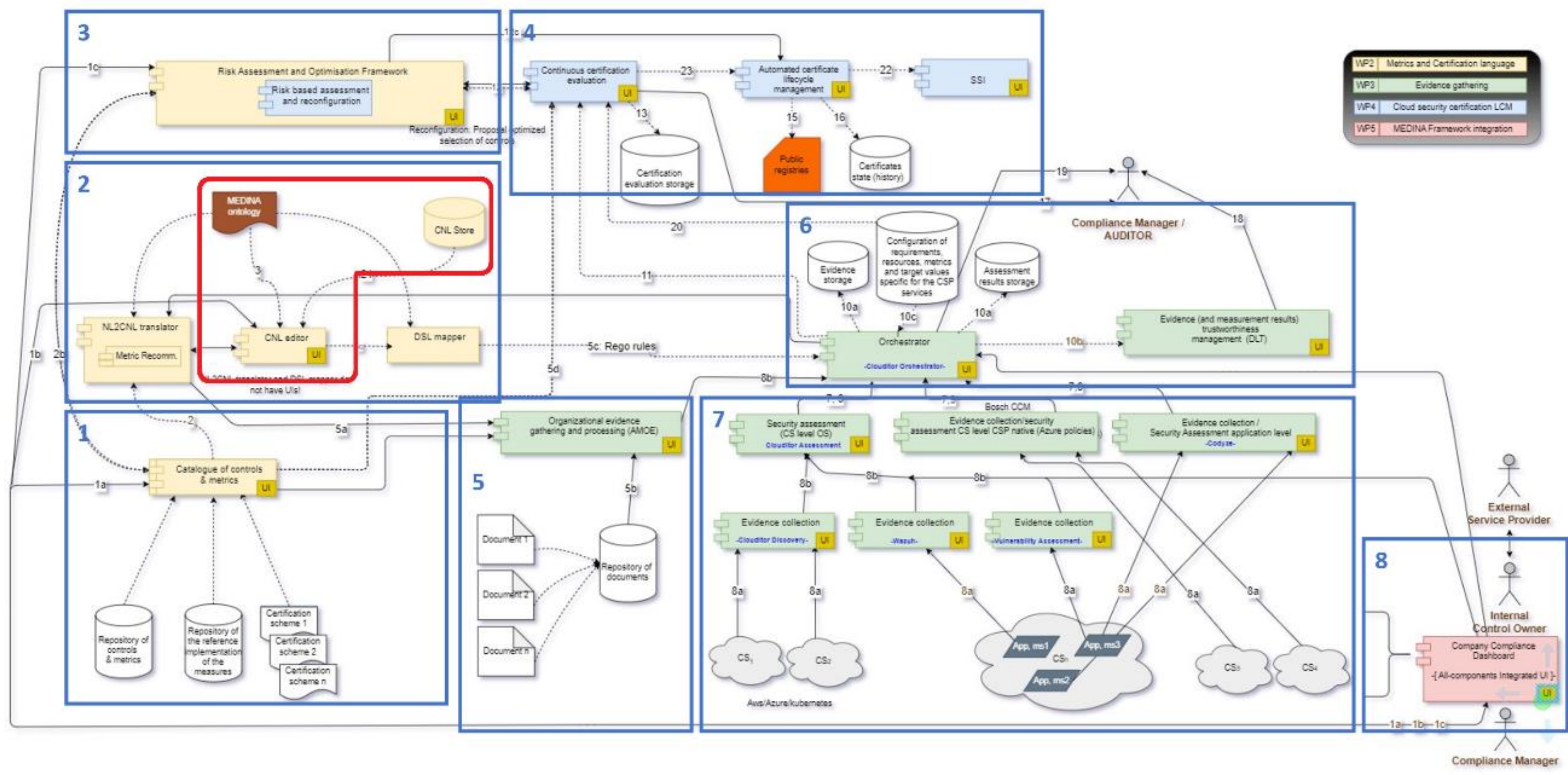


Figure 6. Position of the CNL Editor within the MEDINA architecture (source: D5.2 [6])

(cc) BY-SA

4.1.2 Technical description

In the following, we present the architecture of the CNL Editor and the description of its sub-components, together with the list of APIs available for the other tools.

4.1.2.1 Prototype architecture

The CNL Editor architecture is depicted in Figure 7.

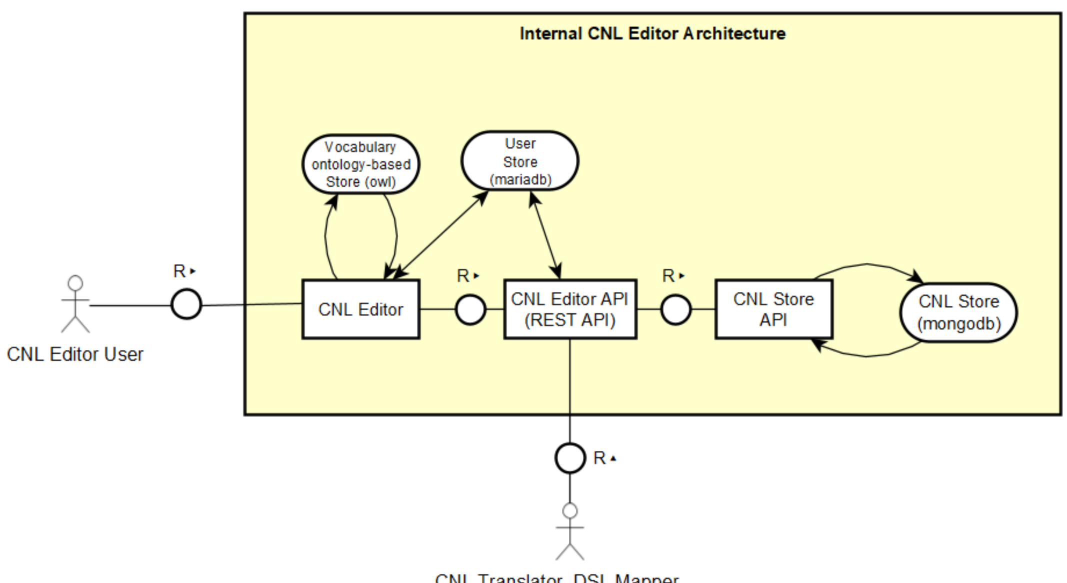


Figure 7. Overview of the CNL Editor architecture

The component is written in Java with the use of Spring Boot[9], GWT (Google Web Toolkit[10]) and Vaadin[11] frameworks. Spring Boot is used for all the API and the CNL Editor Web Application logic, while GWT and Vaadin are used for the UI. The Web GUI is used for the complete functionalities of the CNL Editor. CRUD (Create, Read, Update, Delete) operations on REO are available through REST APIs.

The component is organized in the following modules which are detailed in Section 4.1.2.2:

- CNL Editor
- CNL Editor API
- CNL Store
- CNL Store API
- Vocabulary
- User Store

The CNL Editor has interfaces with:

- The user that can use the Web GUI
- The NL2CNL Translator that builds the input for the CNL Editor (user's REOs file)
- The DSL Mapper that takes as input the REOs located in the CNL Store to map them into Rego code

[9] https://spring.io/projects/spring-boot
[10] https://www.gwtproject.org/
[11] https://vaadin.com/

In addition, the CNL Editor exposes the endpoints (REST API) listed in Section 4.1.2.2, which can be invoked without the use of the Editor GUI.

4.1.2.2 Description of components

This section presents the components available in the CNL Editor and describes how they have been developed and will continue to be updated to meet the MEDINA requirements.

CNL Editor

The CNL Editor (frontend and fore application) manages the user interaction and allows operations on REOs, and the DSL Mapper requests. It has a web interface where a user can show a REO and select operations on it.

The REO structure is composed of two parts:

- Metadata, containing the identification of the REO and the data of the specific TOM:
  - REO name, description, and identification.
  - Vocabulary version.
  - TOM Code, TOM Name, Security Control, Framework, Assurance Level.
- Policies, containing the list of all the obligations associated to the selected TOM.

These two parts also reflect the two sections that the user can see on the web page when showing a REO: Metadata on the upper part and Policies at the bottom (see Figure 8).

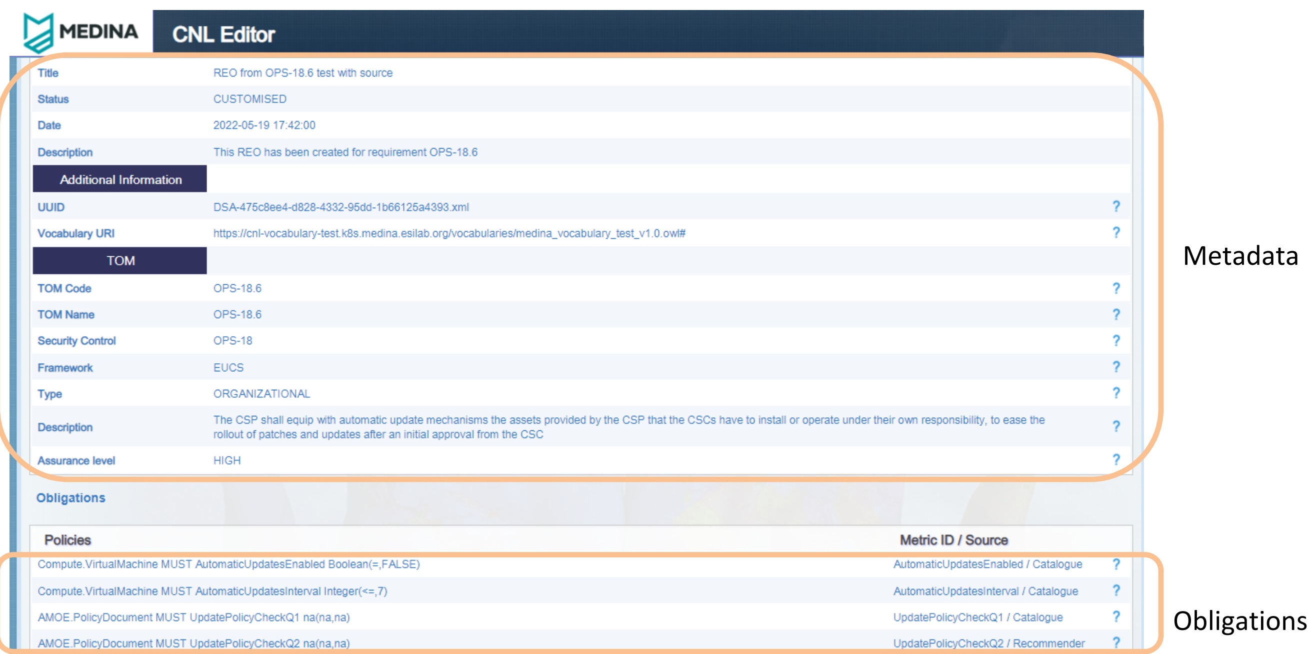


Figure 8. Screenshot of a REO object as it appears in the CNL Editor GUI

CNL Editor API

The CNL Editor exposes REST APIs for CRUD operations on REOs. They are listed in Table 6.

(cc) BY-SA

Table 6. List of available endpoints for the CNL Editor component

Function name	Parameters	Return Type	Description
/reo/create/{username}	username	HTTP Response	Store a new REO in the CNL Store and return a unique REO identifier. This function is designed to be called by the NL2CNL Translator which must transmit as a parameter the identifier of the user that owns the REO. This is a POST function, since it generates a file .xml containing the REO data. If successful, the status code returned will be of type 200, otherwise an error code will be returned depending on the problem occurred.
/reo/delete/{reoid}	reoid	HTTP Response	Delete from the CNL Store the REO specified by REO identifier. If successful, the status code returned will be of type 200, otherwise an error code will be returned depending on the problem occurred.
/reo/get/{reoid}	reoid	HTTP Response	Retrieve the REO XML file by its identifier. If successful, the status code returned will be of type 200, otherwise an error code will be returned depending on the problem occurred.
/reo/update/{reoid}	reoid	HTTP Response	Rewrite XML file of the REO specified by identifier. It is used by the CNL Editor to update a REO when the user changes the operator and/or the target value. If successful, the status code returned will be of type 200, otherwise an error code will be returned depending on the problem occurred.

CNL Store and API

The CNL Store is the database containing all the REO files. There is an internal API (usable only by the CNL Editor internal code) to access the CNL Store that directly acts on the database.

Vocabulary

The Vocabulary stores the ontology that the CNL Editor uses to constrain the user in selecting policy parameters like the operator and the target values.

User Store

The User Store is the internal database that contains username and REO associations for each username. Users are created in alignment with Keycloak.

4.1.2.3 Technical specifications

The component is written in Java with the use of Spring Boot, GWT and Vaadin frameworks.

Both the API and the Web GUI are served through Tomcat application server.

The vocabulary ontology-based store is a file in RDF/XML format, with. owl extension, that supports OWL Web Ontology Language and is W3C Standard Compliant. Details on the Editor ontology are in *Appendix D: Editor Ontology*.

The User Store is a MySQL database (mariadb software) that manages Editor users and the association between users and REOs.

The CNL Store (mongodb software) is a document oriented (no relational) database that contains all the REO files.

4.2 Delivery and usage

4.2.1 Package information

The CNL Editor code is stored in a private GitLab group named "CNL Editor Tools" (see Section 4.2.5). Inside this folder there are five different projects related the tool: the "CNL Store API", which implements the functionalities of the CNL Store and the mongodb database for the data persistence; the "CNL Editor API", which implements the CNL Editor API module; the "CNL Editor" and the "CNL Editor Frontend", which implement the CNL Editor core application and its web interface; and the "CNL Vocabulary" for the storage of the ontology (see Figure 9).

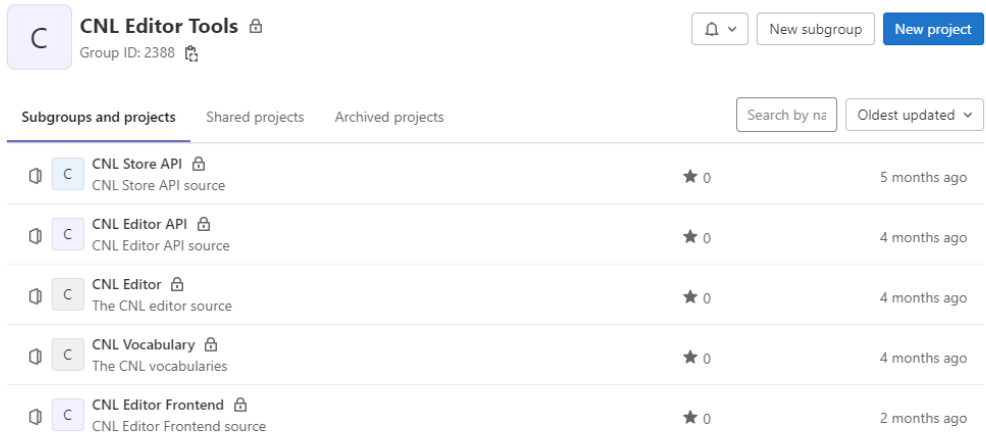


Figure 9. CNL Editor folders' structure

4.2.2 Installation instructions

The CNL Editor is developed using the microservice architecture and is made up by 5 microservices containerized in Docker. To run the components in a local machine, you can access with your credentials to the GitLab repository and pull the code, then from the root folder it is possible to build the Docker image of each microservice as follows:

1. CNL Store API:
   ```
   $ docker build -t cnl-store-api ./CNL Store API
   ```
2. CNL Editor API
   ```
   $ docker build -t cnl-editor-api ./CNL Editor API
   ```
3. CNL Editor
   ```
   $ docker build -t cnl-editor ./CNL Editor
   ```
4. CNL Vocabulary
   ```
   $ docker build -t cnl-vocabulary ./CNL Vocabulary
   ```
5. CNL Editor Frontend
   ```
   $ docker build -t cnl-editor-frontend ./CNL Editor Frontend
   ```

After that, it is possible to run the entire application using Kubernetes.

Indeed, each component has the manifests files declared in a dedicated “Kubernetes” folder, as depicted in Figure 10.

