



Deliverable D1.1

Data Modelling and interaction mechanisms – v1

Editor(s):	Franz Deimling
Responsible Partner:	Fabasoft R&D GmbH
Status-Version:	Final - v1.0
Date:	31.07.2024
Type:	R
Distribution level (SEN, PU):	PU

Project Number:	101120688
Project Title:	EMERALD

Title of Deliverable:	D1.1 - Data Modelling and interaction mechanisms – v1
Due Date of Delivery to the EC	31.07.2024

Workpackage responsible for the Deliverable:	WP1 - Concept and methodology of EMERALD
Editor(s):	Franz Deimling (FABA)
Contributor(s):	CNR, FABA, FhG, SCCH, TECNALIA
Reviewer(s):	Iñaki Etxaniz, Gorka Benguria Elguezabal, Cristina Martínez (TECNALIA)
Approved by:	All Partners
Recommended/mandatory readers:	WP1, WP2, WP3, WP4, WP5

Abstract:	Initial version of the overview of data models and techniques used for creating and linking the data to evidence (annotation, etc)
Keyword List:	Data diagram, data model, component overview
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA 4.0 DEED https://creativecommons.org/licenses/by-sa/4.0/)
Disclaimer:	Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. The European Union cannot be held responsible for them.

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	27.03.2024	First draft version	FABA
v0.2	27.06.2024	Comments and suggestions received by consortium partners	WP1 partners
v0.3	30.06.2024	Contributions from component partners added	WP2 and WP3 partners
v0.4	09.07.2024	Figures and listings updated	FABA, Tecnalia, FhG
v0.5	19.07.2024	QA Review	TECNALIA
v0.6	24.07.2024	Addressed all comments received in the Internal QA review	FABA
v1.0	31.07.2024	Submitted to the European Commission	TECNALIA

Table of contents

Terms and abbreviations.....	6
Executive Summary.....	7
1 Introduction.....	8
1.1 About this deliverable.....	8
1.2 Document structure.....	8
2 Data Model Overview.....	9
3 Component Data Models	11
3.1 Evidence Collector Data Models.....	13
3.1.1 AI-SEC.....	13
3.1.2 AMOE.....	14
3.1.3 Cloudfitor-Discovery	15
3.1.4 Codyze	17
3.1.5 eknows.....	18
3.2 Trustworthiness System (TWS) Data Model.....	20
3.3 Mapping Assistant for Regulations with Intelligence (MARI) Data Model	21
3.4 Repository of Controls and Metrics (RCM) Data Model	22
3.5 Orchestrator Data Model.....	25
3.6 Evidence Store Data Model	27
3.7 Assessment Data Model	27
3.8 Evaluation Data Model	28
4 Interactive Documentation	30
4.1 PlantUML	30
4.2 Web Service	30
4.2.1 Implementation details	31
4.3 Data model versioning.....	31
5 Data Exchange and Formats.....	33
5.1 Interaction mechanisms between components.....	33
5.2 Sequence diagrams.....	35
6 Conclusions.....	37
7 References.....	38
APPENDIX: Release 1.0.9 of Architecture and Data Modelling.....	39

List of figures

FIGURE 1. EMERALD DATA DIAGRAM	10
FIGURE 2. OVERVIEW OF THE EMERALD COMPONENTS.....	12
FIGURE 3. OVERVIEW OF THE AI-SEC COMPONENT DATA MODEL	13
FIGURE 4. OVERVIEW OF THE AMOE COMPONENT DATA MODEL	15
FIGURE 5. OVERVIEW OF THE CLOUDITOR-DISCOVERY COMPONENT DATA MODEL.....	16
FIGURE 6. CODYZE COMPONENT OVERVIEW	18
FIGURE 7. OVERVIEW OF THE EKNOWS COMPONENT DATA MODEL.....	19
FIGURE 8. OVERVIEW OF THE TRUSTWORTHINESS SYSTEM COMPONENT DATA MODEL	20
FIGURE 9. OVERVIEW OF THE MARI COMPONENT DATA MODEL.....	22
FIGURE 10. OVERVIEW OF THE RCM COMPONENT DATA MODEL.....	24
FIGURE 11. OVERVIEW OF THE ORCHESTRATOR COMPONENT DATA MODEL	26
FIGURE 12. OVERVIEW OF THE EVIDENCE STORE COMPONENT DATA MODEL	27
FIGURE 13. OVERVIEW OF THE ASSESSMENT COMPONENT DATA MODEL.....	28
FIGURE 14. OVERVIEW OF THE EVALUATION COMPONENT DATA MODEL.....	29
FIGURE 15. INTERACTIVE SVG - HIGHLIGHT NEIGHBOURS ON CLICK	30
FIGURE 16. LANDING PAGE OF THE INTERACTIVE DOCUMENTATION.....	31

List of listings

LISTING 1. EXAMPLE OF VIRTUAL MACHINE PROPERTIES.....	17
LISTING 2. AMOE EVIDENCE IN JSON.....	33
LISTING 3. CLOUDITOR EXAMPLE EVIDENCE IN JSON	34
LISTING 4. A EUCS REQUIREMENT MAPPING IN OSCAL	35

Terms and abbreviations

AIC4	Artificial Intelligence Cloud Service Compliance Criteria Catalogue
AMOE	Assessment and Management of Organisational Evidence
API	Application Programming Interface
AST	Abstract Syntax Tree
BSI	Bundesamt für Sicherheit in der Informationstechnik
CI/CD	Continuous Integration / Continuous Delivery
CLI	Command Line Interface
CSP	Cloud Service Provider
DoA	Description of Action
EC	European Commission
EUCS	European Cybersecurity Certification Scheme for Cloud Services
GA	Grant Agreement to the project
GASTM	Generic Abstract Syntax Tree
gRPC	gRPC Remote Procedure Call (created by Google)
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MARI	Mapping Assistant for Regulations with Intelligence
NLP	Natural Language Processing
OSCAL	Open Security Controls Assessment Language
PDF	Portable Document Format
PNG	Portable Network Graphics
RCM	Repository of Controls and Metrics
REST	Representational State Transfer
SARIF	Static Analysis Results Interchange Format
SVG	Scalable Vector Graphics
TRL	Technology Readiness Level
TWS	Trustworthiness System
UML	Unified Modelling Language
UUID	Universally Unique Identifier
WP	Work Package

Executive Summary

This deliverable, the first version of the data modelling and interaction mechanisms, provides an initial report on the data diagrams, design and documentation of the EMERALD framework and its components. The goal of the corresponding task T1.1 in work package 1 is to coordinate the different types of data shared between the components of WP2, WP3 and WP4. The deliverable provides an overview of the data model, as well as the setup of the interactive documentation. Furthermore, the data exchange and formats are described.

D1.1 lays the foundation of the data model – the underlying work of Task 1.1. The resulting documentation serves as a common ground to develop the different components and their APIs. It should offer a high-level overview of the components – displaying the flow of the data. Technical details can be found in the overall data diagram and data format descriptions. Additionally, an overview per component is provided, so as not to be overwhelmed by details, and to be able to focus only on parts of the EMERALD framework.

The document is structured in four main parts – the data model, the component overview, the interactive documentation and finally the data exchange and format description. It starts by giving detailed insights into the data classes used in EMERALD. The following section summarizes each component, starting with the evidence collectors (WP2) and continues with the different components of WP3. In the interactive documentation section, the technical setup of the documentation is described. Finally, the plans for the interaction mechanisms are outlined.

There will be a second version of this deliverable (D1.2 in M18), which will include updates to the data model and interaction mechanisms. The next steps will be for the different components to implement the data classes and APIs which will be described in the respective component deliverables. Depending on the requirements coming from the pilots (WP5), workflows (WP4) and technical work packages (WP2 and WP3), updates to the data diagrams will be included in the release cycle of the web service and included in the future version of this deliverable.

1 Introduction

This section explains the goal and purpose of the deliverable, its context and its structure.

1.1 About this deliverable

This deliverable is the first release of the task T1.1 “Data modelling and information sharing mechanisms” of WP1 of the EMERALD project [1]. It shall provide an overview of the data model that is used in the EMERALD framework. Furthermore, the deliverable provides an overview of each component’s data and how it is linked to other components. The goal is to provide insights of the current state of the data used in EMERALD and how it is organized. A second version of this deliverable will be D1.2, which is due to in M18.

The data model will be used by all the components in collaboration with WP2 and WP3 as well as the **EmeraldUI** component that will be developed in WP4. The interaction mechanism between the different software components will be described and preferred data formats will be presented to facilitate data access and sharing.

The task uses the existing data classes of the components and focuses on providing relevant information to the different partners, unfamiliar to the different components. Different abstraction layers will be used to provide an overview and detailed insights. The diagrams will be adjusted over the course of the project and adopted to the requirements of the different components. In order not to lose track of any changes, dedicated processes (see Section 4.3) have been set up to check this.

1.2 Document structure

The document is organized into four main sections:

- Data model
- Component overview
- Interactive documentation
- Data exchange and formats

The data model overview section, Section 2, depicts and describes the current state of the whole data model used in EMERALD. It shall give detailed insights into the inter-component relationships of the EMERALD data.

In order to have a more abstract view and not get lost in the details, an overview of the components is provided in Section 3. This section contains a subsection dedicated to each EMERALD component.

Section 4 describes the deployment and core implementation of the interactive documentation approach used to share the data model within the EMERALD project. There are three subsections, starting with a section describing PlantUML and how it is used to create the diagrams. This is followed by a description of the web service. Finally, the process on versioning and updating the diagrams is described.

Section 5 describes the different formats used in the project and how the components communicate. The deliverable is summarized in Section 6.

Finally, the current release of the interactive documentation can be found in the *APPENDIX: Release 1.0.9 of Architecture and Data Modelling*.

2 Data Model Overview

This section describes the current version of the EMERALD data model. The model describes the different data classes as well as their connections within and between components. The goal is to provide insights to developers and users of the EMERALD framework. Therefore, the data diagram is presented in an interactive system¹ that is explained in more detail in Section 4. There are different abstraction layers, to allow for a “drill down” on the details.