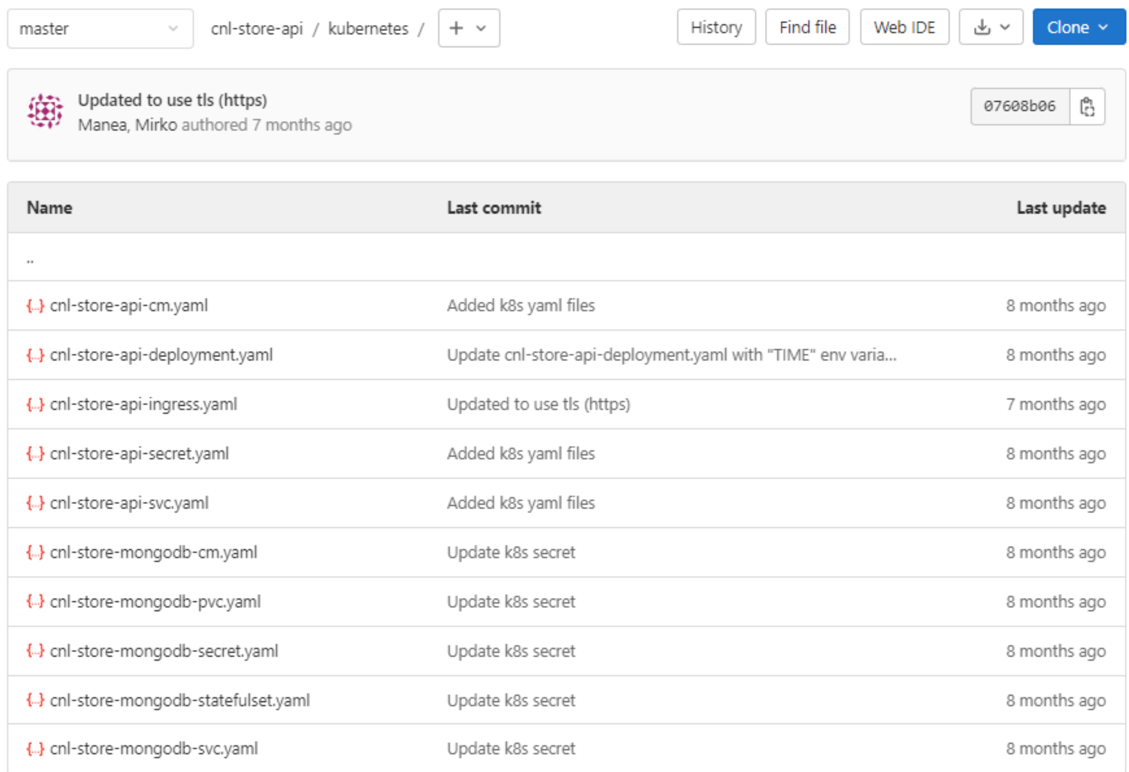


Figure 10. Example: CNL Store API kubernetes manifests

To start up the entire tool run the following commands:

```
$ kubectl apply -k ./CNL Store API/Kubernetes
$ kubectl apply -k ./CNL Editor API/kubernetes
$ kubectl apply -k ./CNL Editor/kubernetes
$ kubectl apply -k ./CNL Vocabulary/kubernetes
$ kubectl apply -k ./CNL Editor Frontend/kubernetes
```

4.2.3 User Manual

In this section the use of the CNL Editor and its functionalities are explained.

The user can access the CNL Editor through the MEDINA UI:

https://integrated-ui-test.k8s.medina.esilab.org/ [internal use only - authentication required]

by selecting on the left menu “Rules and Obligations”, as shown in Figure 11.

(cc) BY-SA

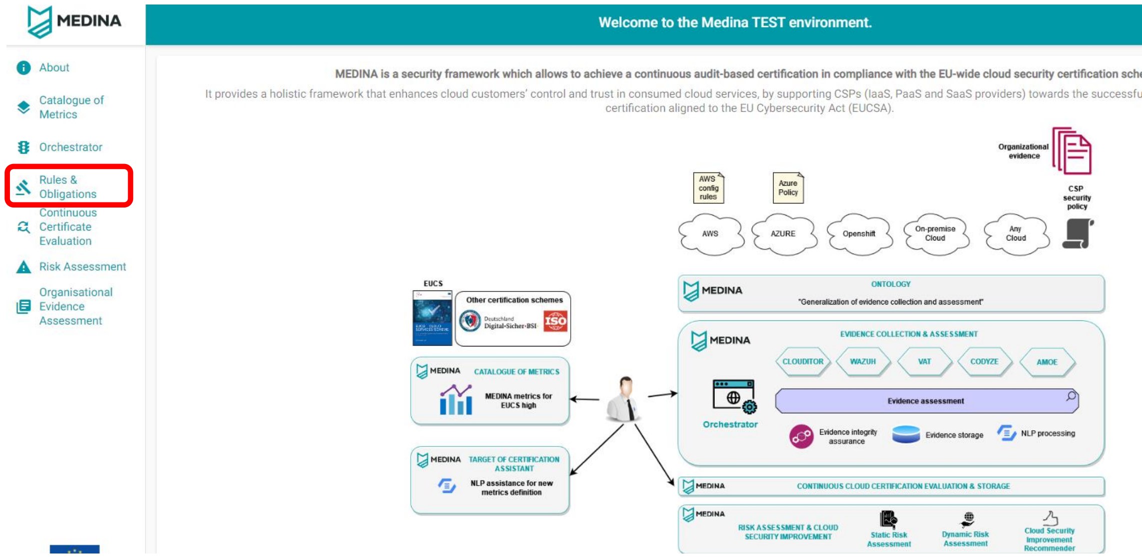


Figure 11. Accessibility of the CNL Editor from the MEDINA UI

The CNL Editor presents all REO objects created and associated with the user (see Figure 12).

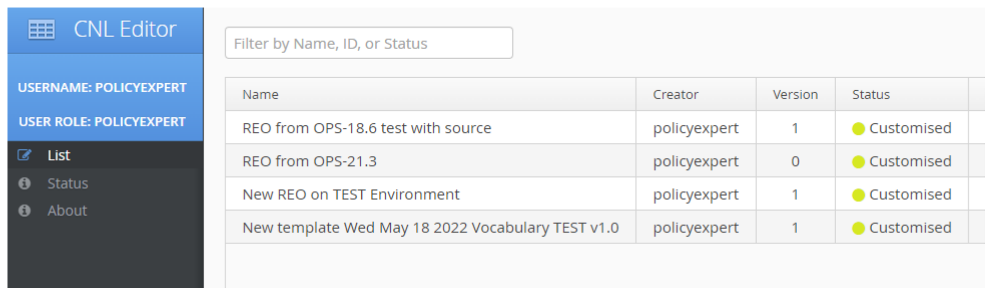


Figure 12. List of REOs available for the user POLICYEXPERT in the CNL Editor

REO selection

The user can select a REO from the list and choose an operation on it: Show, Edit, Copy, Raw, Map or Delete (see Figure 13).

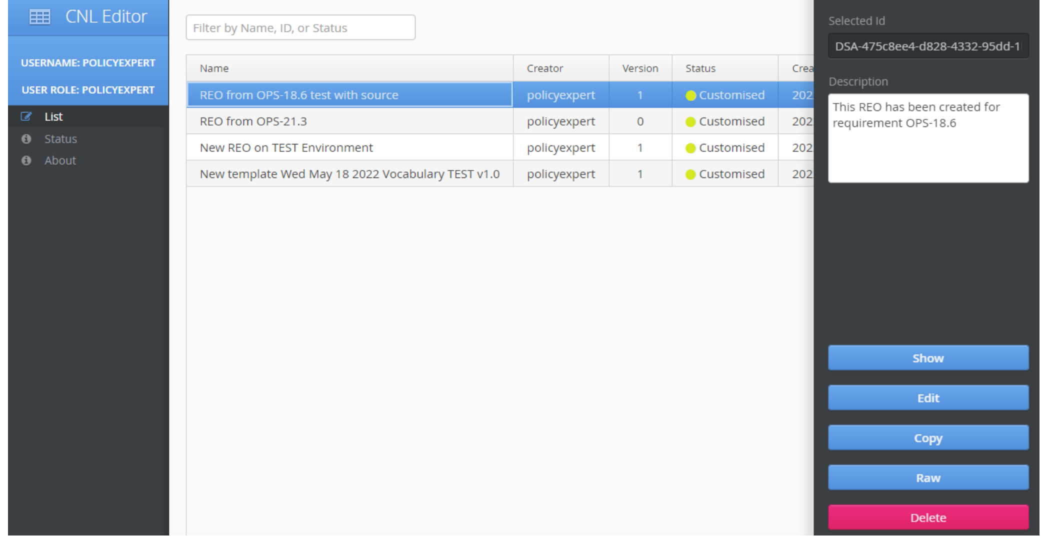


Figure 13. List of available operations for a selected REO in the CNL Editor.

CC BY-SA

Show a specific REO

When a user selects "Show" on a specific REO, a page is displayed with the REO data. This feature allows the user to visualize the Requirement specification for that REO, the list of Obligations (Policies section) with details and their source (Catalogue of controls and metrics or Recommender). Figure 8 shows an example of REO in the CNL Editor GUI.

Edit a specific REO

When a user selects "Edit", the page reported in Figure 14 is displayed, showing the Requirement and Obligation list, where the user can change the value for the target value of an obligation or she/he can delete the obligation itself. This functionality allows the user to refine/adapt the REO to a specific Cloud Service Provider or Cloud Service or Resource specific needs with respect to a generic obligation.

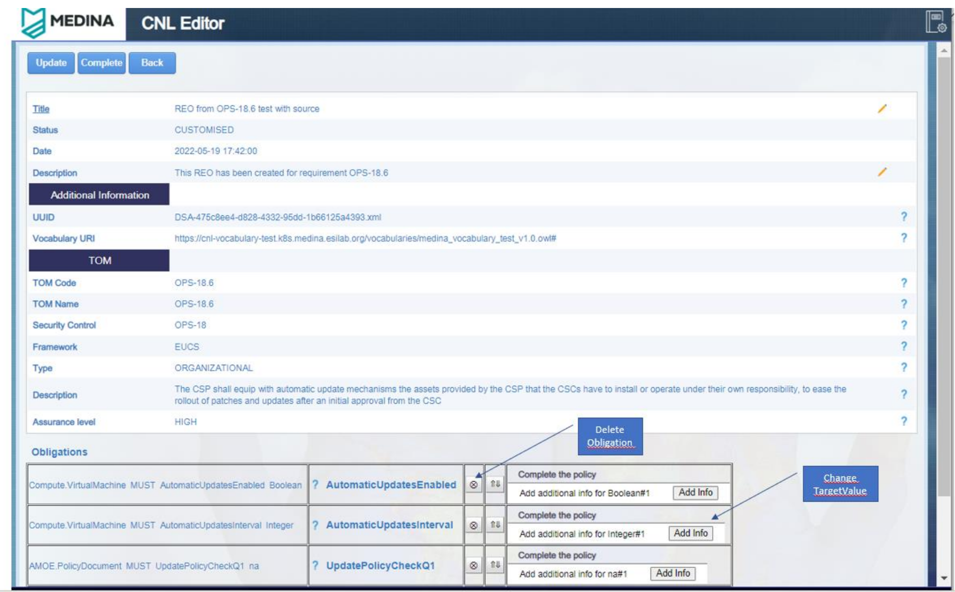


Figure 14. Editing a REO object in the CNL Editor

Map a specific REO

With this selection, the REO is mapped, i.e., the CNL Editor calls the DSL Mapper that will generate, for this REO, the correspondent Rego code (see Section 5).

CNL Editor APIs

The CNL Editor APIs (see Figure 15) are accessible through the following link: https://cnl-editor-api-test.k8s.medina.esilab.org/swagger-ui.html#/reo-operations-controller

(cc) BY-SA

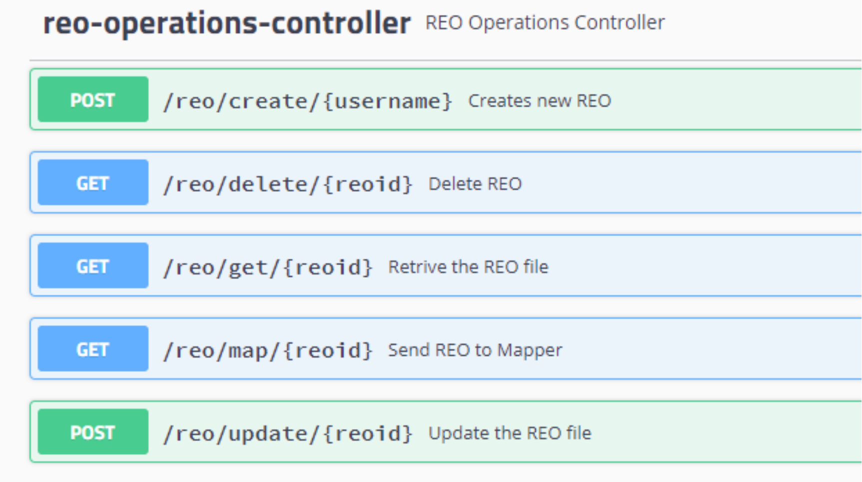


Figure 15. CNL Editor APIs

4.2.4 Licensing information

The CNL Editor is close source/proprietary code (Copyright by HPE).

4.2.5 Download

The CNL Editor is stored in a private GitLab repository and not in a public repository since it is not Open Source.

The URL of the private GitLab repository URL is as follows:

https://git.code.tecnalia.com/medina/wp2/task_2.4/cnl-editor-tools [internal use only - authentication required]

4.3 Advancements within MEDINA

During the second year of the MEDINA project, several changes have been implemented in the CNL Editor component:

- A Vocabulary has been set for the MEDINA ontology and then updated to reflect the changes in the Catalogue of controls and metrics (Editor Ontology).
- The XML structure of the files managed by the CNL Editor (REO objects) has been redesigned according to the two other components that use it (NL2CNL Translator and DSL Mapper) to accommodate the project requirements.
- The API list and interface have been revised to manage REOs and the new XML format has been introduced.
- The CNL Editor module has been revised to implement the authentication with Keycloak[12], to be compatible with the new XML format and to invoke the DSL Mapper. Furthermore, its interface has been redesigned to be more user friendly and to support new fields.
- Jenkins, GitLab and K8s integrations have been implemented for all modules.

[12] https://www.keycloak.org

(cc) BY-SA

4.4 Limitations and future work

Next steps for the rest of the project will be the improvement of the CNL Editor frontend look and feel and the update and refinement of the CNL Vocabulary and CNL Editor APIs to deal with the requirements defined in the August 2022 draft candidate version of the EUCS scheme [9].

CC BY-SA

5 DSL Mapper

The DSL Mapper is a component of the MEDINA framework that has the aim of mapping the obligations expressed in CNL into executable policies expressed in DSL. In particular, the obligations resulting from the previous steps are embedded in an XML object, called REO, while the output generated by the DSL Mapper is expected to be compliant with the DSL chosen in MEDINA, i.e., the Rego language[13].

The Rego language allows the creation of policies that can be used to automatically assess evidence, collected by the other tools (the evidence collectors are described in detail in deliverable D3.5 [4]).

When evaluating Rego policies, three elements are considered:

- An *input*, i.e., the request.
- A *policy*, also called rule.
- (Optional) *external data*.

The *input* is then checked against the *policy* to evaluate if it complies, possibly considering the *external data*.

In MEDINA, Rego policies are used according to the following: the evidence collectors create evidence that need to be assessed using the MEDINA metrics. For example, evidence that describes an object storage may need to be assessed using a metric which defines that this storage needs to have a backup. This is a typical policy evaluation task: a request (the evidence describing the storage) is provided and needs to be checked for compliance with a pre-defined policy (the metric), sometimes using also external data (the target value and the operator of the metric). The role of the DSL Mapper is to generate the description of the policies and the description of the external data, while the inputs are provided by the evidence collectors.

5.1 Implementation

5.1.1 Functional description

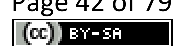
The DSL Mapper interacts with other components of the Cloud Security Certification Language system and of the general MEDINA framework. Specifically, its functionality is triggered by the CNL Editor, through its UI, thus the DSL Mapper uses the information stored in the CNL Store as source of data. Finally, after performing the mapping of obligations into Rego rules, the output is sent to the Orchestrator in order to be further processed.