Figure 1 shows the resulting data model for the whole EMERALD framework². It depicts each component in a separate box, whereas the background colour denotes the EMERALD work package to which it is related. Evidence collection components (WP2) are coloured in orange, and WP3 components are coloured in teal. Each component box contains the data classes that are relevant for other developers and inter-component communication. Component specific information can be found in the respective subsection of Section 3.

This version of the data model is loosely based on the data model that was created in a similar predecessor project called MEDINA³ - the MEDINA data model was reported in deliverable D5.2⁴. The EMERALD project uses some of the components that were part of the MEDINA data model – such the **Evidence Store**, the **Orchestrator**, the **Repository of Controls and Metrics (RCM)** and the **Trustworthiness System**. Data classes related to components not relevant to EMERALD were excluded.

Please note that the **Questionnaire** is a subcomponent of the **RCM** and is therefore shown with a dedicated box in Figure 1. However, as the Questionnaire data model is quite large, it is not shown in Figure 1 but in the **RCM** component overview (see Figure 10).

¹ <https://models.emerald.digital.tecnalia.dev/>

² Please note that an enlarged view of the EMERALD data model is available in *APPENDIX: Release 1.0.9 of Architecture and Data Modelling*.

³ <https://medina-project.eu/>

⁴ https://medina-project.eu/wp-content/uploads/2023/05/MEDINA_D5.2_MEDINA_RequirementsDetailed_architectureDevOps_infrastructure_v2_v1.0.pdf

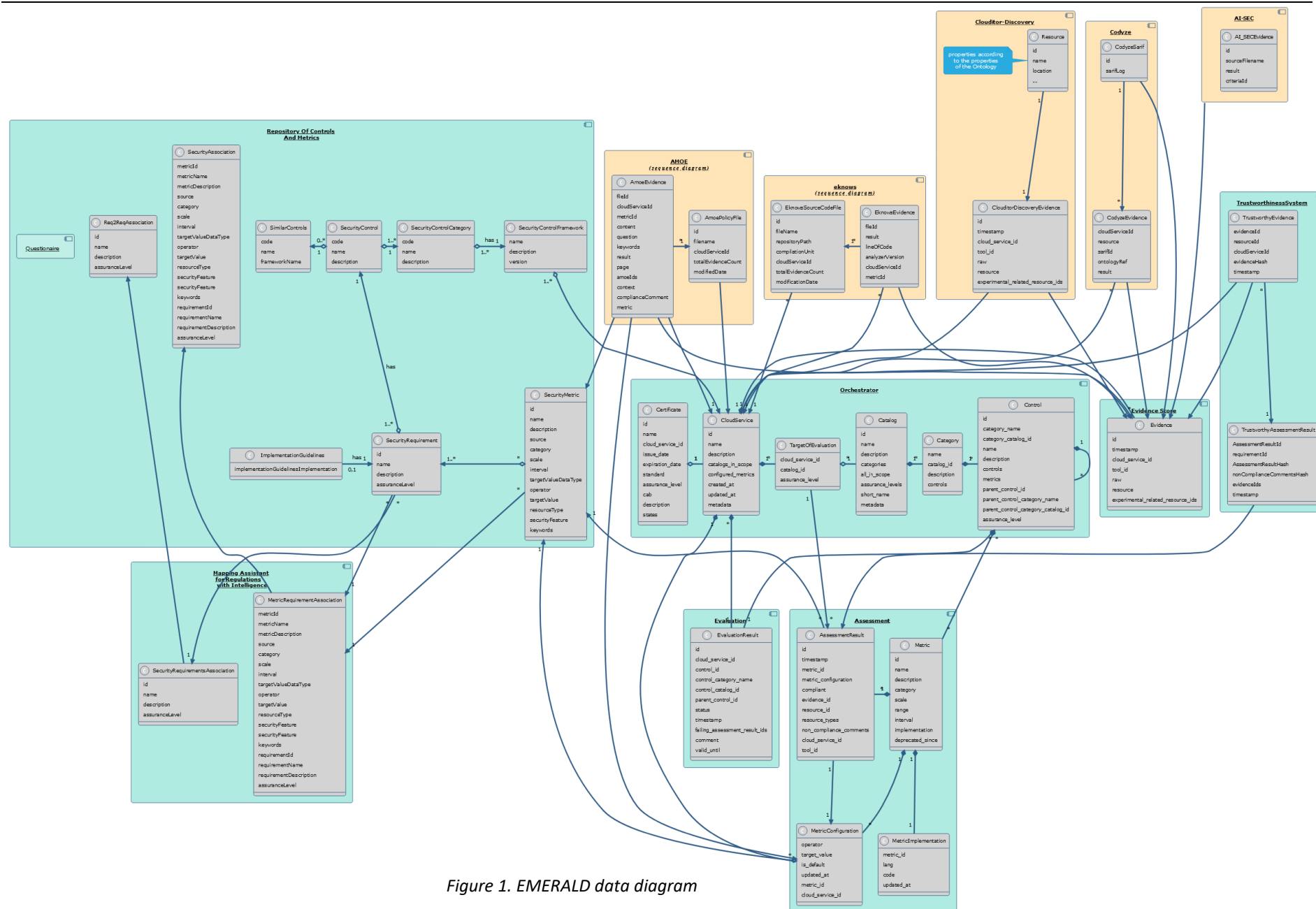


Figure 1. EMERALD data diagram

3 Component Data Models

This section describes each EMERALD component from a data-oriented point of view. It covers the different evidence extraction tools, where the evidence is stored and assessed, and the tools that provide and assist with the management of the security schemes and metrics. The different views have been integrated in the interactive documentation (see *APPENDIX: Release 1.0.9 of Architecture and Data Modelling* and can be reached via links.

Figure 2 depicts an abstracted view of the main EMERALD components and serves as a starting point for users as well as developers. The diagram shows the general data flow between all the components. The direction of the arrows indicates the direction the data flows. As also explained in the legend, a dashed line indicates that a component at the end of the arrow pulls data from the component at the other end, while a full line indicates that a component actively pushes data to another component using its API. The components are coloured according to the respective work package they are related to. The colour – work package associations can be found in the legend (see Figure 2).

Please note that the legend has been omitted in the figures of the component overviews to save space, as the information is redundant.

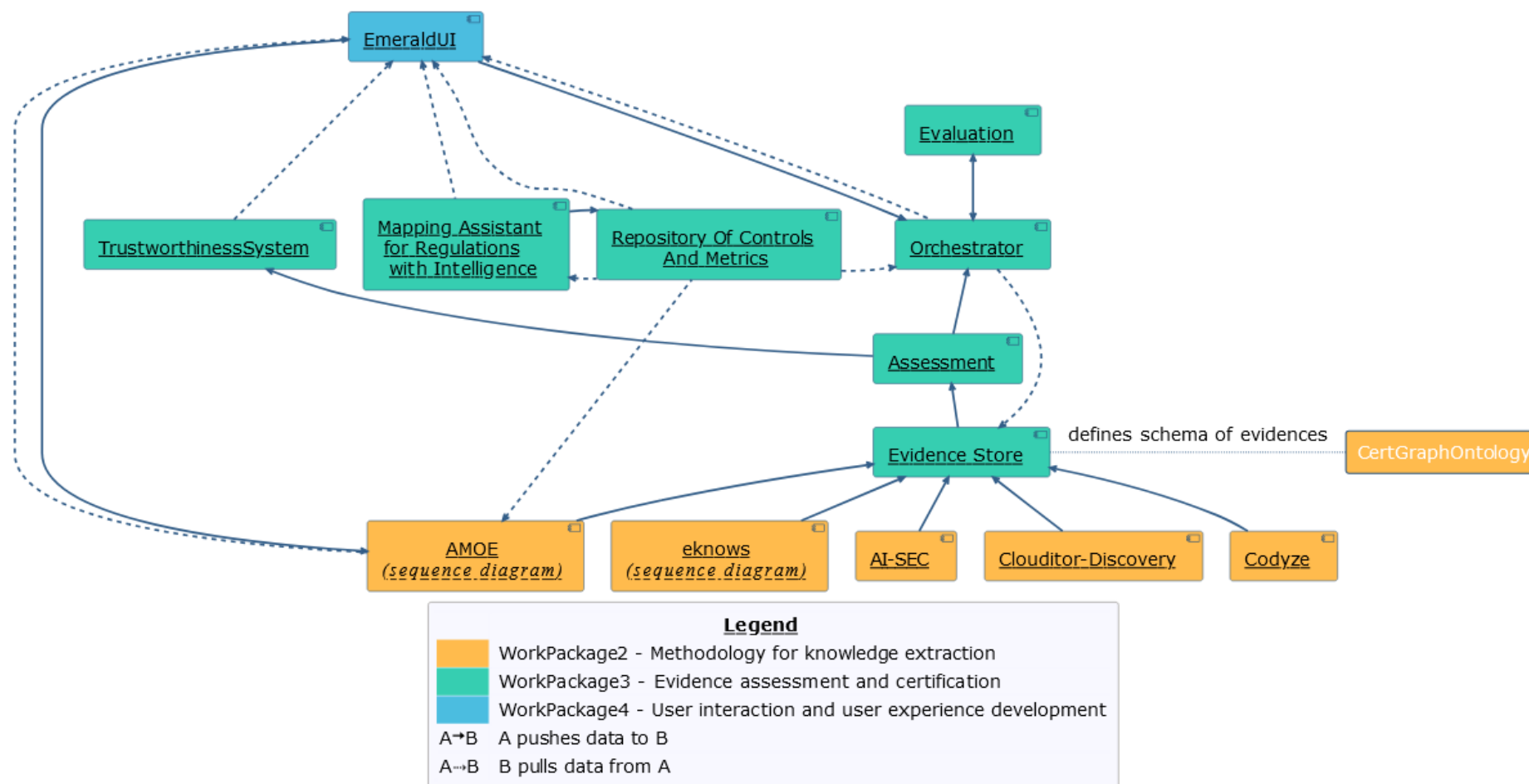


Figure 2. Overview of the EMERALD Components

3.1 Evidence Collector Data Models

All the evidence collector components developed in WP2 collect different forms of data and extract evidence. The results are then shared in the EMERALD framework. This section describes relevant data classes used internally by them and how they relate to other components. The main connections of these components are to the **Evidence Store** and the **Repository of Controls and Metrics**.

Furthermore, the subsections below describe the main techniques for transforming raw evidence data into the EMERALD evidence class objects. Part of the data classes of the components (e.g., **Clouditor-Discovery**) are based on the *CertGraphOntology* model (the EMERALD Graph Ontology), which is described in D2.1 [2]. Please note that the *CertGraphOntology* is not a component, but a central ontology for storing evidence in a graph-based format.

3.1.1 AI-SEC

AI-SEC is an evidence collection tool that extracts various information from ML models. The data model of the tool currently consists of a single main class, *AI-SECEvidence*, which represents the extracted evidence (see Figure 3). Evidence results and closely related information are also stored in the *AI-SECEvidence* class (*result*). This class also contains a unique identifier (*id*), given resources, such as data and model (*sourceFilename*), and the criteria used for extracting evidence (*criteriaId*).

AI-SEC employs various measurements to extract evidence from ML models. By providing **AI-SEC** with a set of data and a trained model, the tool can extract evidence and information about different properties of the model. The output results (evidence and information) can be a string, a vector or a matrix, depending on the measurement used.

Measurement methods are chosen on the basis of the Criteria Catalogue for AI Cloud Services – AIC4⁵, such as adversarial robustness or explainability of the model. A detailed description of the annotation plan and process will be provided in two future dedicated deliverables D2.6 “ML model certification–v1” (M12) and D2.7 “ML model certification–v2” (M24).

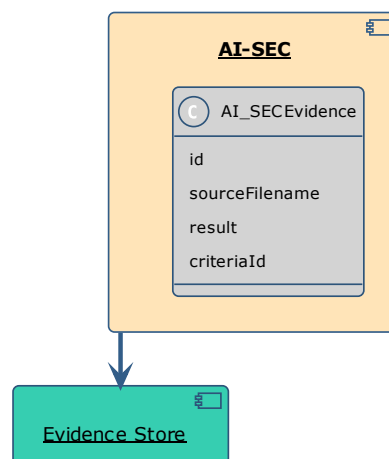


Figure 3. Overview of the AI-SEC component data model

⁵ https://www.bsi.bund.de/EN/Themen/Unternehmen-und-Organisationen/Informationen-und-Empfehlungen/Kuenstliche-Intelligenz/AIC4/aic4_node.html

3.1.2 AMOE

The **AMOE – Assessment and Management of Organisational Evidence** – component extracts evidence from policy PDF documents. The component stores the uploaded files, as well as relevant metadata related to the document and metrics. At the moment of writing, there are two main data classes in the data model (see Figure 4): *AmoePolicyFile* and *AmoeEvidence*. *AmoePolicyFile* serves as an internal representation of the uploaded file, which can be linked to a *Cloud Service* via its id, while *AmoeEvidence* is the internal representation of the extracted data and is created for a set of Security Metrics during the extraction process. The evidence result is stored in the *AmoeEvidence* class in closely related fields, such as *context* or *complianceComment*.

The data is stored in a MongoDB⁶ and can be retrieved through the **AMOE** API endpoints. The internal data classes of **AMOE** will change in the next few months, according to the requirements elicited for **EmeraldUI** (detailed in D4.1 [3]) and further development of **AMOE**.

AMOE is using an NLP (Natural Language Processing) based approach to extract evidence. It utilizes pre-trained models to select text of the policy documents that are relevant for audits. The models used at the moment of writing are specialized on different aspects, such as question answering or computing text representations (embeddings) or text classification. The extracted text passages are then stored in *AmoeEvidence*. The relevant information stored in *AmoeEvidence* will be transformed into an *Evidence* class object and will be forwarded to the **Evidence Store** component. Details on the approach of the **AMOE** component and its related Task 2.3 will be reported in two future dedicated deliverables D2.4 “AMOE–v1” (M12) and D2.5 “AMOE–v2” (M24).

To ensure high quality output from **AMOE**, it is necessary to associate the text samples of the with the Security Metrics. Therefore, *AmoeEvidence* is directly related to the *SecurityMetric* class of the **Repository of Controls and Metrics** (see full data model in Figure 1, AMOE data model in Figure 4, and RCM data model in Figure 10). The plans on annotation and the detailed description of the process will be conducted in the Task 2.3 and reported in the previously stated deliverables.

Furthermore, *AmoeEvidence* is related to the **Orchestrator** (*Cloud Service*) and the **Evidence Store** (*AssessmentResult*, *Evidence*), and *AmoeFile* is related to the **Orchestrator** (*Cloud Service*)

Finally, the information from **AMOE** can be accessed via API and used via the upcoming **EmeraldUI**.

⁶ <https://www.mongodb.com/>

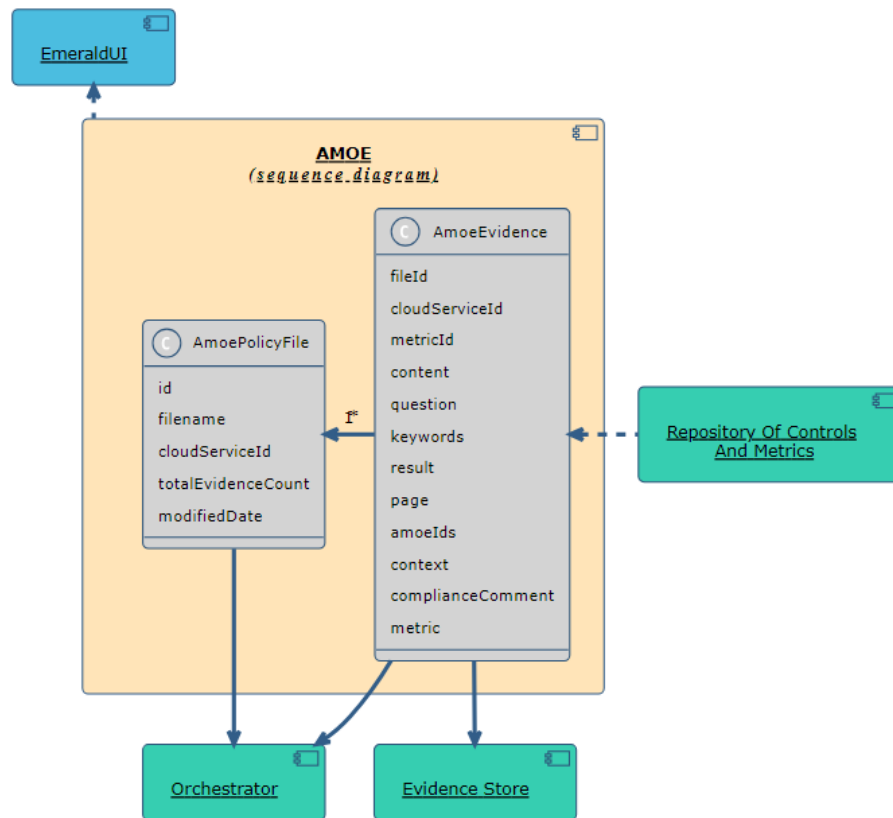


Figure 4. Overview of the AMOE component data model

3.1.3 Cloudditor-Discovery

The **Cloudditor-Discovery** component is an evidence gathering tool which extracts Cloud configurations for different Cloud resources (e.g., Virtual Machine, Object Storage, Network Interface) from several Cloud providers (e.g., Azure) via API calls.

The retrieved Cloud configuration information is stored in an internal *Resource class* object with the properties according to the respective definition in the EMERALD Graph Ontology (see D2.1 [2]). An example of a *Resource* object of a Virtual Machine can be found in Listing 1.

Besides the *Resource* class object, the **Cloudditor-Discovery** stores the gathered information in the *CloudditorDiscoveryEvidence* class object, which is the same class object as the *Evidence* provided by the **Evidence Store** component (see Figure 5). *Evidence* objects are stored in the **Evidence Store** component, a description of the *Evidence* can be found in Section 3.6.

The link from the **Orchestrator** to the *cloud_service_id* property in the *CloudditorDiscoveryEvidence* class refers to the *Cloud Service* defined in the **Orchestrator** component.

Details on the approach of the **Cloudditor-Discovery** component and its related Task 2.5 will be reported in two future dedicated deliverables D2.8 “Runtime evidence extractor–v1” (M12) and D2.9 “Runtime evidence extractor–v2” (M24).

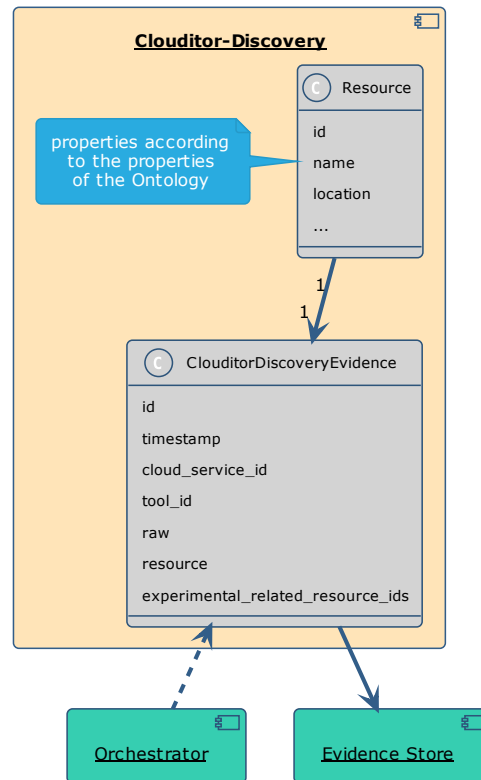


Figure 5. Overview of the Cloudfitor-Discovery component data model


```

message VirtualMachine {
    option (resource_type_names) = "VirtualMachine";
    option (resource_type_names) = "Compute";
    option (resource_type_names) = "CloudResource";
    option (resource_type_names) = "Resource";

    google.protobuf.Timestamp creation_time = 2132;
    string id = 15888 [(buf.validate.field).required = true];
    bool internet_accessible_endpoint = 11229;
    map<string, string> labels = 12634;
    string name = 5434 [(buf.validate.field).required = true];
    // The raw field contains the raw information that is used to
    fill in the fields of the ontology.
    string raw = 17236;
    ActivityLogging activity_logging = 17610;
    AutomaticUpdates automatic_updates = 7698;
    repeated string block_storage_ids = 14852;
    BootLogging boot_logging = 4303;
    EncryptionInUse encryption_in_use = 5839;
    GeoLocation geo_location = 17337;
    MalwareProtection malware_protection = 5352;
    repeated string network_interface_ids = 150;
        OSLogging os_logging = 14872;
    repeated Redundancy redundancies = 11599;
    RemoteAttestation remote_attestation = 16051;
    optional string parent_id = 7061;
    ResourceLogging resource_logging = 17205;
    UsageStatistics usage_statistics = 4834;
}

```

Listing 1. Example of Virtual Machine properties

3.1.4 Codyze

The **Codyze** component is a static source code analysis tool which analyses source code of applications comprising Cloud services and assesses security-relevant implementation details. The analysis report presents implementation details that meet or respectively violate specified security requirements. As part of a CI/CD pipeline, **Codyze** acts as a quality and compliance gate allowing only the delivery of applications that meet security requirements and preventing it otherwise. Each update to the application's source code or new release can trigger an execution of the CI/CD pipeline and thereby **Codyze**. In addition, manual or scheduled assessments are possible.