Related requirements

The following tables include the functional requirements (from deliverable D5.2 [6]) related to this component, together with a description of how and to what extent these requirements are implemented at this point of development.

Requirement id	DSLM.01
Short title	Translation to selected DSLs
Description	The controlled natural language output of NL2CNL translator and further edited - when needed - with the CNL editor, will be semi-automatically mapped (meaning, with little human intervention) to the enforceable languages (aka, Domain Specific Languages, DSLs) inputs to tools such as Clouditor, or whatever will be the chosen DSL in MEDINA.

[13] https://www.openpolicyagent.org/



Implementation state	Partially implemented

The DSL Mapper aims at mapping automatically a set of obligations into executable policies, expressed in a DSL. These policies are then used by the Orchestrator to evaluate a certain TOM. The DSL chosen for representing policies in MEDINA is the Rego language. The current implementation of the DSL Mapper prototype allows the mapping of a subset of obligations into Rego rules. This functionality will be extended in the future by allowing the DSL Mapper to map all the available obligations.

Requirement id	**DSLM.02**
Short title	Mapping elements
Description	The mapping process will take into account relevant elements of the target certification framework, including (some) technical and organizational measures, quantitative/qualitative security metrics, complex compliance conditions, and cloud supply chain elements. The mapping process will prioritize the translation of those requirements in CNL that can automatically be enforced by WP4 and that are considered highly relevant by the EU authorities at stage.
Implementation state	Discarded

This requirement is an aggregate of requirements that have already been considered in other components. Specifically, it is represented by requirements ECO.02 - Conformity to selected assurance level, CCCE.01 - Continuous Evaluation of Assessment Results, UC00.06 - Security Controls Translator, UC00.07 - Define Measurement Targets, UC00.17 - MEDINA complies to EUCS. Therefore, this requirement is no longer requested from the DSL Mapper.

Requirement id	**DSLM.03**
Short title	DSL output compliancy
Description	The tool will output REGO rules, compliant with the input required by the Orchestrator.
Implementation state	Partially implemented

As already introduced, the output expected by the DSL Mapper must be compliant with the Rego language and with the input required by the Orchestrator. Currently the Orchestrator needs two files to describe a Rego rule, namely the *metric.rego* (containing the policy) and the *data.json* (containing the external data). The current implementation of the DSL Mapper is able to generate only the *data.json* file. This functionality is being updated.

Main innovations

The main innovation of the DSL Mapper consists in providing a translation from a non-executable language (CNL) to an executable one (DSL). In fact, the output policies of the DSL Mapper, expressed in Rego code, are used as input for the MEDINA assessment tools, which can automatically evaluate these policies to verify the compliance of a CSP to a certification schema.

5.1.1.1 Fitting into overall MEDINA Architecture

Figure 16 highlights the DSL Mapper within the MEDINA architecture.

The DSL Mapper interacts with the CNL Editor, from which it receives the trigger to start the mapping. It also needs to access the CNL Store through the CNL Editor APIs to retrieve the REO

objects containing the information to be mapped. Finally, it interacts with the Orchestrator to send the results of the mapping, i.e., the Rego rules generated for each obligation.

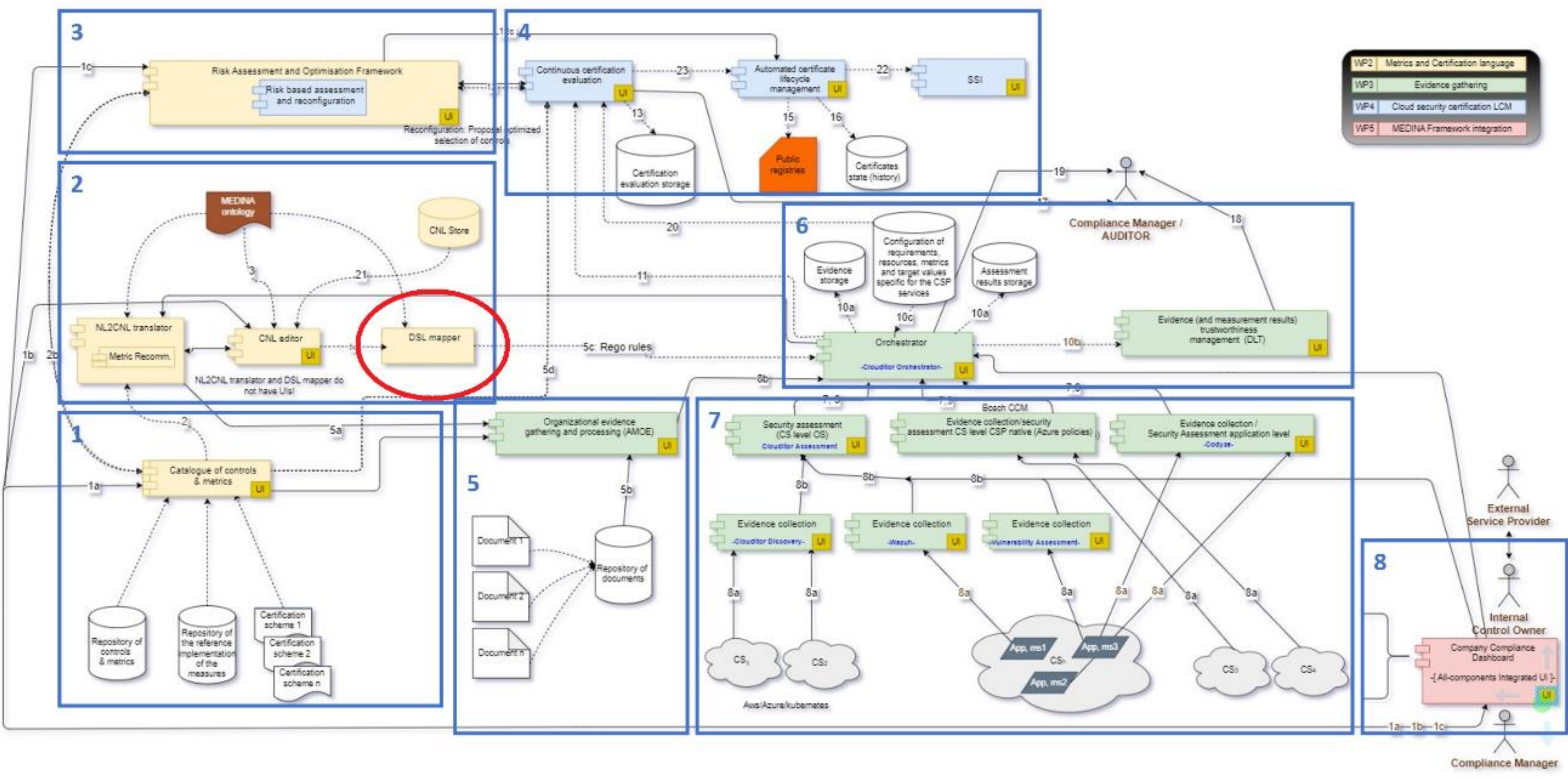


Figure 16. Position of the DSL Mapper within the MEDINA architecture (source D5.2 [6])

5.2 Technical description

This section describes the technical details of the implemented DSL Mapper prototype.

It is worth highlighting that the first stable version of the DSL Mapper was integrated into the implementation of the NL2CNL Translator, due to the fact that its development in month M18 was very limited. Thus, the DSL Mapper is not available in the current MEDINA framework deployed on the *test environment* of the Kubernetes cluster. Nevertheless, the development of the DSL Mapper is currently ongoing, thus in the following sections we will refer to this *work-in-progress* version of the DSL Mapper, being its code available within the public GitLab repository (https://git.code.tecnalia.com/medina/public/dsl-mapper).

Moreover, the development version of the DSL Mapper is deployed for testing purposes in the *development* environment of the Kubernetes cluster that hosts the MEDINA framework[14].

5.2.1.1 Prototype architecture

The DSL Mapper is written in Python 3, and it is organized into modules. Its functionalities are available through a REST API, thus similarly to the NL2CNL Translator, the FastAPI framework[15] has been used for implementing it. Figure 17 shows the architecture of the DSL Mapper.

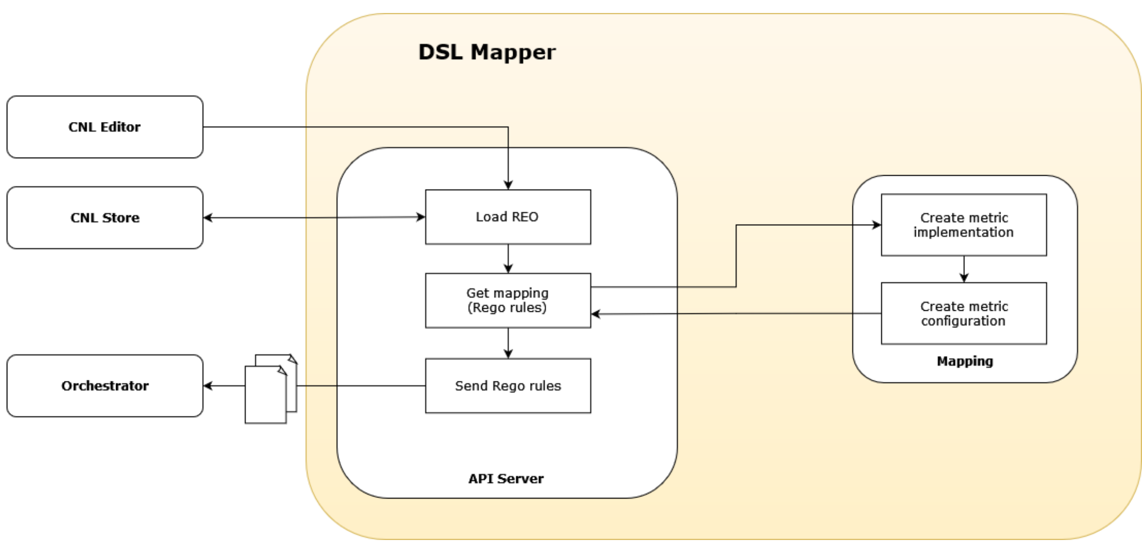


Figure 17. Overview of the DSL Mapper architecture

The available modules are organized in two components: the first is called API Server and is responsible for the API interface to other MEDINA components. Moreover, it coordinates all the DSL Mapper operations. The second is the Mapping component, which implements the generation of the Rego rules.

The endpoints available to interact with the DSL Mapper are reported in Table 7.

[14] https://dsl-mapper-dev.k8s.medina.esilab.org/docs#/
[15] https://fastapi.tiangolo.com

Table 7. List of available endpoints for the DSL Mapper component

Function name	Parameters	Return Type	Description
map_obligations _to_rego	reo_id	HTTP Response	This function is designed to be called by the CNL Editor, which must transmit as a parameter the identifier of the REO to be assessed. This is a POST function, since it generates a Rego rule for each obligation included in the specified REO. If successful, the status code returned will be of type 200, otherwise an error code will be returned depending on the problem occurred.
livez	None	HTTP Response	This function is used by the Kubernetes primary node agent to know when to restart a container. In fact, many applications running for long periods of time eventually transition to broken states, and cannot recover except by being restarted.
readyz	None	HTTP Response	This function is used by the Kubernetes primary node agent to know when a container is ready to start accepting traffic.

5.2.1.2 Description of components

This section presents the components available in the DSL Mapper and describes how they have been developed and will continue to be updated to meet the MEDINA requirements.

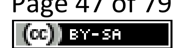
API Server

The API Server is the main component of the DSL Mapper, since it connects and coordinates with the other components. Its activity is performed with the following steps:

1. The server waits for requests from the CNL Editor. When a CNL Editor user is satisfied with the obligations associated to a REO, he/she invokes the DSL Mapper through the *Map* button. The identifier of the current REO object is passed as a parameter to the DSL Mapper.
2. The DSL Mapper loads the REO object from the CNL Store using the CNL Editor APIs, by using the REO identifier received from the CNL Editor.
3. The DSL Mapper calls the Mapping component to obtain the Rego rules corresponding to the obligations.
4. Finally, the Rego rules are sent to the Orchestrator to be further processed.

Mapping

The Mapping component aims at generating the Rego rules for the obligations associated to a certain requirement. As previously explained, for each obligation, the Orchestrator requires two input files, *metric.rego*, which includes a policy description, and *data.json*, which reports the operator and the target value associated to an obligation and specified by the user through the CNL Editor interface.



Let the reader consider, for example, the following obligation, associated to requirement OPS-05.3*: The CSP shall automatically monitor the systems covered by the malware protection and the configuration of the corresponding mechanisms to guarantee fulfilment of OPS-05.1*.

Evidence	Action	Metric	TargetValue Type	(Operator, TargetValue)
Compute.VirtualMachine	MUST	MalwareProtectionEnabled	Boolean	(=,true)

This obligation expresses that the evidence describing a Virtual Machine object needs to be assessed using a metric which defines that this virtual machine needs to have a malware protection mechanism enabled.

Thus, the Mapping component is expected to generate a *data.json* file containing the following information:

```
{
  "operator" : "==",
  "target_value" : true
}
```

The *metric.rego* file, which contains the policy to be assessed, will be similar to the following:

```
package clouditor.metrics.malware_protection_enabled
import data.clouditor.compare
default applicable = false
default compliant = false
enabled := input.malwareProtection.enabled
applicable {
      enabled != null
}
compliant {
      compare(data.operator, data.target_value, enabled)
}
```

The two files described above are generated by the Mapping component for each obligation it founds in the REO object under consideration.

5.2.1.3 Technical specifications

The DSL Mapper prototype is written in Python, version 3.8. A selection of the most important libraries is shown in the following and a full list including all the used libraries can be found in the GitLab repository[16].

- fastapi

[16] https://git.code.tecnalia.com/medina/public/dsl-mapper

- uvicorn
- pandas
- python-keycloak
- xmlschema
- xmltodict

5.3 Delivery and usage

The following sections give a short overview of the delivery and usage of the prototype.

5.3.1 Package information

The package is delivered in a repository hosted on GitLab. Even if it is not intended to be used as a standalone tool, it is possible to use it for testing purposes by accessing its APIs through a web browser. The component is available as a Docker image. The main folders and files are reported in Table 8.

Table 8. Most important files and folders implementing the DSL Mapper

Folder/file	Description
api_server.py	This file contains the code needed for implementing the APIs and for managing the communication between components.
Dockerfile	This file contains the list of commands that the Docker client calls while creating an image.
config.py	This file includes all the configurations (variables, paths, etc.) used by the prototype.
app_utils/	This folder includes a set of utility functions.
clouditor_orchestrator_client_legacy/	This folder contains the client to interact with the Orchestrator component.
cnl_editor_client/	This folder contains the client to interact with the CNL Editor component.
openapis/	This folder contains the auto-generated OpenAPI files.

5.3.2 Installation instructions

The full up-to-date installation instructions can be found in the README at the DSL Mapper repository on GitLab at this link: https://git.code.tecnalia.com/medina/public/dsl-mapper/-/blob/main/README.md

To setup the prototype locally, it is possible to build the Docker image with the following command:

```
$ docker build -t dsl_mapper_image .
```

5.3.3 User Manual

As stated before, the DSL Mapper is not intended to be used as a standalone tool, since it needs to strictly interact with the CNL Editor to function properly. In particular, the main endpoint available in the DSL Mapper takes as parameter the identifier of a REO object saved into the CNL

Store. At this stage of development of the prototype, the DSL Mapper cannot be easily configured to work as a standalone tool, since it depends on the output of the CNL Editor.

In any case, it is possible to run the Docker image with the following command:

```
$ docker run -p 8000:8000 -it –name dsl_mapper dsl_mapper_image
```

The generated container can be started and stopped with the following commands:

```
$ docker start dsl_mapper
```

```
$ docker stop dsl_mapper
```

The DSL Mapper provisional APIs can be tested through the command line by exploiting the CURL command, or through a browser by using the interactive APIs. Nevertheless, the user will receive an error message when trying to map a REO object, due to the missing input file that is taken from the CNL Store of the CNL Editor.

5.3.4 Licensing information

The DSL Mapper prototype is open source, under the Apache License 2.0.