Codyze is developed in Kotlin⁷ and uses a graph-based representation of source code utilizing the concept of a code property graph. The resulting representation is largely programming language agnostic. Thus, it facilitates the implementation of generic, reusable source code

⁷ [https://en.wikipedia.org/wiki/Kotlin_\(programming_language\)](https://en.wikipedia.org/wiki/Kotlin_(programming_language))

analysis techniques. Currently, **Codyze** supports the programming languages C, C++, Java, Go and Python.

Within EMERALD, **Codyze** interacts with the **Orchestrator** to orchestrate its analysis, and reports its findings as evidence to the **Evidence Store** (see Figure 6). Thereby, **Codyze** generates an analysis report in SARIF⁸ (*CodyzeSarif*). This report contains raw evidence from **Codyze**'s analysis, which is persisted to the **Evidence Store** to facilitate further analysis externally to **Codyze**. Moreover, **Codyze** processes the findings in the SARIF report into evidence for the EMERALD framework. Each finding is converted into a *CodyzeEvidence* that identifies the analysed Cloud Service (*cloudServiceId*), specifies the analysed resource (*resource*), links it to the underlying SARIF report (*sarifId*), classifies the finding according to the EMERALD ontology (*ontologyRef*) and summarizes the result (*result*).

Details on the approach of the **Codyze** component and its related Task 2.2 will be reported in dedicated deliverables D2.2 "Source Evidence Extractor–v1" (M12) and D2.3 "Source Evidence Extractor–v2" (M24).

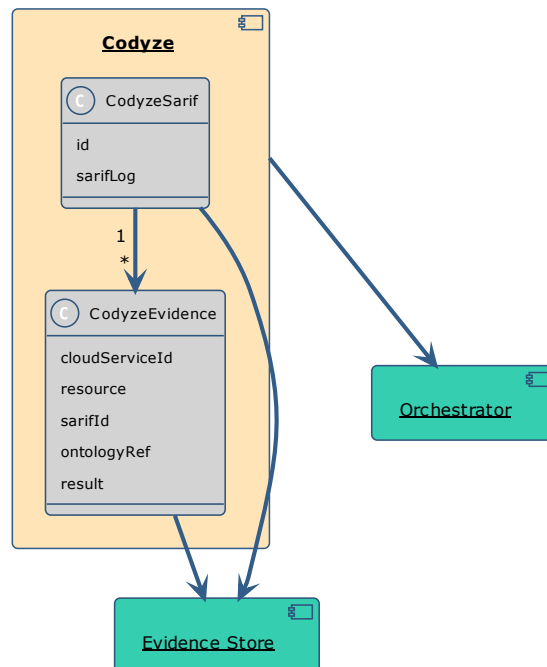


Figure 6. Codyze component overview

3.1.5 eknows

The **eknows** component – based on a platform for multi-language reverse engineering and documentation generation – extracts evidence from source code files. The source code files are collected from the Cloud Service environment at certain points in time. A set of predefined triggers will be available (e.g., once a week/month/etc., or upon changes) to configure the points in time according to the respective use case. **eknows** stores the collected files, as well as relevant metadata related to the sources (e.g., from code repositories) and metrics.

eknows uses static code analysis to extract evidence. The Java-based software platform provides a modular, extensible set of software components for (i) source code parsing using language-

⁸ Static Analysis Results Interchange Format (SARIF), <https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html>

specific frontends (currently more than 16 programming languages, including Java and Python) (extraction), (ii) transformation of parsed source code into a generic abstract syntax tree (GASTM), (iii) structural and language-independent analysis of security-related information, and (iv) reporting of analysis results for security metrics. The extracted and analysed raw evidence is then forwarded to the **Evidence Store** component.

At the moment of writing, **eknows** comprises two main data classes (see Figure 7): *EknowsSourceCodeFile* and *EknowsEvidence*.

EknowsSourceCodeFile serves as an internal representation of the source code file to be analysed, identified by a unique identifier (*id*). It contains attributes denoting the repository from where to collect the file (*repositoryPath*), the name of the file (*filename*), the corresponding *Cloud Service* (*cloudServiceId*), the generated abstract syntax tree (AST) model for the parsed source code (*compilationUnit*), and further metadata, such as the *modificationDate*.

EknowsEvidence is the internal representation of the extracted data of the source code file (*fileId*). This is created for a security metric (*metricId*) during the extraction process. The evidence result (*result*) is stored in the *EknowsEvidence* class, as well as closely related attributes, such as *lineOfCode* or *analyzerVersion*. The internal data class of **eknows** will change in the next few months, according to the requirements defined for the **EmeraldUI** in D4.1 [3] and further needs of the pilot partners.

EknowsEvidence is related to the **Orchestrator** (*Cloud Service*) and the **Evidence Store** (assessment results, raw evidence). *EknowsSourceCodeFile* is related to the **Orchestrator** (*Cloud Service*). The information from **eknows** can be accessed via APIs (Java, REST and/or CLI (Command Line Interface)) and used via the upcoming **EmeraldUI**.

Details on the approach of the **eknows** component and its related Task 2.2 will be reported in dedicated deliverables D2.2 “Source Evidence Extractor–v1” (M12) and D2.3 “Source Evidence Extractor–v2” (M24).

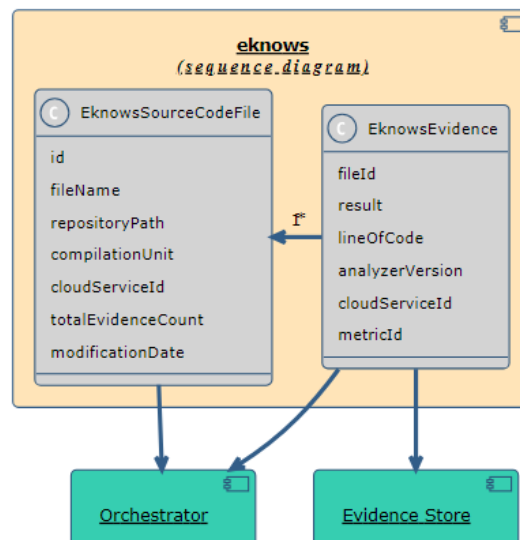


Figure 7. Overview of the *eknows* component data model

3.2 Trustworthiness System (TWS) Data Model

The **TWS** component securely stores the information and associated metadata of evidence and assessment results on the Blockchain to be able to guarantee its integrity and transparency through the **EmeraldUI**.

Due to the use of Blockchain, sensitive information such as evidence and assessment results are not stored and just a summary of them is recorded on the Blockchain through identifiers and hashes. In fact, in the case of assessment results, two different hashes are included: the assessment result itself and the compliance comments. The evidence and assessment result themselves are kept in a local storage - **Evidence Store** and **Assessment** components respectively.

In addition, **TWS** also records metadata information to provide some context. In the case of evidence, they are usually related to specific Cloud Services (*cloudServiceId*) and the cloud resources to which they refer (*resourceId*). In the case of an Assessment Result, the requirement to which it refers (*requirementId*), and the associated evidence identifiers considered in the assessment (*evidenceIds*) are also stored. Finally, for both evidence and assessment results, recording information about the timestamp when they were created (*timestamp*) is also useful.

As a result, Figure 8 summarises the current data model for evidence (*TrustworthyEvidence*) and assessment results (*TrustworthyAssessmentResult*) to be recorded on the Blockchain-based **TWS**. It also shows the interactions with other components: i) with the **Assessment** component, which provides information to be recorded in the **TWS**, and from where the **TWS** retrieves the actual evidence and assessment results to validate their integrity; ii) with the **EmeraldUI**, which provides a graphical interface for users to automatically validate the integrity status of the Evidence and Assessment Results.

Details on the approach of the **TWS** component and its related Task 3.5 have been reported in D3.1 [4].

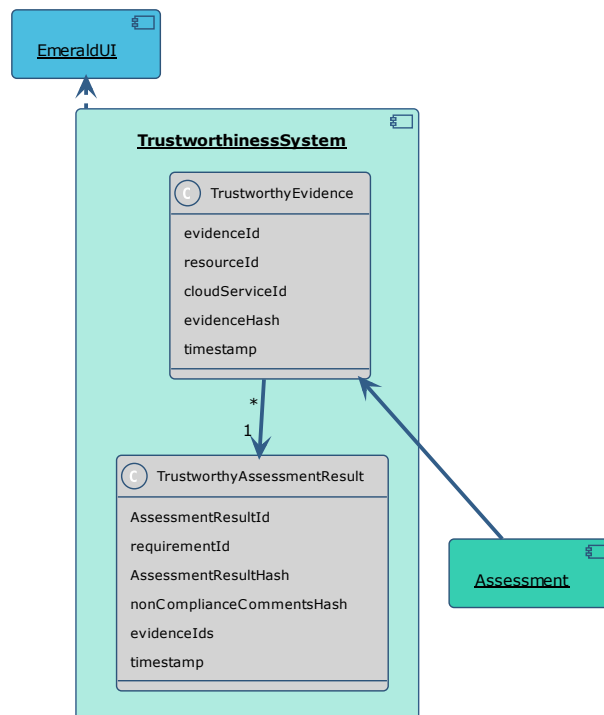


Figure 8. Overview of the Trustworthiness System component data model

3.3 Mapping Assistant for Regulations with Intelligence (MARI) Data Model

MARI – Mapping Assistant for Regulations with Intelligence - is the component that using Deep Learning and state-of-the-art NLP tools is able to create an automatic association between:

- A security control and a security metric
- Two security controls from two different certification schemes.

MARI is based on the previous work in MEDINA's Metric Recommender⁹ [5], which took the description of an EUCS security requirement in natural language, the description of a list of metrics, again in natural language, and as a result returned the list of metrics in descending order of relevance. To do this, the textual descriptions of the metrics and requirements are transformed into feature vectors by pre-trained models (in particular, the best association results in MEDINA were obtained with fastText¹⁰). A K-d tree is computed on the feature vectors of the metrics, which can be used to select the k closest neighbours of the requirement vector, based on the shortest Euclidean distance. Thus, we were able to obtain a metric-requirement association.

At the time of writing, the development of **MARI** is focused on the internal architecture rather than on interactions with other EMERALD components. Also, we are considering different clustering techniques and different embedding production techniques. We will also extend the **MARI** functionalities to deal with more certification schemes (in fact, the automatic association between controls from different schemes is a novelty of EMERALD), and as per the work description, we will develop different strategies to work with (e.g., take a subset of metrics that are useful to get a certain level of certification).

Figure 9 shows a first approach of the **MARI** data model., based on the EUCS scheme [6]. The **RCM** data classes *SecurityMetric* and *SecurityRequirement* are taken as input to produce two new data classes, *SecurityRequirementsAssociation* and *MetricRequirementAssociation*. These associations are the results of **MARI** processing. Please note that this data model is subject to change in the coming releases due to the introduction of other certification schemes.

In addition, a refined internal data class, *data storage*, and calls to the component will be reviewed over the coming months, both based on requirements from other components that **MARI** interacts with, and as the component evolves. Details on the approach of **MARI** component and its related Task 3.3 have been reported in D3.1 [4].

⁹ <https://git.code.tecnalia.com/medina/public/nl2cnl-translator>

¹⁰ <https://fasttext.cc/>

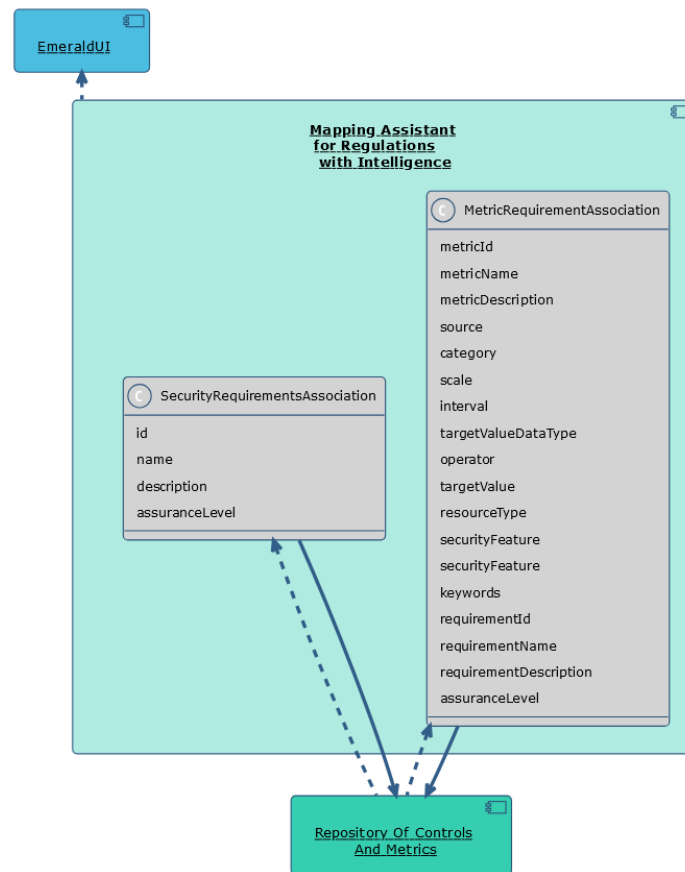


Figure 9. Overview of the MARI component data model

3.4 Repository of Controls and Metrics (RCM) Data Model

The **Repository of Controls and Metrics (RCM)** provides a central point in EMERALD framework where the certification schemes are stored and managed. The repository can contain different schemes and includes a complete information of each scheme, with the corresponding categorization.

A first approach of the **RCM** internal data model is based on the EUCS scheme [6] (see Figure 10)¹¹, while it is subject to change in the coming releases due to the introduction of other schemes (e.g., BSI C5¹² or AIC4¹³ are some foreseen ones). Because of this, the principal data classes implemented in the **RCM** are *SecurityControlFramework*, *SecurityCategory*, *SecurityControl* and *SecurityRequirement*, that reflect the organization of the EUCS framework. Along with these, some other auxiliary entities are implemented, such as *SimilarControls* – to support mapping among controls of different schemes - and *ImplementationGuidelines* – to help the user with the implementation of the requirements. **RCM** also incorporates the definition of the *SecurityMetric* class used in EMERALD to define what to measure to assess the collected evidence.

The RCM classes have interactions with other EMERALD components as follows:

¹¹ Please note that an enlarged view of the RCM data model is available in *APPENDIX: Release 1.0.9 of Architecture and Data Modelling*.

¹² https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/CloudComputing/ComplianceControlsCatalogue/2020/C5_2020.pdf?__blob=publicationFile&v=3

¹³ https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/CloudComputing/AIC4/AI-Cloud-Service-Compliance-Criteria-Catalogue_AIC4.html

- *SecurityControlFramework*, *SecurityControl*, *SecurityMetric* and *SecurityRequirement* are related with the **Orchestrator**, which needs also to internally manage the schemes.
- *SecurityMetric* is also related with the **AMOE** and the **Assessment** components.
- *SecurityMetric* and *SecurityRequirement* are also shared with the **MARI** component.

Another functionality offered by the **RCM** is a *Questionnaire* to provide users the possibility to perform a self-assessment to check compliance with the EUCS scheme. The Questionnaire-related data classes, which are enclosed in a box in the diagram (see Figure 10), are as follows: *Questionnaire*, *QuestionnairePurpose*, *QuestionnaireLevel*, *Question*, *QuestionnaireAnswer*, *QuestionnaireNonConformities*, and *jhiUser*. All these entities are devoted to (i) Implement several questions per requirement, (ii) manage the responses given; (iii) calculate the results for this specific user, and (iv) offer the degree of compliance with the EUCS scheme regarding the selected assurance level.

Finally, the **EmeraldUI** component is also related with the data entities used in the **RCM** in order to provide the final user with a graphical view of the schemes contained in the **RCM** and all the associated information.

Details on the approach of the **RCM** component and its related Task 3.2 have been reported in D3.1 [4].