5.3.5 Download

The code is currently available on MEDINA's git repository, on GitLab hosted by TECNALIA:

https://git.code.tecnalia.com/medina/public/dsl-mapper

5.4 Advancements within MEDINA

The prototype of the DSL Mapper has been implemented from scratch during the second year of the MEDINA project. The main advancements can be summarized as follows:

- The APIs has been defined and their management has been implemented, relying on the *fastapi* Python library.
- Two clients have been embedded in the prototype: The *Catalogue* client, which has been inherited from the NL2CNL Translator component and the *Orchestrator* client, which has been generated ex-novo.
- The connection with the CNL Editor has been implemented and tested and it works seamlessly, i.e., the CNL Editor is able to call the *mapping* endpoint and to get a response from it.
- The secure identity and access management to the DSL Mapper endpoints is guaranteed through the use of Keycloak[17].

5.5 Limitations and future work

The DSL Mapper is still in a preliminary stage of development. The creation of the Rego rules is still incomplete since the creation of the file containing the policy is missing. However, this part is under development and will be completed soon.

An additional problem to consider is the definition of the return values of the mapping function: currently, the DSL Mapper always returns the same status code, whereas it should vary depending on the success or failure of the mapping function, which in turn depends on the success or failure of the assessment by the Orchestrator.

[17] https://www.keycloak.org

BY-SA

As plan for the rest of the project, we will update the DSL Mapper to deal with the August 2022 draft candidate version of the EUCS scheme [9].

6 Conclusions

This document describes the main results achieved during the second year of the MEDINA project regarding the definition of the Cloud Security Certification Language, a machine-readable language in which the policies derived from the EUCS Cloud Certification Requirements are expressed and fed into the MEDINA assessment tools. The main achievements are as follows:

- The definition and the implementation of the NL2CNL Translator, for the representation of Cloud Security requirements into the MEDINA CNL.
- The definition and the implementation of the CNL Editor, a tool that allows users to visualize -and modify part of- the policies elicited for security controls.
- The definition and the implementation of the DSL Mapper, a tool that can translate the generated CNL to an enforceable machine-oriented Domain Specification Language.

For each tool, this document described its purpose and scope, the current coverage of MEDINA requirements, the component's internal architecture and its subcomponents, the relation to other components, the implementation state at the time of writing this deliverable, and technical details of the component, including the programming languages, and used libraries, information about the packaging and installation of the component, and licensing. Moreover, the advancements within the MEDINA framework are highlighted and the planned steps for the future are described.

The plan for the next months is to: 1) bring forward the implementation of the prototypes and their integration with other components of the MEDINA framework, 2) achieve full coverage of the prototypes' requirements, and 3) update the prototypes to deal with the requirements of the August 2022 draft candidate version of the EUCS scheme.

7 References

[1] MEDINA Consortium, "D2.1 - Continuously certifiable technical and organizational measures and catalogue of cloud security metrics-v1," 2021.

[2] ENISA, "EUCS – Cloud Services Scheme," [Online]. Available: https://www.enisa.europa.eu/publications/eucs-cloud-service-scheme. [Accessed October 2022].

[3] MEDINA Consortium, "D2.3 - Specification of the Cloud Security Certification Language v1," 2021.

[4] MEDINA Consortium, "D3.5 - Tools and techniques for collecting evidence of technical and organisational measures – v2," 2022.

[5] MEDINA Consortium, "D5.1 - MEDINA Requirements, Detailed architecture, DevOps infrastructure and CI/CD and verification strategy-v1," 2021.

[6] MEDINA Consortium, "D5.2 - MEDINA Requirements, Detailed architecture, DevOps infrastructure and CI/CD and verification strategy-V2," 2022.

[7] MEDINA Consortium, "D3.2 - Tools and techniques for the management of trustworthy evidence v2," 2022.

[8] MEDINA Consortium, "D6.3 - Use cases development and validation - prototypes-v1," 2022.

[9] ENISA, "EUCS – Cloud Services Scheme (2022)," Draft version provided by ENISA (August 2022) - not intended for being used outside the context of MEDINA.

[10] A. Veizaga, M. Alferez, D. Torre, M. Sabetzadeh and L. Briand, "On systematically building a controlled natural language for functional requirements," *Empirical Software Engineering,* vol. 26, no. 79, 2021.

[11] D. Méndez Fernández and others, "Naming the Pain in Requirements Engineering: Contemporary Problems, Causes, and Effects in Practice," *Empirical Software Engineering,* vol. 22, pp. 2298-2338, 2017.

[12] K. Pohl, Requirements engineering - fundamentals, principles, and techniques, 2010.

[13] C. Arora, M. Sabetzadeh, L. Briand and F. Zimmer, "Automated extraction and clustering of requirements glossary," *IEEE Trans Software Eng,* vol. 43, no. 10, pp. 918-945, 2017.

[14] T. Khun, "A Survey and Classification of Controlled Natural Languages," *Computational Linguistics,* vol. 40, no. 1, pp. 121-170, 2014.

[15] R. Schwitter, "Controlled Natural Language for Knowledge Representation," in *COLING* , 2010.

CC BY-SA

[16] A. Wyner, “On controlled natural languages: Properties and prospects,” in *Controlled natural language*, 2009.

[17] K. Pohl and C. Rupp, Requirements engineering fundamentals - a study guide for the certified professional for requirements engineering exam: Foundation Level - IREB compliant, 2011.

[18] A. Mavin, P. Wilkinson, S. Gregory and E. Uusitalo, “Listens learned (8 lessons learned applying EARS).,” in *In: 24th IEEE international requirements engineering conference*, 2016.

[19] S. Withall, Software requirement patterns, Pearson Education, London, 2007.

[20] M. Riaz, J. King, J. Slankas and L. Williams, “Hidden in plain sight: automatically identifying security requirements from natural language artifacts,” in *IEEE 22nd international requirements engineering conference*, 2014.

[21] C. Denger, B. DM and E. Kamsties, “Higher quality requirements specifications through natural language patterns.,” in *2003 IEEE International conference on software - science, technology and engineering (SwSTE 2003)*, 2003.

[22] J. Eckhardt, A. Vogelsang, H. Femmer and P. Mager, “Challenging incompleteness of performance requirements by sentence patterns,” in *24th IEEE international requirements engineering conference*, 2016.

[23] I. Matteucci, M. Petrocchi and M. Sbodio, “CNL4DSA: a controlled natural language for data sharing agreements,” in *Symposium of Applied Computing*, 2010.

[24] N. Fuchs, K. Kaljur and T. Kuhn, “Attempto Controlled English for knowledge representation,” in *Reasoning Web — 4th International Summer School 2008, number 5224 in Lecture Notes in Computer Science*, 2008.

[25] G. Hart, M. Johnson and C. Dolbear, “Rabbit: Developing a Control Natural Language for Authoring Ontologies,” in *European Semantic Web Conference*, 2008.

[26] A. Cregan, R. Schwitter and T. Meyer, “Sydney OWL syntax - Towards a controlled natural language syntax for OWL 1.1,” in *CEUR Workshop Proceedings 258*, 2007.

[27] S. Konrad and B. Cheng, “Real-time specification patterns,” in *27th International conference on software engineering (ICSE 2005)*, 2005.

[28] A. Post, I. Menzel and A. Podelski, “Applying restricted english grammar on automotive requirements - does it work? A case study.,” in *Requirements engineering: foundation for software quality*, 2011.

[29] M. Abadi, “Logic in access control,” in *LICS*, 2003.

[30] A. Arenas, B. Aziz, J. Bicarregui and M. Wilson, “An Event-B approach to data sharing agreements,” in *IFM, LNCS 6396*, 2010.

[31] Q. Ni and e. al, “Privacy-aware Role-based Access Control,” *ACM Transactions on Information and System Security,* 2010.

[32] R. De Nicola, G. Ferrari and R. Pugliese, "Programming Access Control: The KLAIM Experience," in *CONCUR 2000. LNCS, vol. 1877*, 2020.

[33] R. Hansen, F. Nielson, H. Nielson and C. Probst, "Static validation of licence conformance policies," in *ARES*, 2008.

[34] K. Larsen and B. Thomsen, "A modal process logic," in *Third Annual Symposium on Logic in Computer Science*, 1988.

[35] J. Bergstra, A. Ponse and S. Smolka, Handbook of Process Algebra, Elsevier, 2011.

[36] Amass Consortium D3.6, "Prototype for Architecture-Driven Assurance (c)," 2018.

[37] "Wikipedia - BERT (language model)," October 2021. [Online]. Available: https://en.wikipedia.org/wiki/BERT_(language_model). [Accessed October 2022].

[38] J. Devlin, M.-W. Chang, K. Lee and K. Toutanova, *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,* 2018.

[39] "Wikipedia - Word2vec," October 2021. [Online]. Available: https://en.wikipedia.org/wiki/Word2vec. [Accessed October 2022].

[40] "FastText," October 2021. [Online]. Available: https://fasttext.cc/. [Accessed October 2022].

[41] "Wikipedia - Stop word," [Online]. Available: https://en.wikipedia.org/wiki/Stop_word. [Accessed October 2022].

[42] "Common crawl," [Online]. Available: https://commoncrawl.org/. [Accessed October 2022].

[43] "Wikipedia - PCA," [Online]. Available: https://en.wikipedia.org/wiki/Principal_component_analysis. [Accessed October 2022].

[44] "Wikipedia - TSNE," [Online]. Available: https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding. [Accessed October 2022].

[45] "scikit learn Truncated SVD," [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html. [Accessed October 2022].

[46] "Wikipedia - Evaluation measures (information retrieval)," [Online]. Available: https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval). [Accessed October 2022].

[47] "Wikipedia - DCG," [Online]. Available: https://en.wikipedia.org/wiki/Discounted_cumulative_gain. [Accessed October 2022].

[48] C. Banse, I. Kunz, A. Schneider and K. Weiss, "Cloud Property Graph: Connecting Cloud Security Assessments with Static Code Analysis," in *IEEE 14th International Conference on Cloud Computing (CLOUD)*, 2021.

(cc) BY-SA

APPENDIX A: The Cloud Certification Language

1. Motivation

Cloud certification schemes consist of a set of rules, technical requirements, standards, and procedures to strengthen the cybersecurity of ICT services and products offered to citizens, and their terms and conditions are usually published in natural language. This is the case, e.g., of the EUCS draft candidate Cloud Certification Scheme [9]. Thus, a rigorous translation procedure is required to produce a machine-readable format out of textual NL requirements. This translation should minimise as much as possible human intervention - which is prone to errors and time consuming.

The Cloud Certification language is the language that the MEDINA consortium is developing in the course of the project, a language which will permit to express rules for cloud certification, in a uniform way and without the ambiguity of natural language (the latter being natively more complex). In addition, this language will be machine readable and will be the input of the MEDINA Assessment Tools. The methodology followed by the consortium to achieve the ultimate Cloud Certification Language is defined in next section.

2. Methodology

For a lean and seamless *trait d'union* between what is dictated by official documents of the European Commission in terms of certification and the definition of the MEDINA DSL, we intend to proceed as follows:

1) Semi-automatically translate Natural Language (NL) certifications terms and conditions, as they appear on official documents like the EUCS scheme, into policies expressed in a Controlled Natural Language (CNL).
2) Visualize and possibly revise the generated CNL via a CNL editor tool, to verify the generated policy statements before proceeding with the mapping to the MEDINA DSL.
3) Map the CNL to a runtime-enforceable DSL language that can be used by the MEDINA assessment tools to check the compliance status of the certification terms and conditions.

Figure 2 highlights the components of the MEDINA architecture that constitute the building blocks to achieve the Cloud Certification Language:

- The NL2CNL translator translates EUCS NL requirements into their MEDINA CNL representation. Part of the translation from NL to CNL will be done via NLP techniques. The CNL translator is the main output of MEDINA task 2.3.
- The CNL Editor is the user interface that allows users to visualize and possibly revise the translation of the requirements into the MEDINA CNL. The CNL editor is the main output of Task 2.4.
- The DSL Mapper is the MEDINA component that maps the yet not executable MEDINA CNL into the MEDINA Domain Specific Language (DSL), whose statements are instead machine-readable. The DSL is the Cloud Certification Language. The DSL mapper is the main output of Task 2.5.

All the components interface with the Catalogue of controls and metrics, described in Deliverable D2.1 [1]. For the representation of the requirements, all the components rely on the MEDINA Ontology, see *APPENDIX D: MEDINA Vocabularies and Ontologies*.

APPENDIX B: Patterns and Controlled Natural Languages for Requirements specifications

This section provides an introduction to Controlled Natural Languages (CNLs) and their benefits in various fields of applications, including the MEDINA scenario.

Quoting from [10], `*natural language (NL) is pervasive in [...] requirements specifications (RSs). However, despite its popularity and widespread use, NL is highly prone to quality issues such as vagueness, ambiguity, and incompleteness*.' Work in [11] identifies common issues affecting usability of NL requirements, like, e.g., the adoption of an unclear – or a too complex, or too poor – terminology, the missing of a requirements specification template, the duplication of requirements that simply use a diverse wordiness, the omission of significant descriptions, the vagueness and the ambiguity of terms and meaning.

With the aim of improving the quality of NL requirements, by, e.g., reducing its ambiguity and making them more precise and complete, Pohl proposes the following three approaches [12]:

- **Glossaries**. Requirements glossaries are lists of specific words that make explicit and provide definitions for the salient terms in a RS. Glossaries may also provide information about the synonyms, related terms, and example usages of the salient terms [13].
- **Patterns**. They are pre-defined sentence structures with optional and mandatory components. Patterns guide stakeholders in writing more standardized NL requirements by restricting their syntax, and thus possibly avoiding omissions and/or over-specifications.
- **Controlled natural languages**. Controlled natural languages (CNLs) are a subset of natural languages, specifically conceived to make language processing simpler. A CNL is, in essence, a developed language that is based on natural language, but it is more restrictive in terms of lexicon, syntax, semantics, while at the same time retaining most of its natural properties. CNLs prevent quality problems in requirements documents, while maintaining the flexibility to write and communicate requirements in an intuitive and universally understood manner [14]. CNLs are considered an extension of the pattern category which, in addition to restricting the syntax (the grammatical structures), also provide language constructs with which it is possible to precisely define the semantics of NL requirements. By adopting a contrived representation, in terms of grammar and vocabulary, CNLs may reduce the ambiguity and complexity of a complete language [15], e.g., English, Spanish, French, Swedish, Mandarin, etc. [16].

In the following, we will concentrate on the last two approaches (patterns and CNLs), being language representations the target of this section.

1. Patterns

Many patterns have been proposed in the literature. In [17], Pohl and Rupp present a single pattern to specify functional software requirements, while Mavin et al., in [18], consider the aviation domain and propose a set of fine patterns to describe functionalities of the domain. Requirements writing according to these patterns has been proved to be easier to understand and less ambiguous.

Both functional and not functional requirements have been presented in [19] by Withall, all related to the business domain, while Riaz et al. in [20] define a set of patterns expressing security requirements. The latter comes with an assistant tool that helps the user in selecting the appropriate pattern according to the security aspects relevant in the NL requirement.

Among others, we cite here two other works specifying patterns for the description of embedded systems (Denger et al. [21]) and performance requirements (Eckhardt et al. [22]).

From this non-exhaustive review of existing patterns for drafting NL requirements, we can see that many of them are domain-dependent, that is, laced with terms specific to a certain domain.

2. Controlled Natural Languages

CNLs have been proved to be effective in mitigating linguistic ambiguity challenges, as they can easily be translated into a formal language such as first-order logic or different version of description logic, automatically and mostly deterministically (Schwitter, [15]). CNLs are formal *per se*, being born with an associated formal semantics. In Section 3, we will give an example of the semantics for a language ideated by some members of the MEDINA consortium and conceived for expressing privacy regulations (Matteucci et al, [23]). In general, CNLs can conveniently express the kind of information that occurs in, e.g., software specifications, formal ontologies, business rules, legal and medical regulations.

One interesting feature of CNLs is that they usually maintain a readability that is not so different from that of pure natural languages. This makes them easier to write and understand by people than pure formal languages. Furthermore, they precisely define subsets of natural languages, have a formal foundation, and can therefore be adopted for automated reasoning (Schwitter, [15]).