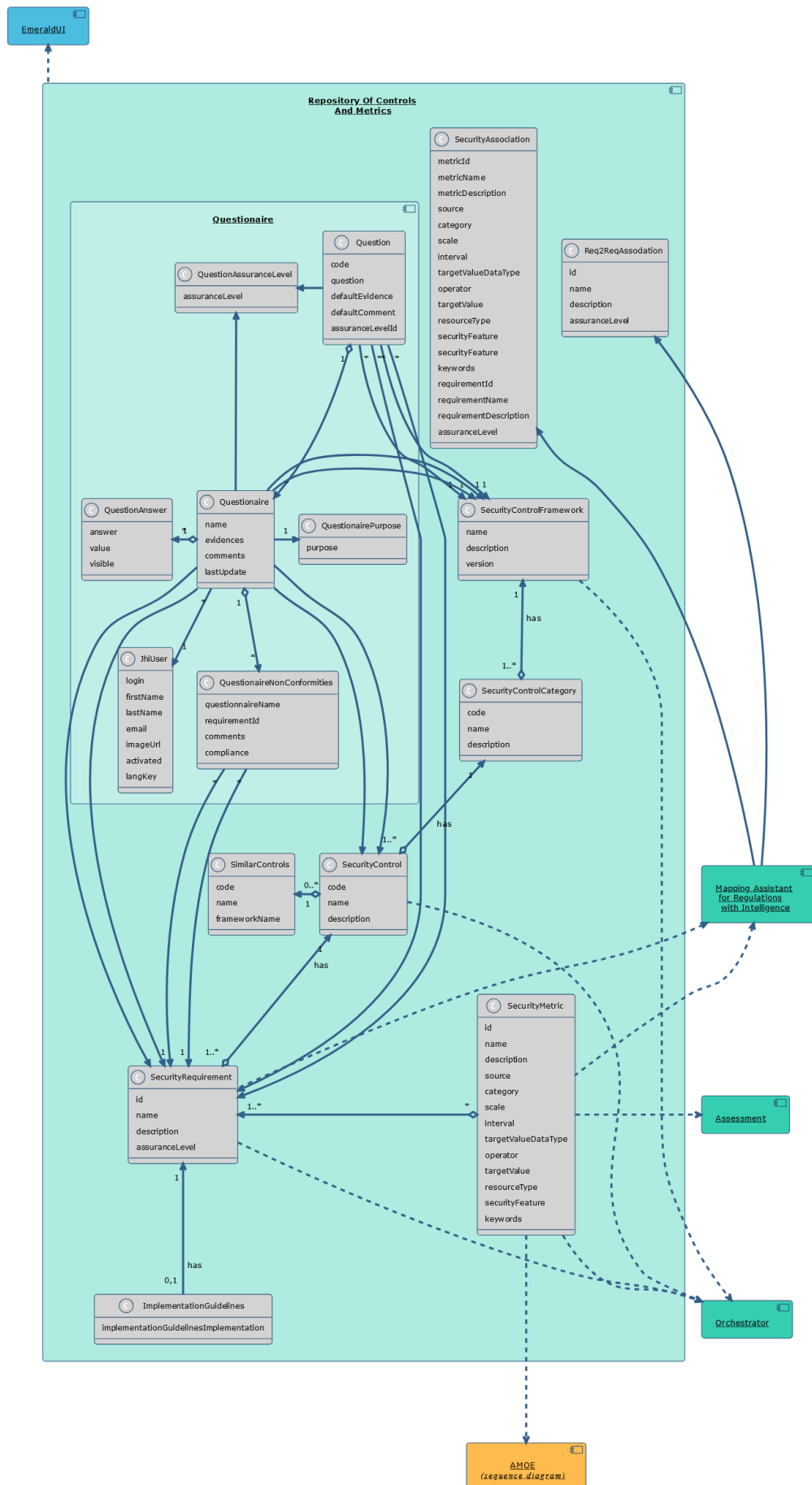


Figure 10. Overview of the RCM component data model

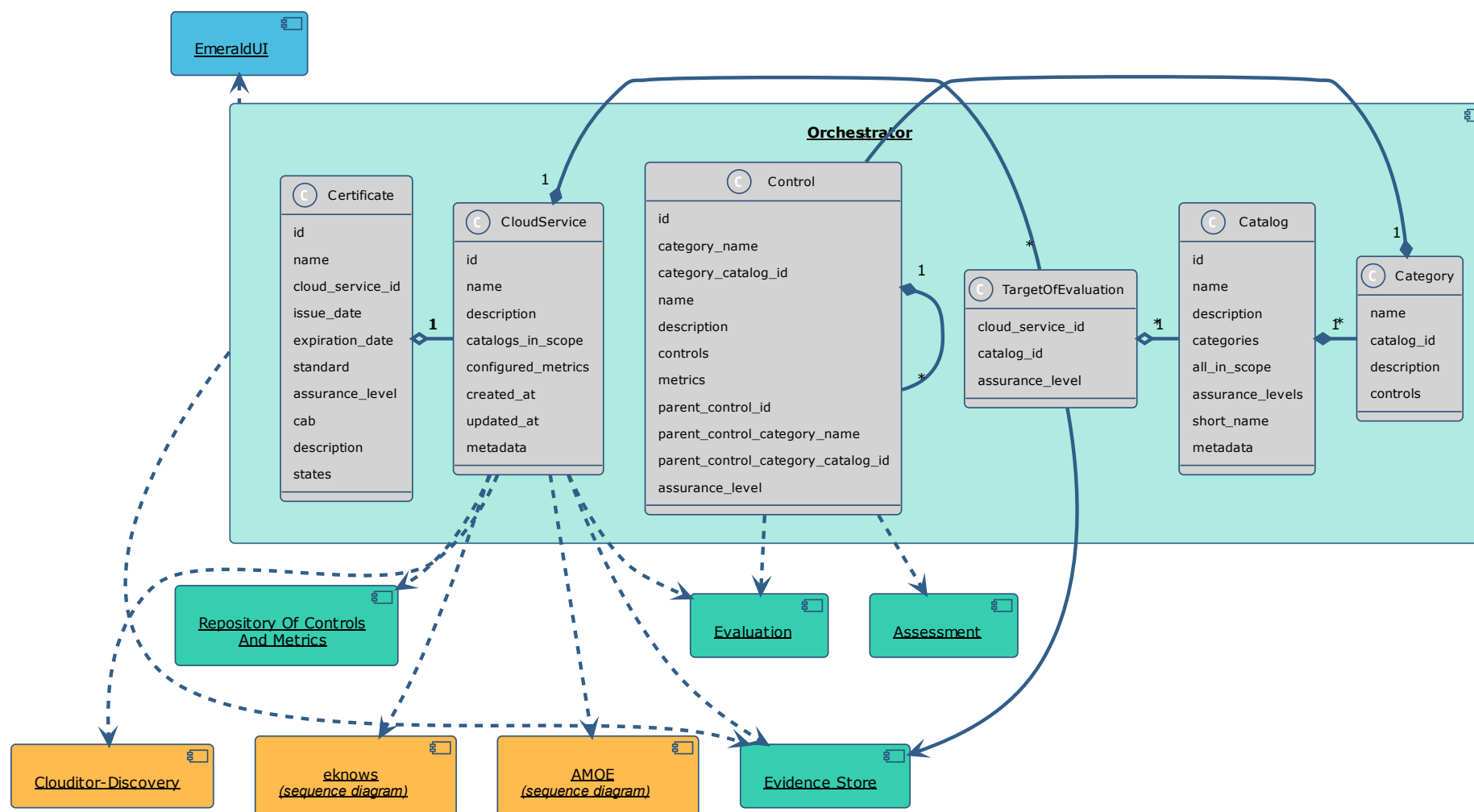
3.5 Orchestrator Data Model

The **Orchestrator** is the central management and orchestration component in EMERALD. Its main purpose is to hold all dynamic information about the current audit process, such as the *Target Of Evaluations*, the evaluated *Cloud Services*, all *Assessment Results*, and the final *Certificate* state (see Figure 11). Furthermore, it fetches static data from the **RCM**, such as the available schemes and its associated metrics. For performance reasons this data (*SecurityControlFramework*, *SecurityControlCategory*, *SecurityControl*, *SecurityRequirement* and *SecurityMetric*) is cached in the **Orchestrator**. The most important dynamic data classes are:

- *CloudService*, which holds the logical representation of a single service, which aims to be certified.
- *TargetOfEvaluation*, which takes an existing *cloud_service_id* and combines it with one dedicated security catalogue to produce a *Certificate*.
- *Certificate*, which is the data class representing different states and is related to the *EvaluationResults*.
- *Control*, which is the neutral representation of either a control, requirement or objective (this definition of *Control* is similar to the term defined in OSCAL¹⁴). Since every *SecurityControlFramework*/security scheme uses different names, the **Orchestrator** normalizes them in the *Control* data class. In addition, each *Control* can have sub-controls, which allows to include different *SecurityControlFrameworks* in EMERALD.

Details on the approach of the **Orchestrator** component and its related Task 3.1 have been reported in D3.1 [4].

¹⁴ <https://pages.nist.gov/OSCAL/resources/concepts/terminology/#control>



3.6 Evidence Store Data Model

The **Evidence Store** is the central component to store evidence from the evidence collector components, which send the generated evidence directly to the **Evidence Store**. The main data class is *Evidence*, which holds the necessary information regarding the collected evidence (see Figure 12) and whose important fields are the following:

- A unique identifier (*id*) for each evidence. It needs to be a UUID
- *timestamp* describing when the evidence was created
- *cloud_service_id* of the *Cloud Service* the evidence belongs to
- *tool_id* is the ID of the evidence collector tool that created the evidence (such as **Codyze**, **eknows**, **Clouditor-Discovery**, ...)
- *resource* contains the resource properties of the discovered resource. It is described according to the terms of the EMERALD Graph Ontology (see D2.1 [2] for an initial version).

Evidence is sent to the **Assessment** component and can be retrieved via the **Orchestrator** API.

Details on the approach of the **Evidence Store** component and its related Task 3.1 have been reported in D3.1 [4].

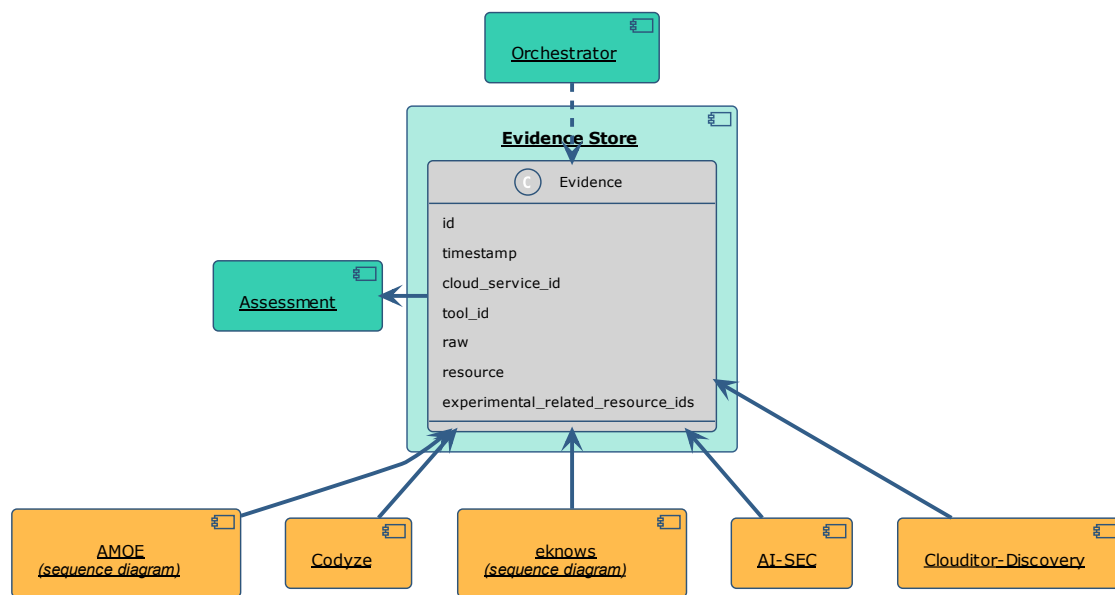


Figure 12. Overview of the Evidence Store component data model

3.7 Assessment Data Model

The **Assessment** component assesses the evidence stored in the **Evidence Store** by using the metric definitions from the **RCM**. The needed metrics are called from the **Orchestrator** and the evidence are sent directly from the **Evidence Store** to the **Assessment**. The important data classes are the following:

- *Metric* contains the metadata and a link to the corresponding *MetricImplementation*
- *MetricImplementation* contains the implementation used by the assessment with the specific code and the code language
- *MetricConfiguration* contains the target value and the operator used in the assessment and can be specified separately for each *Cloud Service*

- *AssessmentResult* contains the result of the assessment, including the used *evidence_id*, *metric_id* and *metric_configuration*.

The *AssessmentResults* are sent to the **Trustworthiness System** and can be retrieved via the API endpoints of the **Orchestrator**. Figure 13 depicts the diagram for the **Assessment** data model.

Details on the approach of the **Assesment** component and its related Task 3.4 have been reported in D3.1 [4].

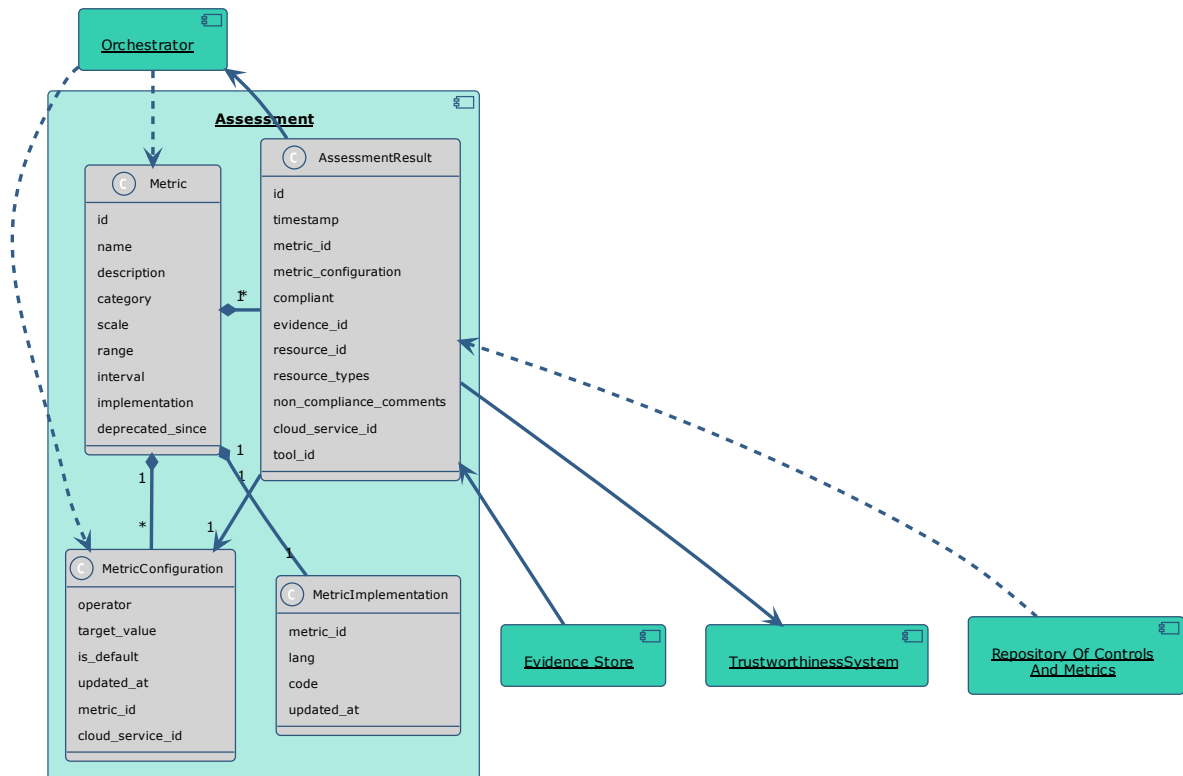


Figure 13. Overview of the Assessment component data model

3.8 Evaluation Data Model

The main purpose of the **Evaluation** component is to map the measurements of individual metrics (i.e., *AssessmentResults*) and combine them according to the mapping of a metric to a *Control*. This is defined as an *EvaluationResult* (see Figure 14), the most important fields of which are:

- Its *id*, which is a UUID to make it unique
- The combination of the *Cloud Service* (through its *cloud_service_id*) and a control (through its *control_id* and associated *catalog_id* identifiers)
- A *timestamp*
- A *status*, which can either be compliant, not compliant or waiting for more data
- Optionally, a second *valid_until* field, which describes the validity of this result. This is mainly used for evaluation results that are created manually (e.g., for controls which cannot be measures automatically).

Usually, one or more metrics define the compliance state of a control. Currently, all of the assessment results need to be compliant for the evaluation result to be compliant. This might change in the future if more sophisticated logical operations are needed. For example, it could

be possible that either one or another metric is sufficient to demonstrate compliance to the control.

Details on the approach of the **Evaluation** component and its related Task 3.4 have been reported in D3.1 [4].

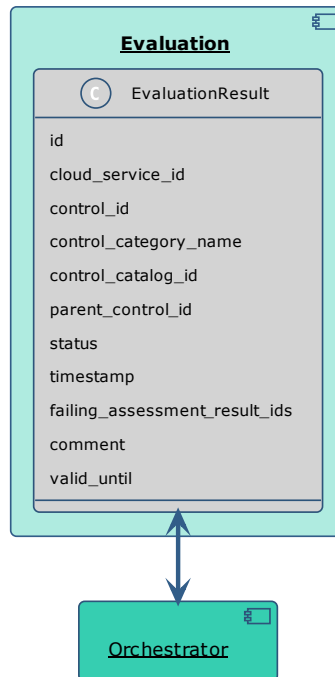


Figure 14. Overview of the Evaluation component data model

4 Interactive Documentation

This section describes the web-based documentation approach used to share the data model within the EMERALD project. The main technologies used are PlantUML¹⁵, Nginx¹⁶ web service and Gitlab¹⁷. The main objective is to have a centralized documentation that can be viewed from any device, without the need to install any tools.

4.1 PlantUML

To allow for easy text-based creation of the data model, the PlantUML tool was chosen. This tool supports a wide range of diagrams – some of which have included in our documentation e.g., class diagrams, sequence diagrams, component diagrams. As the diagrams are based on structured text, very similar to common programming languages, the implementation is straight forward and can therefore be easily adapted into code or vice versa.

PlantUML allows to render the diagrams in different output formats. The most commonly used in the project are PNG and SVG. The latter is important for the web-service – the option ‘!pragma svginteractive true’ switches the diagrams from static boxes to dynamically highlighted on hover or click. As it might be hard to track connected classes in a huge class diagram, a class can be clicked or hovered and all related classes are highlighted as well.

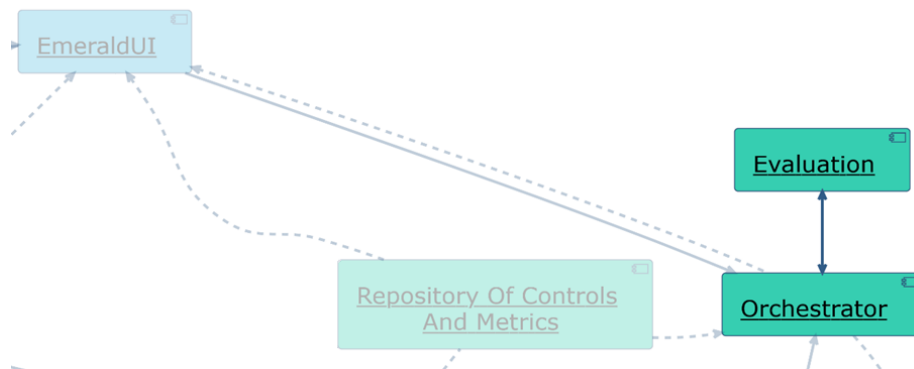


Figure 15. Interactive SVG - highlight neighbours on click

Figure 15 depicts an example: the **Evaluation** component was clicked, and the direct neighbour **Orchestrator** is highlighted, whereas the other links and components are faded.

Furthermore, PlantUML allows to set variables and themes to use the EMERALD colour scheme on all diagrams. This is convenient as the colour scheme can be imported for each diagram and does not need to be set manually for each element. The different diagram files can be included in other files, which reduces redundant information, and the main classes of each subcomponent need to be defined in a single file. The names of the components are set as variables including links to the overview diagrams – so no manual linking is required.

4.2 Web Service

To make the diagrams more accessible, a simple html page was created that includes some basic JavaScript functionalities to switch the diagrams displayed. The landing page (<https://models.emerald.digital.tecnalia.dev/>) shows an overview of the components (see Figure 16). Users can click component titles to switch to the respective overview diagram. Furthermore, a navigation bar at the top of the page allows quick access to the component overview (“Components” menu option) or the data diagram page (“Data diagram” menu

¹⁵ <https://plantuml.com/>

¹⁶ <https://nginx.org/en/>

¹⁷ <https://en.wikipedia.org/wiki/GitLab>

option). The idea is to start with a generic overview and then drill down to see the details of a component. Although the data diagram is quite large, you can focus on a single component by clicking on it.

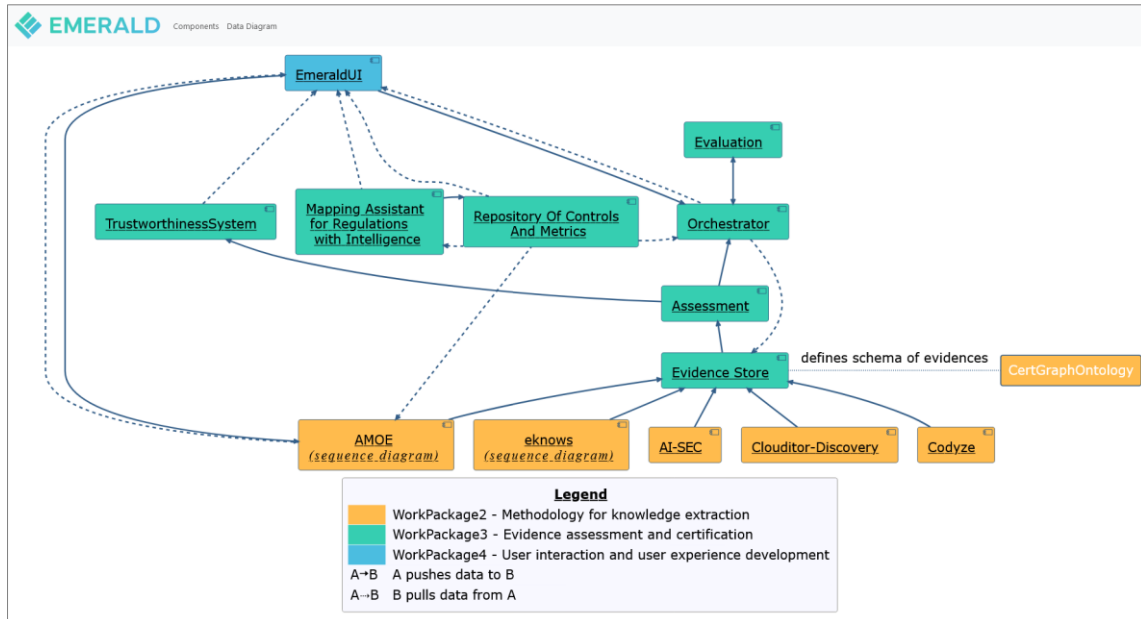


Figure 16. Landing page of the interactive documentation

4.2.1 Implementation details

Once the diagrams are rendered, the interactive documentation can be deployed locally without any need of a web service by simply opening the index.html file. However, for ease of access - and to always have access to the newest release - we are using Dockerfiles¹⁸ to generate a nginx based image that can be deployed on the EMERALD Kubernetes cluster.