CNLs have been proposed and used in many application domains and for different purposes. For example, the Attempto Controlled English (ACE)[18] [24] is a CNL that defines a subset of the English language intended to be used in different domains, such as software specification and the Semantic Web. Attempto can be automatically translated into first-order logic and it is supported by a number of tools, like a Parsing Engine to translate ACE texts into a variant of first-order logic, or a plug-in for the Protége ontology editor[19].

In [25], Hart et al. propose Rabbit as a way to overcome the impediment to the creation and adoption of ontologies. Rabbit is a CNL that can be translated into the Ontology Web Language[20] (OWL) in a way that achieves both comprehension by domain experts and computational preciseness. In a sense, Rabbit can be defined as complementary to OWL, supporting the need to author and understand domain ontologies but overcoming the difficult comprehension of descriptions logics.

Similarly, the idea behind the Sidney OWL Syntax SOS [26] is to propose a new syntax that can be used to write and read OWL ontologies in CNLs. SOS enables the generation of grammatically correct full English sentences to and from OWL syntax. This enables users to write an OWL ontology in a defined subset of English, also improving readability and understanding of OWL statements.

Regarding CNLs developed for specific domains, as an example Konrad and Cheng in [27] consider the automotive domain and propose a language to express real-time properties of systems under specification. As a follow-up of the same language, Post et al in [28] enlarge its expressivity to express other requirements in the same domain.

Just like other types of requirements, EUCS controls and TOMs are expressed in natural language. Although the use of natural language enables users (e.g., CSPs) to read and

[18] http://attempto.ifi.uzh.ch/site/
[19] https://protege.stanford.edu/
[20] https://www.w3.org/OWL/

understand the requirements specific for Cloud Security Certification, a key issue relies in the fact that NLs are not machine readable, and automatic measurements on whether controls and TOMs are actually going to be processed and fulfilled are not feasible. In particular, NL cannot be used as the input language for a rule-assessment software infrastructure to be used for automated, continuous evaluation of TOMs' fulfilment. In fact, such an evaluation requires inputs in a machine-readable form, like, e.g., the de facto standard XACML moving to the field of access control rules.

Like other (cloud) certification schemes, the EUCS scheme controls are groups of statements – the TOMs indeed - that can be likened to policies. There are several examples of CNLs coined by Academia and Industry to express such policies. In the following, we introduce the reader to languages expressing policies for data management, including the one that has inspired our MEDINA CNL.

3. CNLs for expressing policies for secure data management

In the last two decades, data protection has been discussed in many scenarios, from critical infrastructures to social networks, just to cite two fields of interest. Informally, regulations for data protection, data storage and data sharing are written in Natural Language[21,22]. A proper CNL is a flexible mean to fill the gap between a traditional legal contract regulating the sharing of data among different domains, and the software architecture supporting it.

Some examples of languages and associated tools for secure data management are as follows.

- Binder [29] is an open logic-based security language that encodes security authorizations among components of communicating distributed systems.
- The Rodin platform provides an animation and model-checking toolset, for analysing properties of specifications based on the Event-B language[23], able to express security data policies. In [30], the developers of Rodin and Event-B presented a formalization of data management clauses in Event-B, and a model checker is exploited to verify that a system behaves according to its associated clauses.
- Also, work in [31] proposes a comprehensive framework for expressing highly complex privacy-related policies, featuring purposes and obligations. The Klaim family of process calculi [32] provides a high-level model for distributed systems, for programming and controlling access and usage of resources. Also, work in [33] considers policies that moderate the use and replication of information, e.g., imposing that a certain information may only be used or copied a certain number of times. The analysis tool is a static analyser for a variant of Klaim.

We will now describe a language defined by some members of the MEDINA consortium in a previous work. The *Controlled Natural Language* for *Data Sharing Agreement* (CNL4DSA) was introduced with the purpose to reduce the barrier of adoption of data policies in terms of security and privacy as well as to ensure policies mapping to formal languages that allow the automatic verification of the policies [23]. A data sharing agreement is essentially a contract between two or more parties to agree on some terms and conditions with respect to data

[21] Justice Information Sharing, U.S. Dept. of Justice. Online: https://bja.ojp.gov/program/it
[22] North American Electronic Reliability Corporation. Electricity Information Sharing and Analysis Center. Online: https://www.nerc.com/pa/CI/ESISAC/Pages/default.aspx
[23] http://www.event-b.org/platform.html

sharing, storage, and usage. This language, called for the sake of brevity CNL4DSA, permits simple, yet formal, specifications of different classes of privacy policies, as listed below:

- *authorizations*, expressing the permission for subjects to perform actions on objects (e.g., data), under specific contextual conditions
- *obligations*, defining that subject are obliged to perform actions on objects, under specific contextual conditions.

Central to CNL4DSA is the notion of *fragment*, i.e., a tuple ***f* =<*s, a, o*>** where *s* is the subject, *a* is the action, *o* is the object. A fragment simply says that `subject *s* performs action *a* on object *o*'.

By adding the can/must constructs to the basic fragment, a fragment becomes an *authorization* or an *obligation*.

Fragments are evaluated within a specific context *c*, i.e., a predicate that characterizes factors such as user's roles, data categories, time, geographical locations, etc. Contexts are evaluated either as true or false. Simple examples of contexts are `subject hasRole CSP', or `object hasCategory CloudResource'. In order to describe complex policies, contexts are composable. A *composite context C* is defined inductively as follows

$$C = c \mid C \textbf{ and } C \mid C \textbf{ or } C \mid \textbf{not } C$$

Where **and**, **or**, and **not** are Boolean connectors.

The syntax of a composite fragment F_M is described by the following Backus-Naur Form BNF-like syntax:

$$F_M = nil \mid \textbf{mod}\ f \mid F_M; F_M \mid if\ C\ then\ F_M$$

where the modality **mod** ranges over {**can**, **must**} and the subscript *M* ranges over {*A*, *O}* where *A* stands for Authorization and *O* stands for Obligation. By changing the modality and the subscript, we get two different types of policies, namely we have authorizations and obligations. F_A is a composite authorization fragment and F_O is a composite obligation fragment.

Let us now comment on the individual policy constructors:

- *nil* does nothing.
- **mod** *f* is the atomic authorization/obligation fragment that expresses that f = <s,a,o> is allowed/obliged. Its informal meaning is that "*subject s can/must perform action a on object o"*.
- F_M ; F_M is a list of composite fragments. (Subscript *M* takes either only *O* or only *A*).
- *If C then* F_M expresses the logical implication between a composite context *C* and a composite fragment F_M: if *C* holds, then F_M is permitted/obliged (according to the value of *M*).

CNL4DSA has an operational semantics based on a Modal Transition System (MTS), able to express *admissible* and *necessary* requirements to the behaviour of the CNL4DSA specification. To model the behaviour of the specifications, modal transition systems are used. In its original version [34], MTS is a structure

$$(\mathcal{A}, \mathcal{S}, \rightarrow\Diamond, \rightarrow\boxdot)$$

where $\mathcal{S}$ is a set of specifications, like for example processes in the context of Process Algebras,

$\mathcal{A}$ is the set of actions which specifications may perform, and $\rightarrow\Diamond, \rightarrow\boxdot \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$ are the two modal transition relations expressing admissible and necessary requirements to the behaviour of the specifications.

In particular, $S \xrightarrow{a}\Diamond\ S'$ with S, S' $\in \mathcal{S}$ and $a \in \mathcal{A}$ means that it is admissible that the implementation of S performs a and then behaves like S'. Dually, $S \xrightarrow{a}\boxdot\, S'$ represents a transition in which the implementation of S is required to perform a and then behaves like S'.

This works under the assumption that all the required transitions are admissible transitions.

Figure 18 shows the operational semantics of F_A in terms of a modified label transition system $MTS_{Auth} = (\mathcal{AUT}, \mathcal{F}, \rightarrow\Diamond, \mathcal{C})$. As usual, rules are expressed in terms of a set of premises, possibly empty (above the line) and a conclusion (below the line).

Let $\rightarrow_\Diamond$ be the smallest subset of $\mathcal{AUTH} \times \mathcal{F} \times \mathcal{AUTH}$, closed under the following rules:

$$(can)\frac{}{can\, f \xrightarrow{f}_\Diamond nil} \qquad (if)\frac{C = true \quad F_A \xrightarrow{f}_\Diamond F'_A}{if\ C\ then\ F_A \xrightarrow{f}_\Diamond F'_A}$$

$$(;)\frac{F1_A \xrightarrow{f}_\Diamond F1'_A}{F1_A;F2_A \xrightarrow{f}_\Diamond F1'_A;F2_A}$$

Figure 18. Operational Semantics for the composite authorization fragment, where the symmetric rule for (;) is omitted (source: unpublished manuscript, Petrocchi M and Matteucci I.)

MTS_{Auth} deals with authorized transitions only and it considers also the set of contexts because the transitions may depend also on the value of such contexts, see rule (*if)* in Figure 18.

The introduction of $\mathcal{C}$ (= a set of predicates) in a labelled transition system is a standard practice [35]. We observe that the *if* operator implies the binding of variable appearing in the context *C.*

The operational semantics of F_O is expressed in terms of the modal transition system $MTS_{Obl} = (OBL, \mathcal{F}, \rightarrow\boxdot, \mathcal{C})$. The axioms and rules are similar to the ones presented for F_A apart from changing the transition relation, that becomes $\rightarrow\boxdot$.

4. MEDINA CNL

In this section, we introduce the definition of the MEDINA CNL, along with the motivations leading to this format.

We remind the reader that the scope of the certification requirements for MEDINA was defined in D2.1 (Section 3.1) [1] where we focused on a subset of EUCS requirements that were identified from the draft version of the EUCS scheme [9]. These requirements were selected based on two requisites: i) their 'assurance level' is 'high'; and ii) they include the wording "automatically monitor" or variations thereof.

Inspired by the presence of the authorization and obligation modalities in CNL4DSA, presented in the previous section, we further analysed the EUCS draft candidate certification scheme and were able to summarize the requirements, being either technical or organizational [7], in the following general textual formula:

The value assumed by the metric x on the resource of type y can/must be equal/major to/minor to the target value z/must fall in the range of values {z, ...w}.

Paraphrasing part of CNL4DSA, we define a MEDINA *fragment* as a tuple

f = <*rt, m, type(op,tv)*>

where *rt* is a resource type, *m* is a metric, *type(op,tv)* specifies the type of the metric, the designed target value *tv* for that metric, and the operator *op* which relates the value of the metric to *tv* (e.g., =, >, <, etc.).

A MEDINA fragment says that "the metric *m*, measured on the resource type *rt,* has a specific relation with the value *tv* of type *type*, based on the operator *op"*.

This leads to the following syntax for the MEDINA CNL:

$F_M = nil \mid \textbf{mod}\ f \mid F_M; F_M$

where *M* ranges over {*A*, *O*} (Authorization/Obligation) and

- nil does nothing.
- **mod** *f* is the atomic authorization/obligation MEDINA fragment that expresses that *f* = <*rt, m, type(op,tv)*> is allowed/obliged. Its informal meaning is that *"the metric m, measured on the resource type rt, can/must have a specific relation with the value tv of type type, based on the operator op".*
- F_M ; F_M is a list of composite MEDINA fragments.

Analysis of the EUCS certification scheme and connected metrics, set forth in D2.1 [1], have highlighted how the TOMs identified in month 12 and month 24 can be viewed as *obligations*. In particular, the CSP must fulfil specific obligations regarding, e.g., the security configurations of cloud resources.

For this reason, in month 24, the format for expressing in a more formal, but always readable way, the NL TOMs, is represented as follows:

must <*rt, m, type(op,tv)*>

With a little abuse of notation, we will express the MEDINA obligation in this format too:

RT* must *M type(op,tv)

where, as introduced above, ***RT*** is the resource type, ***M*** is a metric associated to the TOM, ***tv*** is a target value, ***op*** is the comparison operator, which indicates how to compare the target value with the value measured on the resource, with respect to metric ***M***; finally, ***type*** indicates the unit of measure of the target value and the measured value.

It is worth noting that, having included both the *must* and the *can* modalities in the MEDINA CNL, it will be possible – when and if necessary – to express not only obligation requirements but also authorization requirements. Also, it will be possible to express a list of composite MEDINA fragments.

APPENDIX C: From NL to CNL TOMs will show some concrete examples of TOMs rendered into the MEDINA obligations.

As a final note, it is fair to point out that other projects have developed more complex languages, and also equipped them with tools devoted to analysing properties of the system that such languages describe. This is the case, for example, of the AMASS (Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems)[24] European

[24] https://www.amass-ecsel.eu/

project. The target of AMASS is the specification of safety and security requirements for Cyber-Physical Systems, which probably feature many variabilities in their specification than requirements for cloud security certification.

AMASS includes multiple modelling languages according to the level of user expertise. They can be based on UML, associated with editor tools with many plug-ins or have formal foundation and as such can be given as specification to property verification tools (see AMASS Deliverable 3.6 `Prototype for architecture-driven assurance' [36]).

The language chosen to represent MEDINA's requirements is a much simpler language, with no correlated analysis tools. However, we argue that the language is suitable for MEDINA's purpose, which is to create a close-to-standard language for the representation of cloud certification requirements, and which is then made machine readable and input to the assessment tools specified in D3.2 [7].

APPENDIX C: From NL to CNL TOMs

The main goal of passing from a NL representation to a CNL representation of the security requirements and associated metrics is to help achieving the automatic and continuous monitoring of the EUCS scheme.

This process is carried out through two consecutive steps. The first step consists in associating each requirement with one or more predefined metrics. In fact, to get a compliance status, each requirement needs to be objectively evaluated and thus it should be associated with metrics that reflect the technical details of the requirement itself. The second step is represented by the translation of each requirement / associated metric into a policy, expressed in CNL. The two steps will be detailed in the following sections. In particular, the first section describes the proposal of a system to automatically associate metrics to requirements, whereas the second one reports the actual translation of requirements/metrics into policies. We remind the reader that, for the scope of the certification in month 12 and 24, the investigated policies consist in obligations. This does not preclude the fact that it is possible to also express permissions since the MEDINA CNL has both modalities, *must* and *can*.

Figure 19 depicts the simplified workflow of Task 2.3: first, for every requirement, a set of metrics is recommended/predicted, then the set is translated into the defined CNL. The output is stored in a dedicated database managed by the CNL Editor, since it will be used by the Editor itself to visualize/refine the result of the recommendation.

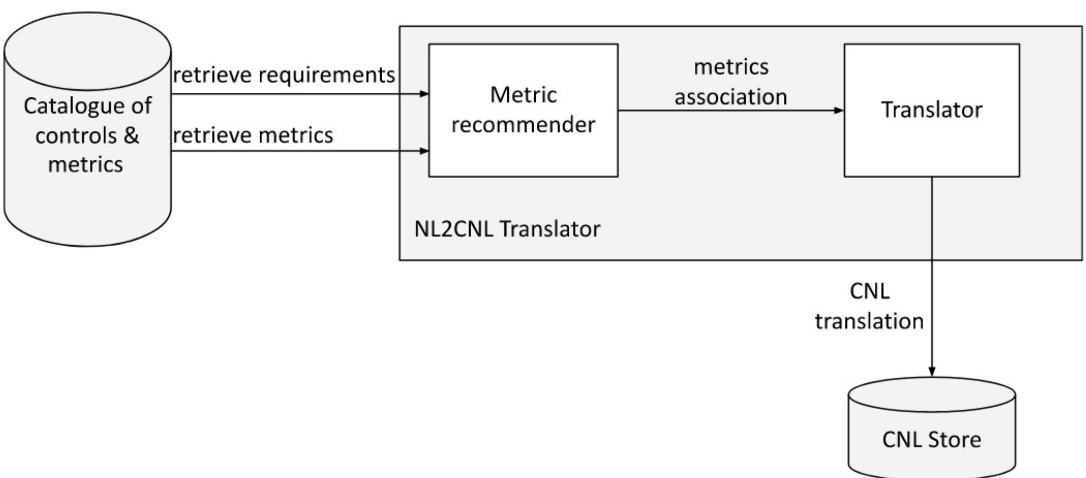


Figure 19. From Natural Language to Controlled Natural Language: Simplified overview (source: MEDINA's own contribution)

1. Metric association

This section explains the process carried out to automatically associate metrics to requirements. The idea behind this proposal comes from the fact that the metrics should be reusable when adding new requirements and the manual association between metrics and requirements is a time-consuming process. For these reasons, a tool for automatic association is proposed, namely a "metric recommender" system. This step is constructed as a research project and therefore single parts might change in the future. This means that the input data, or to be more exact the computed features, will be selected based on what produces the best results, which is an iterative research process.

However, the basic workflow should be fairly consistent over time. To this aim, input data (requirements and metrics, both expressed in NL) are selected from the MEDINA catalogue

database and fed into the recommender system. The idea is to associate metrics to a requirement according to the similarity of their descriptions. To do so, both requirements and metrics descriptions are represented into vectors of numerical features by relying on NLP techniques, with the hypothesis that similar requirements and features descriptions will reside in the same local area in the feature space. The quality of the process is actively supervised in visual analysis and experiment results are computed using state of the art metrics. The following subsections will give more details about this process.

1.1. Data and features

The input of the metric recommender system is taken from the catalogue, which currently includes the requirements available in the EUCS scheme [9]. In particular, the first input is represented by the natural language description of the requirements. The first experiment will focus only on security requirements with assurance level "high". The natural language description of the metrics is also used as a second input for the metric recommender system.

Requirement input data example

The following is an example for a high-level security requirement description in natural language (ReqID=*OIS-02.3H*, category=*Organisation of Information Security*):

```
The CSP introduces and maintains a manually managed
inventory of conflicting roles and enforces the
segregation of duties during the assignment or
modification of roles as part of the role management
process. OIS-02.3H
```

Metric input data example

The following is an example for a metric description in natural language:

```
"This metric is used to assess if access monitoring is
enabled" M237 - MEDINA metric proposal
```

1.2. Experimental setup

First, the input texts are processed using a pre-trained network to produce a high dimensional feature vector. This feature vector represents the requirement or metric in a mathematically comparable way, instead of their original textual description. The basis for the recommender system is the hypothesis that requirements having similar descriptions are expected to be closer in the feature space. Since the same features are also computed for the metrics, this should also hold for the metrics. Which means, that, if the features are selected/fine-tuned to this purpose, the metrics, that should be associated with a requirement, are in the same local area in this high dimensional feature space.

Features

As described above, the input for the metric recommender system, i.e., the model that associates metrics with a requirement, is the computed feature vectors. The features can be computed using knowledge of the domain (e.g., for audio, spectrograms can be used). Alternatively, the features can be learned using machine learning. To train such a model, usually a lot of data is needed (in our case text), which is not available. Therefore, we use the output of pre-trained models that have worked well on state-of-the-art Natural Language Processing tasks.

For our use case, two types of features are considered applicable – the ones related to context-aware models (e.g. BERT [37] and derivatives [38]) and context-free word embeddings (e.g. word2vec [39] or fastText [40]). In the first experiments, fastText features (on

metric/requirement description text) have shown the most promising results so far. Feature selection is of uttermost importance for the final outcome and, since this task is ongoing, it will be probably refined in the future.

The quality of the selected features directly influences the final result; therefore, some data cleaning is needed. In our case, this is mostly based on removing stop words [41]. This means that we remove words that appear often, but do not add useful information to the text.

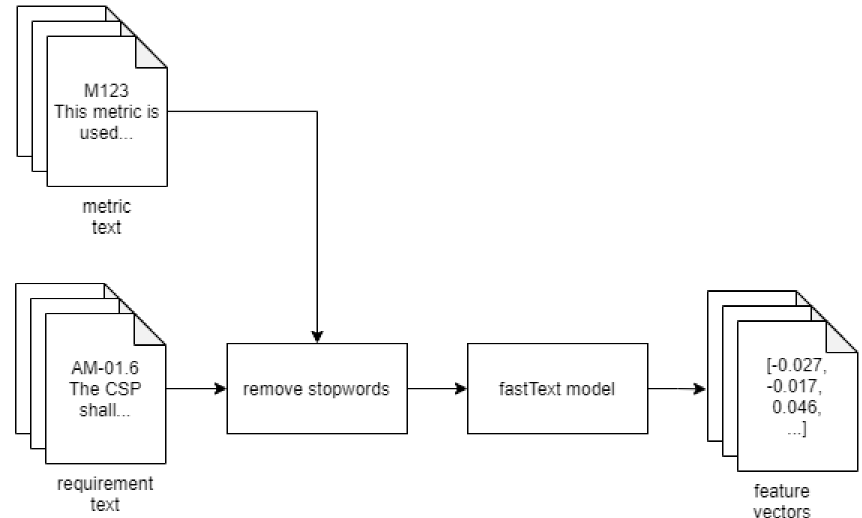


Figure 20. Feature computation workflow (source: MEDINA's own contribution)

Current approach

Features are computed using the fastText model *cc.en.300* which returns a 300-dimensional vector per requirement/metric (see Figure 20.). The fastText model is pre-trained on English texts from Wikipedia and Common Crawl [42]. For visual analysis, this high dimensional vector is reduced to two dimensions using the principal component analysis (PCA) [43] or the T-distributed Stochastic Neighbour Embedding (TSNE) [44] or the Truncated SVD [45].

A K-d tree is computed on the feature vectors of the metrics, which can be used to select the k closest neighbours of a query vector, based on the shortest Euclidean distance. As a query, we use the feature vector of a requirement. The workflow is depicted in Figure 21, and result examples are provided in the following sections.

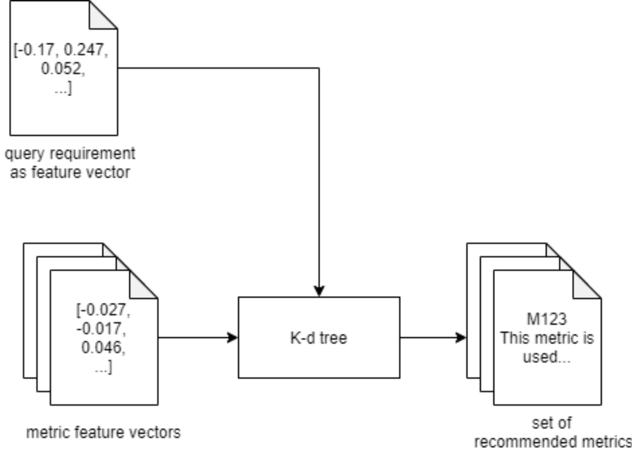


Figure 21. Recommender system workflow (source: MEDINA's own contribution)

CC BY-SA

Performance indicators

To analyse the performance of our system, basic indicators can be used. These indicators allow us to compare different approaches and help choosing the most promising one. Precision@K is typically the metric of choice for evaluating the performance of a recommender systems. However, additional diagnostic metrics and visualizations can be used, since they can offer deeper insights into a model's performance. First experiments results are based on the following indicators.

Precision@k

A quality metric to evaluate model's performance could be *precision@k* [46], e.g., *precision@5* reflects how well the system performs in the top 5 recommendation results. However, this score does not consider the rank of the relevant metrics only whether the relevant metrics are in the set of retrieved metrics. Equation 1shows the adjusted formula to calculate the precision@k score for this task. The numerator in Equation 1 is the amount of metrics in the intersection of *relevant metrics* (=metrics associated with a requirement) and *k* is the number of *retrieved metrics* (=recommended metrics for a requirement), the denominator is the amount of *relevant metrics*.

$$precision@k = \frac{|\{relevant\ metrics\} \cap \{k\ retrieved\ metrics\}|}{|\{relevant\ metrics\}|}$$ *Equation 1*

Normalised Discounted Cumulative Gain nDCG

An alternative quality metric to *precision@*k is the Discounted Cumulative Gain [47] (DCG), which is an indicator that can be used to measure the quality of our recommender system results, including the rank of relevant documents. The resulting metric list is ranked, and the metrics associated with the query requirement (=relevant documents) receive a relevance score $rel_i=1$, while metrics not relevant are set to $rel_i=0$. The position of the document/metric is denoted by *i*. The DCG is calculated as defined in Equation 2. To make the DCG metric comparable to other results it needs to be normalized – therefore an ideal DCG (Equation 3) is calculated, reflecting the ideal result – all associated metrics are in the top results of the recommender. The DCG is normalized to a score between 0 and 1 using the ratio of *DCG* to *IDCG* (Equation 4).

$$DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)}$$ *Equation 2*

$$IDCG_p = \sum_{i=1}^{|REL_p|} \frac{2^{rel_i} - 1}{log_2(i+1)}$$ *Equation 3*

$$nDCG_p = \frac{DCG_p}{IDCG_p}$$ *Equation 4*

1.3. First results

For the purpose of testing the performances of the metric recommender, and only for this scope, at time of writing we considered not only the 33 EUCS requirements selected in D2.1 [1], but also others, resulting in a total of 44. Such 44 requirements have been manually associated with

CC BY-SA

108 metrics in total. This means that these can be actively used for testing this recommender system prototype. So far, the quality of the approach was visually analysed using interactive plots and filtering of the results and measured using the above defined nDCG.

Visual analysis

Figure 22 depicts three plots using the first two components of the down-projected features. The reduction has been done using TSNE, PCA and Truncated SVD, from left to right. Each coloured dot represents either a requirement or a metric. Visual analysis indicates that the features are mostly good, as the distribution of different schemes and metrics are clustered on top of each other. This empirically supports the hypothesis the recommender system is built on.

Especially in the PCA plot, it is noticeable that some metrics are skewed. Using the interactive analysis tool built for examining the data, anomalies such as this one can be directly investigated.

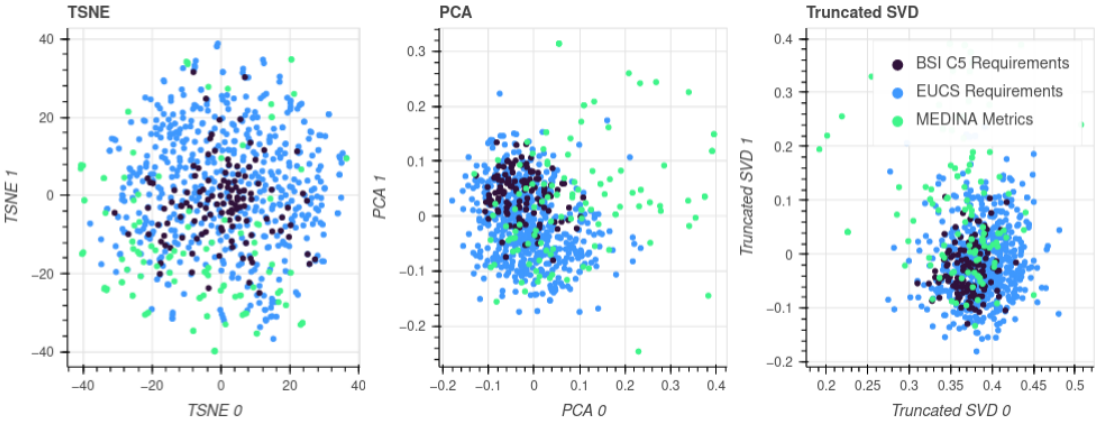


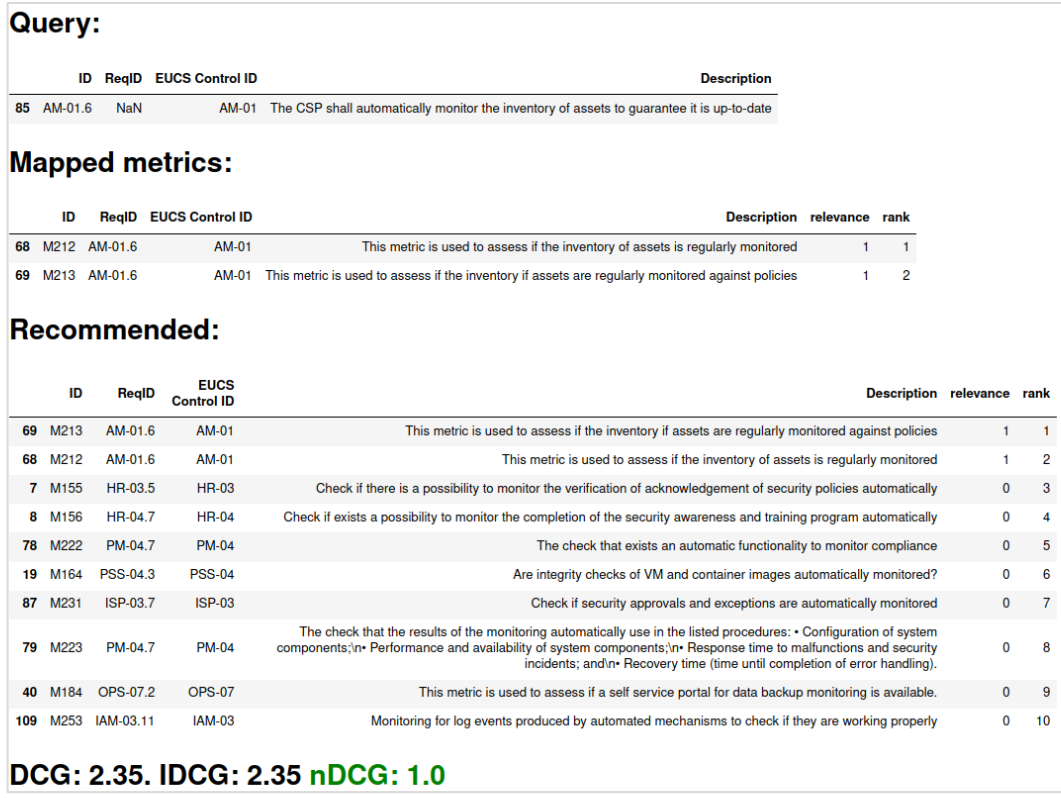
Figure 22. Plot of requirements and metrics using the first two components of the feature vectors, down-projected using TSNE, PCA and Truncated SVD respectively (source: MEDINA's own contribution)

Results

For 27 out of 44 requirements at least a subset of the linked metrics could be retrieved within the top 10 results (see Figure 23 and Figure 24). For the rest 17 requirements, no metrics could be retrieved (for an example see Figure 25). In total the mean of the nDCG is 0.41.

Disregarding the 17 requirements, for which no metrics could be retrieved yet, the "corrected" mean nDCG is 0.66. For 12 requirements 100% of the associated metrics were retrieved in the top results. An example of the latter can be seen in Figure 23.

(cc) BY-SA



Query:

	ID	ReqID	EUCS Control ID	Description
85	AM-01.6	NaN	AM-01	The CSP shall automatically monitor the inventory of assets to guarantee it is up-to-date

Mapped metrics:

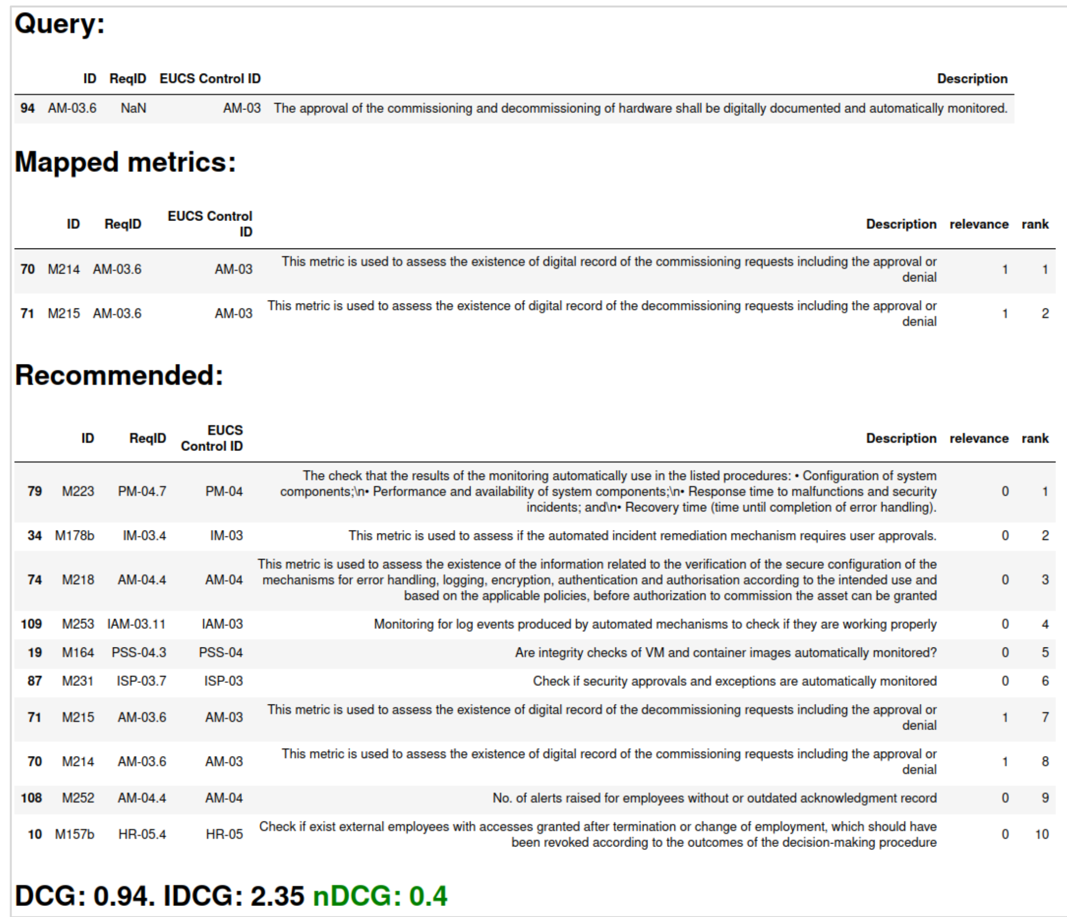
	ID	ReqID	EUCS Control ID	Description	relevance	rank
68	M212	AM-01.6	AM-01	This metric is used to assess if the inventory of assets is regularly monitored	1	1
69	M213	AM-01.6	AM-01	This metric is used to assess if the inventory if assets are regularly monitored against policies	1	2

Recommended:

	ID	ReqID	EUCS Control ID	Description	relevance	rank
69	M213	AM-01.6	AM-01	This metric is used to assess if the inventory if assets are regularly monitored against policies	1	1
68	M212	AM-01.6	AM-01	This metric is used to assess if the inventory of assets is regularly monitored	1	2
7	M155	HR-03.5	HR-03	Check if there is a possibility to monitor the verification of acknowledgement of security policies automatically	0	3
8	M156	HR-04.7	HR-04	Check if exists a possibility to monitor the completion of the security awareness and training program automatically	0	4
78	M222	PM-04.7	PM-04	The check that exists an automatic functionality to monitor compliance	0	5
19	M164	PSS-04.3	PSS-04	Are integrity checks of VM and container images automatically monitored?	0	6
87	M231	ISP-03.7	ISP-03	Check if security approvals and exceptions are automatically monitored	0	7
79	M223	PM-04.7	PM-04	The check that the results of the monitoring automatically use in the listed procedures: • Configuration of system components;\n• Performance and availability of system components;\n• Response time to malfunctions and security incidents; and\n• Recovery time (time until completion of error handling).	0	8
40	M184	OPS-07.2	OPS-07	This metric is used to assess if a self service portal for data backup monitoring is available.	0	9
109	M253	IAM-03.11	IAM-03	Monitoring for log events produced by automated mechanisms to check if they are working properly	0	10

DCG: 2.35. IDCG: 2.35 nDCG: 1.0

Figure 23. Prototypical results for EUCS requirement AM-01.6, optimal results on rank 1 and 2



Query:

	ID	ReqID	EUCS Control ID	Description
94	AM-03.6	NaN	AM-03	The approval of the commissioning and decommissioning of hardware shall be digitally documented and automatically monitored.

Mapped metrics:

	ID	ReqID	EUCS Control ID	Description	relevance	rank
70	M214	AM-03.6	AM-03	This metric is used to assess the existence of digital record of the commissioning requests including the approval or denial	1	1
71	M215	AM-03.6	AM-03	This metric is used to assess the existence of digital record of the decommissioning requests including the approval or denial	1	2

Recommended:

	ID	ReqID	EUCS Control ID	Description	relevance	rank
79	M223	PM-04.7	PM-04	The check that the results of the monitoring automatically use in the listed procedures: • Configuration of system components;\n• Performance and availability of system components;\n• Response time to malfunctions and security incidents; and\n• Recovery time (time until completion of error handling).	0	1
34	M178b	IM-03.4	IM-03	This metric is used to assess if the automated incident remediation mechanism requires user approvals.	0	2
74	M218	AM-04.4	AM-04	This metric is used to assess the existence of the information related to the verification of the secure configuration of the mechanisms for error handling, logging, encryption, authentication and authorisation according to the intended use and based on the applicable policies, before authorization to commission the asset can be granted	0	3
109	M253	IAM-03.11	IAM-03	Monitoring for log events produced by automated mechanisms to check if they are working properly	0	4
19	M164	PSS-04.3	PSS-04	Are integrity checks of VM and container images automatically monitored?	0	5
87	M231	ISP-03.7	ISP-03	Check if security approvals and exceptions are automatically monitored	0	6
71	M215	AM-03.6	AM-03	This metric is used to assess the existence of digital record of the decommissioning requests including the approval or denial	1	7
70	M214	AM-03.6	AM-03	This metric is used to assess the existence of digital record of the commissioning requests including the approval or denial	1	8
108	M252	AM-04.4	AM-04	No. of alerts raised for employees without or outdated acknowledgment record	0	9
10	M157b	HR-05.4	HR-05	Check if exist external employees with accesses granted after termination or change of employment, which should have been revoked according to the outcomes of the decision-making procedure	0	10

DCG: 0.94. IDCG: 2.35 nDCG: 0.4

Figure 24. Prototypical results for EUCS requirement AM-03.6, results on rank 7 and 8

CC BY-SA

Query:

	ID	ReqID	EUCS Control ID	Description
441	IM-03.4	NaN	IM-03	The CSP shall allow customers to actively approve the solution before automatically approving it after a certain period

Mapped metrics:

	ID	ReqID	EUCS Control ID	Description	relevance	rank
33	M178a	IM-03.4	IM-03	This metric is used to assess if automated incident management (detection, response) and SIEM has been enabled on a cloud service / asset	1	1
34	M178b	IM-03.4	IM-03	This metric is used to assess if the automated incident remediation mechanism requires user approvals.	1	2
89	M233	IM-03.4	IM-03	(BSI-C5 / Sim-04) Check if customers have the ability to review security incident solutions.	1	3
90	M234	IM-03.4	IM-03	(BSI-C5 / Sim-04) Check if security incident solutions are up to date.	1	4

Recommended:

	ID	ReqID	EUCS Control ID	Description	relevance	rank
13	M158	HR-05.4	HR-05	Check if access rights are revoked on contract termination or change according to the decision making procedure automatically	0	1
74	M218	AM-04.4	AM-04	This metric is used to assess the existence of the information related to the verification of the secure configuration of the mechanisms for error handling, logging, encryption, authentication and authorisation according to the intended use and based on the applicable policies, before authorization to commission the asset can be granted	0	2
10	M157b	HR-05.4	HR-05	Check if exist external employees with accesses granted after termination or change of employment, which should have been revoked according to the outcomes of the decision-making procedure	0	3
9	M157a	HR-05.4	HR-05	Check if exist internal employees with accesses granted after termination or change of employment, which should have been revoked according to the outcomes of the decision-making procedure	0	4
7	M155	HR-03.5	HR-03	Check if there is a possibility to monitor the verification of acknowledgement of security policies automatically	0	5
79	M223	PM-04.7	PM-04	The check that the results of the monitoring automatically use in the listed procedures: • Configuration of system components;\n• Performance and availability of system components;\n• Response time to malfunctions and security incidents; and\n• Recovery time (time until completion of error handling).	0	6
8	M156	HR-04.7	HR-04	Check if exists a possibility to monitor the completion of the security awareness and training program automatically	0	7
2	M92	NaN	-	percentage of relevant employees who have received training on the privacy program and policies in place.	0	8
4	M105	NaN	-	the percentage of employees who have certified their acceptance of responsibilities for activities that involve handling of private data.	0	9
87	M231	ISP-03.7	ISP-03	Check if security approvals and exceptions are automatically monitored	0	10

DCG: 0.0. IDCG: 3.7 nDCG: 0.0

Figure 25. Prototypical results for EUCS requirement IM-03.4, no results

2. CNL translations

Once the list of metrics associated to a requirement is obtained, it is possible to translate each pair requirement/metric in CNL. Here, we give some examples of translation from NL to CNL.

Let the reader consider the following requirement:

If we consider EUCS requirement ReqID = OPS-21.3, category Operational Security [9]

```
The CSP shall automatically monitor the service components
under its responsibility for compliance with hardening
specifications.
```

The metric recommender associates this requirement with the metric metricID=JavaVersion, whose description is: “This metric is used to assess the Java Runtime version used by the cloud service/asset”. The metric data type is Integer in the range of [< 11; 11]. The target value is 11, and the operator that compares the target value with the measured value is “=”. This metric can thus be expressed as the following obligation:

```
"Application" MUST "JavaVersion", Integer(=,11)
```

For the sake of clarity, we give another translation example:

BY-SA

If we consider EUCS requirement ReqID = PM-04.8, category Procurement Management [9]:

```
The CSP shall automatically monitor Identified violations
and discrepancies, and these shall be automatically
reported to the responsible personnel or system components
of the Cloud Service Provider for prompt assessment and
action.
```

The metric recommender associates this requirement with the metric metricID=AutomaticallyDetectedViolationsDiscrepancies, whose description is: "The percentage of violations and discrepancies which can be automatically detected". The metric data type is Integer, ranged over [0;100]. The target value is 100, and the operator that compares the target value with the measured value is "=". Then, this metric can be expressed as the following obligation:

```
"ComplianceMonitoringSoftware" MUST
"AutomaticallyDetectedViolationsDiscrepancies", Integer(=,100)
```

Finally, we report the example of a requirement associated with two different metrics, i.e., the requirement ReqID = OPS-5.3, category Operational Security:

```
The CSP shall automatically monitor the systems covered
by the malware protection and the configuration of the
corresponding mechanisms to guarantee fulfilment of OPS-
05.1.
```

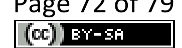
The metric recommender associates this requirement with metrics MetricID=MalwareProtectionEnabled and MetricID=MalwareProtectionOutput, respectively. The description of the metric MalwareProtectionEnabled is "This metric is used to assess if the antimalware solution is enabled on the respective resource", its data type is Boolean and the values it can assume are [true, false]. The default target value is "true" and the operator that compares the target value with the measured value is "=". Then, this metric is translated into the following obligation:

```
"Compute.VirtualMachine" MUST "MalwareProtectionEnabled",
Boolean(=,true)
```

Similarly, the description of the metric MalwareProtectionOutput is "These metric states whether automatic notifications are enabled (e.g., e-mail) about malware threats. This relates to EUCS definition of continuous monitoring". The metric data type is Boolean and the values it can assume are [true, false]. The default target value is "true" and the operator that compares the target value with the measured value is "=". Then, this metric is translated into the following obligation:

```
"Compute.VirtualMachine" MUST "MalwareProtectionOutput",
Boolean(=,true)
```

It is worth highlighting that the example obligations shown so far are reported in the CNL format, whereas in the actual implementation they will be rendered in XML, according to the language expected by the CNL Editor.



APPENDIX D: MEDINA Vocabularies and Ontologies

This section presents the MEDINA Ontologies that are at the base of the formation of CNL obligations in the CNL Editor Tool, and of the rules in Rego Code, the format chosen as the MEDINA Certification Language. Before introducing the ontologies, we will give a brief background on the concept of taxonomy and ontology.

We invite the reader to notice that the ontologies have been already presented in the previous deliverable D2.3 [3], Section 5.3, and we have included this section in the present document for the sake of completeness.

1. Background: Taxonomies and Ontologies

Taxonomies are classifications of arbitrary objects, usually in a hierarchical order. Their purpose is to create abstract classes to which any concrete instance can be assigned. This way, general methods and tools can be developed that apply to all instances of a certain class. Taxonomies exist in many different domains, for example in biology to classify animals and plants or in computer science to classify security flaws or cloud resources (as proposed in MEDINA).

Sometimes, taxonomies are not sufficient to describe classes and their relationships. Ontologies can integrate several taxonomies and describe their relations in arbitrary ways. In MEDINA, for instance, we use an ontology to subsume two taxonomies which describe cloud resources and security features, and add relationships between the two taxonomies to describe which cloud resource generally has which security feature.

2. Editor Ontology

The CNL Editor functionality of managing requirements is based on a specific vocabulary, saved in a .owl file (RDF format). CNL Editor allows users, for a selected requirement, to change the Target Value choosing from a predefined list saved in the vocabulary. The use of free text is also allowed in special situations (e.g., where there is a need of a value that cannot be enumerated in the ontology, like an integer number). This way, the user is interactively bound to make changes as defined in the vocabulary.

The CNL Editor vocabulary consists of entities (items) that are defined under *Action* and *Term* classes. Additional actions and terms must be defined as subclasses of these existing classes in any vocabulary that will be created.

In the CNL Editor, the *Term* (and its subclasses) is used to instantiate resource types and target values, while the *Action* (and its subclasses) is used to instantiate metrics. Figure 26 shows an example of the vocabulary, with *Action* and *Term* examples highlighted.

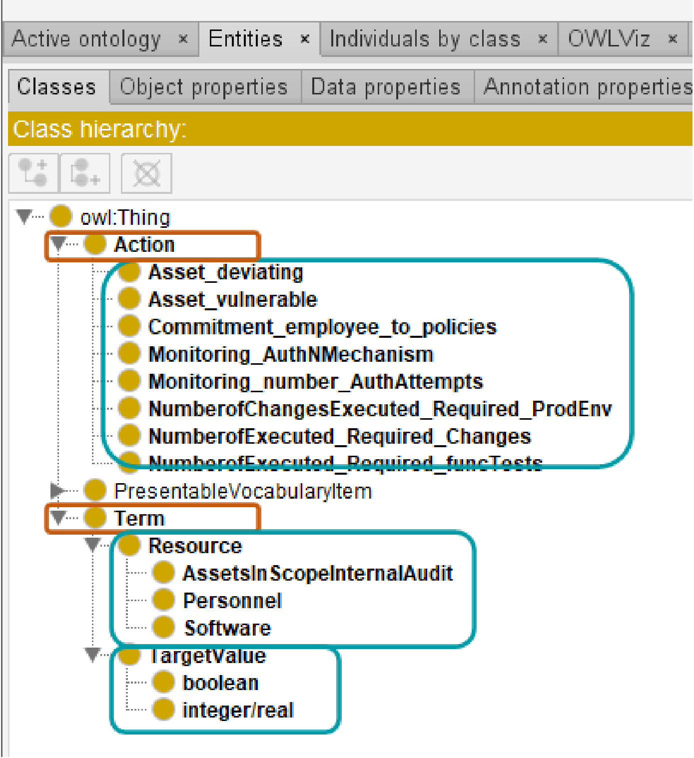


Figure 26. CNL Editor Vocabulary Structure Example

In the CNL Editor Vocabulary for MEDINA project, items come from the Catalogue of controls and security metrics. As anticipated, *Actions* are the Metrics Names and under the *Term* class we can find: i) Resource that contains ResourceTypes, and ii) TargetValue with the TargetValueTypes and values admitted.

For each Metric in the vocabulary, the type of resource and the type of target value associated are specified. The association can be retrieved from the Catalogue of controls and security metrics.

When editing an obligation, the user is allowed to choose only values defined in the vocabulary.

As an example, Figure 27 shows the association of ResourceType and TargetValueType to the Metric named BackupEncryptionEnabled.

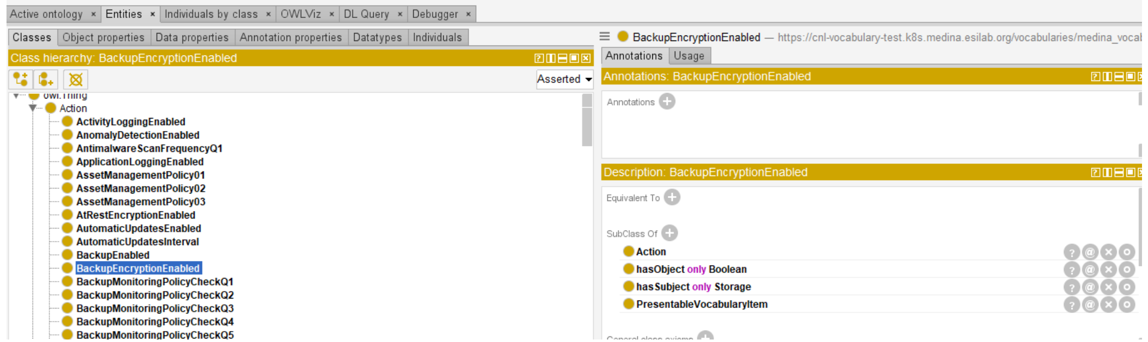


Figure 27. CNL Editor Vocabulary Metric "BackupEncriptionEnabled"

Figure 28 shows that the TargetValueType is "Boolean".

CC BY-SA

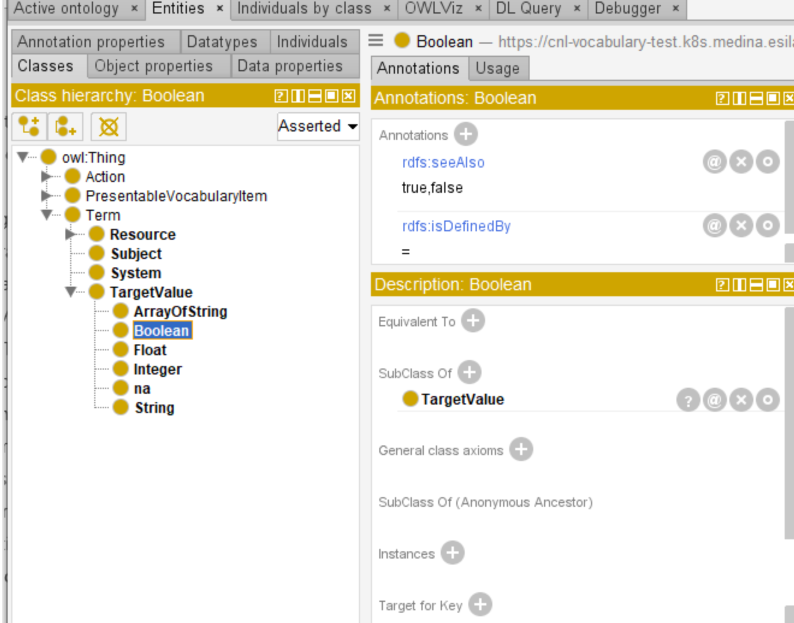


Figure 28. CNL Editor Vocabulary TargetValueType "Boolean"

For TargetValueType "Boolean", the vocabulary defines possible values (true or false) and possible operators that in this case is only "=".

The vocabulary must be defined considering as input what is specified in the Catalogue of controls and security metrics and must be aligned to it.

The MEDINA vocabulary is created and modified via the Protégé[25] tool, a free, open-source ontology editor, with an easy-to-use GUI.

3. Cloud Resource Security Ontology

The Cloud Resource Security Ontology (CRSO) is not directly related to the CNL Editor ontology described above, since these two ontologies serve two different purposes. They do, however, both include a classification of cloud resources which will be aligned in a future iteration.

The Cloud Resource Security Ontology has been developed to harmonise evidence gathering and assessment across certifications, cloud vendors, and resource types. It includes several taxonomies, including a taxonomy of cloud resources and a taxonomy of security properties. As an example, the cloud resource taxonomy includes computing resources which in turn can be virtual machines, containers, or functions.

Figure 29 shows the current status of the cloud resource taxonomy. This taxonomy classifies cloud resources across all major cloud providers and architectures, like Microsoft Azure, Amazon Web Services, Google Cloud Platform, and OpenStack. It is ordered by cloud service categories which is the typical classification the major cloud providers also use. For instance, resources that can be used to execute code (virtual machines, serverless functions, etc.) are grouped in a *Compute* category, while resources that provide networking capabilities (virtual networks and subnetworks, IP addresses, routing rules, etc.) pertain to a *Networking* group. The higher-order classes in this taxonomy are the following:

- Compute

[25] https://protege.stanford.edu/

- Identity Management
- Container Orchestration
- Container Registry
- Continuous Integration/Continuous Delivery Service
- Logging
- Networking
- IoT
- Storage
- Account
- Image

The security properties taxonomy classifies security properties that can configured in a cloud service. We have ordered it by STRIDE categories. STRIDE is an acronym for the security threats Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, and Elevation of Privilege. The security properties taxonomy includes their respective counterparts, i.e., the following protection goals:

- Authentication
- Integrity
- Non-repudiation
- Confidentiality
- Availability
- Authorization

The Confidentiality category, for example, includes *encryption* as a security property, while the Integrity category includes the *immutability* property which may be implemented by some storage resources. A further example is presented in Figure 30 and the complete security properties taxonomy is shown in Figure 31.

In particular, Figure 30 shows an excerpt from the CRSO: A BlockStorage is a sub-type of Storage which in turn is a Service. Security properties that are offered on one level of this hierarchy are inherited by child nodes. For instance, Storage offers AtRestEncryption which is inherited by the BlockStorage. Furthermore, BlockStorage offers two specific types of AtRestEncryption, i.e., CustomerManagedKeyEncryption and CPManagedKeyEncryption.

Without harmonizing evidence gathering across cloud vendors with the help of Cloud Resource Security Ontology and related taxonomies, processing evidence would be much more tedious, since, e.g., two resources hosted by different cloud providers but with similar security properties have to be parsed by dedicated pieces of code, and measured by dedicated metrics.

CC BY-SA

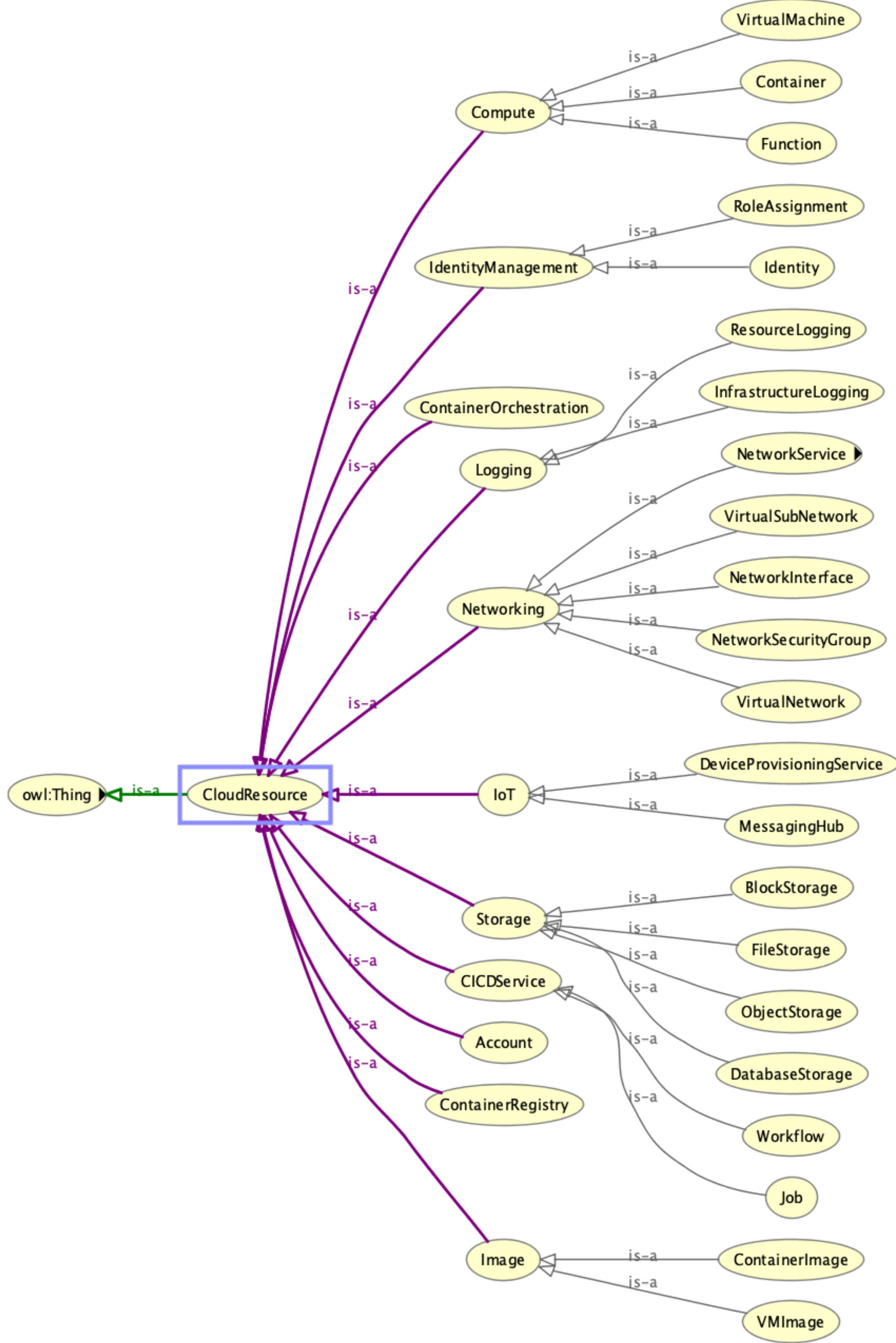


Figure 29. The cloud resource taxonomy which classifies cloud resources according to their functional purpose, like compute, storage, and networking

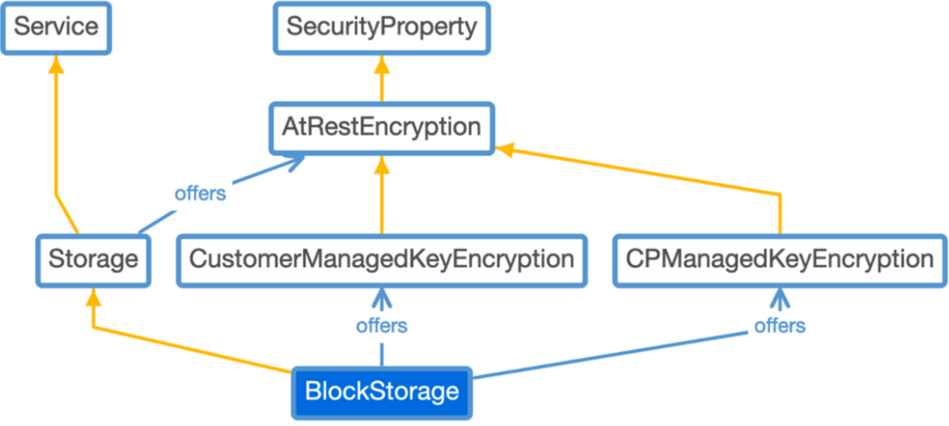


Figure 30. An excerpt from the CRSO

A further advantage of this harmonization is that requirements defined in different certifications or catalogues can be assigned to the ontological concepts they refer to. This assignment to a common ontological classification allows to assess evidence collectively and improves the assessment's reusability and modifiability. The ontological types can therefore be used in metric definitions related to the certifications and catalogues. This way, metrics are defined for abstract resource types, e.g., object storages, and their security properties, e.g., at-rest-encryption, rather than for cloud-provider-specific properties, e.g., an encryption property that is specifically defined for an AWS S3 bucket.

The ontology has been created using Protegé and is published in the online Protegé platform webprotege[26]. Please note that it has been described and used in the publication *Cloud Property Graph: Connecting Cloud Security Assessments with Static Code Analysis* (IEEE CLOUD 2021) as well, which has been funded by MEDINA [48]. In this publication, the ontology is used as a basis for a combined security analysis of cloud infrastructures and deployed software. Such an analysis similarly requires a harmonisation of security concepts, e.g., encryption or logging functionalities need to be recognised on the infrastructure level as well as on the source code level to allow for a comprehensive assessment of security requirements.

The ontology will be extended and may be further exploited, for instance it can be used to cover the resource and security properties specified in different certifications and control catalogues.

[26] https://webprotege.stanford.edu/#projects/8e5d391e-6436-41c0-b9a9-9226e90a38a9/edit/Classes

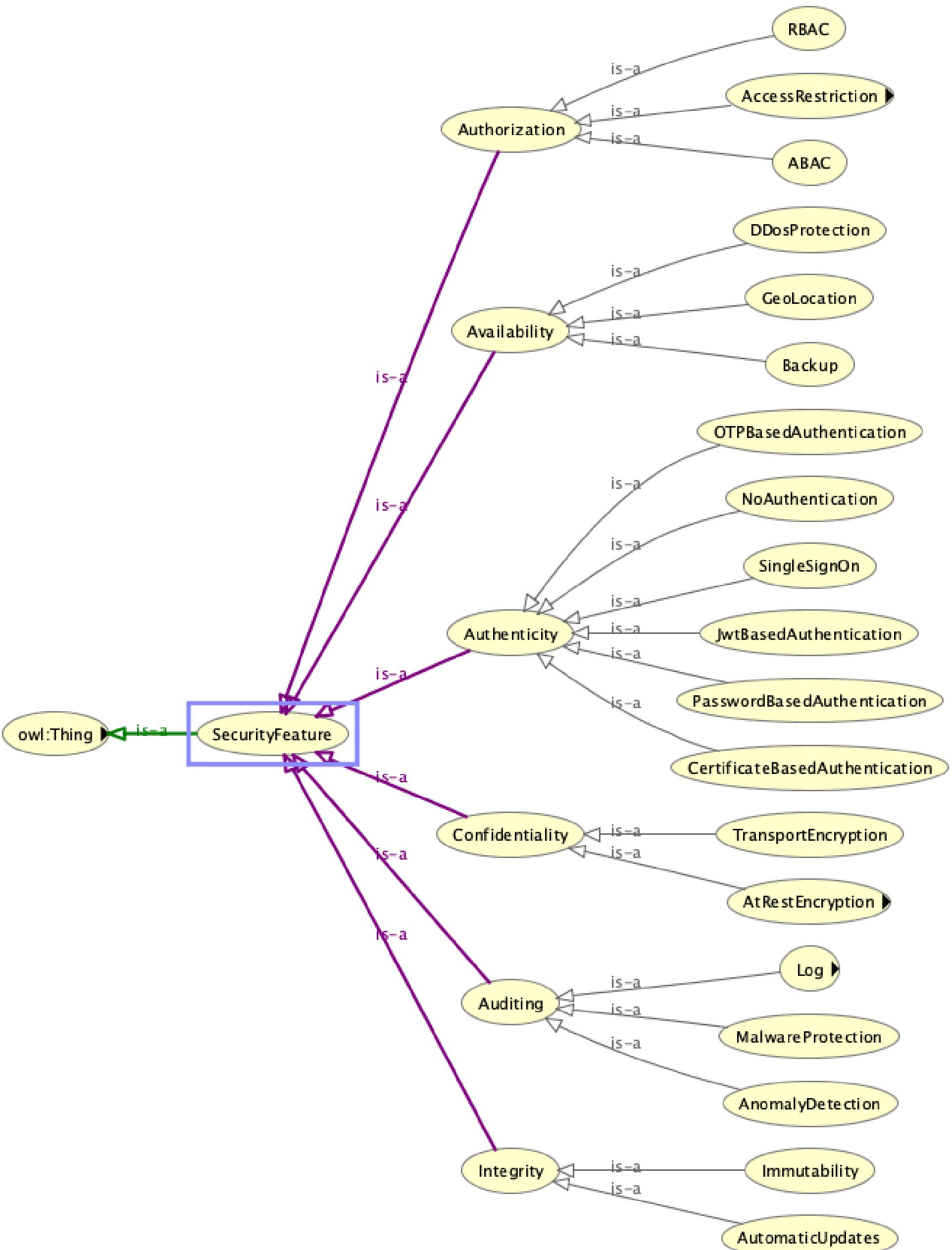


Figure 31. The security property taxonomy which classifies security properties according to their targeted STRIDE-based goal

(cc) BY-SA