The structure of the web service is as follows:

```
./index.html
./imgs/logo.svg
./out/*_data.svg
./out/*_component.svg
./out/*_Sequence_Diagram.svg
```

The index.html file contains the basic structure and scripts to load the diagrams. The EMERALD logo is stored in /imgs. The rendered diagrams are stored in /out and are loaded on demand. This implementation is portable to any device that supports a modern web browser by simply copying the files. In the Nginx web service, the files are located in /usr/share/nginx/html.

4.3 Data model versioning

As the PlantUML based diagrams contain text/code, the files can be used in versioning systems such as git¹⁹. This allows for different organisational processes, that are not possible in common online tools with graphical support (e.g., draw.io²⁰ – although it allows versioning, there are no processes to keep different versions of the diagrams and proposed changes, as it is possible

¹⁸ <https://docs.docker.com/reference/dockerfile/>

¹⁹ <https://git-scm.com/>

²⁰ <https://www.draw.io>

using text-based diagrams and git + GitLab²¹). Different versions of the diagrams can be stored in commits, and merge requests can be created to deal with changes to the diagrams.

The process to add changes to the data model was defined as follows: major changes are completed in a separate branch – when finished, a merge request should be created in the EMERALD GitLab and the changes will be reviewed to check for inconsistencies and breaks to the interactive, web-service-based deployment. After the review, the new version will be merged, which triggers the build pipeline and a new release will be deployed to the EMERALD Kubernetes cluster. The latest release version of the diagrams will then be available to all developers and can be retrieved at <https://models.emerald.digital.tecnalia.dev/>. If there are any problems, or additional diagrams are needed, Gitlab's issue functionality can be used to document, communicate and coordinate the required changes.

²¹ <https://gitlab.com/>

5 Data Exchange and Formats

This section provides a short overview of the planned data exchange approach, as well as the formats used. Although all EMERALD components use different data types, they all communicate in a standardized way and format, which speeds up development, as components do not need to build special data connectors for different tools.

5.1 Interaction mechanisms between components

The interaction between the components will be implemented using REST²² – representational state transfer. Each component is using and/or serving REST-APIs that are documented in the OpenAPI²³ specification files. This helps developers to share the different endpoints and allows for code for client interfaces to be generated. Some components may also offer gRPC connections (Remote Procedure Call framework by Google) to share data between closely related components such as **Evidence Store** and **Assessment**.

The most common format for REST-API will be JSON²⁴, as it allows for easy access of attribute-value pairs and arrays. In EMERALD, some components are based on the predecessor versions developed in MEDINA and have existing APIs following the same approach. These APIs can be extended and adjusted to the needs of the EMERALD framework.

Listing 2 shows the JSON for a piece of evidence that is sent from **AMOE** to the **Evidence Store**. Similarly, Listing 3 shows a more extensive example for data represented in JSON and how it is used by some EMERALD components, such as **Clouditor-Discovery**.

```
{
  "id": "b11a1b4b-4c9f-4135-afbb-f6e30364d881",
  "timestamp": "2024-06-26T18:23:45.123456",
  "cloud_service_id": "3f1c2e4c-8bd5-45d1-a6a3-0f9a9a8e4d35",
  "tool_id": "amoe",
  "raw": "password must contain more than 15 characters",
  "resource": {"id": "165483", "type": ["PolicyDocument"]}
}
```

Listing 2. AMOE evidence in JSON

²² <https://en.wikipedia.org/wiki/REST>

²³ https://en.wikipedia.org/wiki/OpenAPI_Specification

²⁴ <https://en.wikipedia.org/wiki/JSON>

```

{
  "id":
  "/subscriptions/XXXXX/resourcegroups/democlouditorhappy/providers/microsoft
.storage/storageaccounts/democlouditordiagnostics",
  "cloudServiceId": "00000000-0000-0000-0000-000000000000",
  "toolId": "Clouditor Evidences Collection",
  "properties": {
    "@type":
    "type.googleapis.com/clouditor.ontology.v1.ObjectStorageService",
    "creationTime": "2023-07-09T10:35:18.246911100Z",
    "id":
    "/subscriptions/XXXXX/resourcegroups/democlouditorhappy/providers/microsoft
.storage/storageaccounts/democlouditordiagnostics",
    "labels": {
      "owner": "clouditor"
    },
    "name": "democlouditordiagnostics",
    "raw": "/*...*/",
    "geoLocation": {
      "region": "westeurope"
    },
    "httpEndpoint": {
      "url":
      "https://democlouditordiagnostics.[file,blob].core.windows.net/",
      "transportEncryption": {
        "enabled": true,
        "enforced": true,
        "protocol": "TLS",
        "protocolVersion": 1.2,
        "cipherSuites": []
      }
    },
    "parentId":
    "/subscriptions/XXXXX/resourcegroups/democlouditorhappy"
  }
}

```

Listing 3. Clouditor example evidence in JSON

Some components will offer data import / export functionality. The **Repository of Controls and Metrics** is planning to allow import of security schemes using the OSCAL²⁵ format. The API description and more details on the format will be described in the future deliverable D3.3 “Evidence assessment and Certification–Implementation-v1” (M12). The OSCAL format allows different file types and data formats such as YAML²⁶ and JSON. Listing 4 shows a tentative example of the mapping of an EUCS Requirement in OSCAL. It can be seen how the parts of the Control (**ops-02**) are specified using the OSCAL elements “id”, “title”, “properties”, and also with

²⁵ <https://pages.nist.gov/OSCAL/>

²⁶ <https://en.wikipedia.org/wiki/YAML>

“parts” and “prose”; the Requirements are implemented with “parts” within the upper “parts” of Control. The Requirement ID (OPS-02.3) is specified with “properties”, and the requirement itself with “prose”.

```

"controls": [
  {
    "id": "ops-02",
    "title": "CAPACITY MANAGEMENT - MONITORING",
    "properties": [
      {
        "name": "label",
        "value": "OPS-02"
      }
    ],
    "parts": [
      {
        "id": "ops_02_obj",
        "name": "control-objective",
        "prose": "The capacities of critical resources such as
personnel and IT resources are monitored."
      },
      {
        "id": "ops-02_smt",
        "name": "statement",
        "parts": [
          {
            "id": "ops-02_smt.3",
            "name": "item",
            "properties": [
              {
                "name": "label",
                "value": "OPS-02.3"
              }
            ],
            "prose": "The provisioning and de-provisioning of
cloud services shall be automatically monitored to guarantee fulfilment of
OPS-02.1"
          }
        ]
      }
    ]
  }
]

```

Listing 4. A EUCS Requirement mapping in OSCAL

5.2 Sequence diagrams

To illustrate the interactions between the components, sequence diagrams will be created and extended in the future work of Task 1.1. Additional documentation will be provided which can be included in the interactive PlantUML diagrams. At the time of writing this deliverable, the

sequence diagrams for **AMOE** and **eknows** have already been integrated into the diagram collection. The sequence diagrams will be included in future deliverables of EMERALD WP1, in particular in D1.3 “EMERALD solution architecture-v1” (M12) and D1.4 “EMERALD solution architecture-v2” (M24).

6 Conclusions

This document provides an overview of the overall EMERALD data model, as well as a more detailed view of the component data models. The general data model is loosely based on the data model of the predecessor project MEDINA. However, to increase the TRL of the reused MEDINA components and adjust them to the EMERALD framework, all component diagrams received updates. Furthermore, it was extended with additional components, such as **AI-SEC** or **eknows**.

The data model is presented in a web service, to allow interactive investigation of the different diagrams. The diagrams are based on text instructions using PlantUML and then rendered in SVG files. This allows the diagrams to be versioned and the various functionalities of the EMERALD GitLab repository can be used to manage and coordinate the updates. The basic idea of this interactive documentation is to start with an abstract overview (landing page) and then drill down to the different components of interest. The different classes and components of the diagrams can be clicked/hovered to navigate and highlight direct connections.


Finally, this deliverable describes the main data format that will be used for data exchange between EMERALD components and external sources – JSON. To provide more insight, an example for **AMOE** and **Clouditor-Discovery** evidence have been provided. The **Repository of Controls and Metrics (RCM)** will provide import/export functionality of security schemes in OSCAL format – for which a JSON example was also provided.

The data diagrams will be updated according to the needs and changes of the different components. These changes will be subject to the described processes in this deliverable, shared with the consortium in different version releases, and deployed in the EMERALD Kubernetes infrastructure. The updates will be collected and described in a second version of this deliverable (D1.2 – Data modelling and interaction mechanisms v2), planned to be submitted in M18 of the project (end of April 2025).

7 References

- [1] EMERALD Consortium, “EMERALD - Annex 1 - Description of Action - GA 101120688,” 2022.
- [2] EMERALD Consortium, “D2.1 Graph Ontology for Evidence Storage,” 2024.
- [3] EMERALD Consortium, “D4.1 Results of the UI-UX requirements analysis and the work processes–v1,” 2024.
- [4] EMERALD Consortium, “D3.1 Evidence assessment and Certification–Concepts-v1,” 2024.
- [5] MEDINA Consortium, “D5.5 MEDINA integrated solution-v3,” 2023. [Online]. Available: https://medina-project.eu/wp-content/uploads/2023/09/MEDINA_D5.5_MEDINA-integrated-solution-v3_v1.0.pdf. [Accessed July 2024].
- [6] ENISA, “EUCS - Cloud Services Scheme,” [Online]. Available: <https://www.enisa.europa.eu/publications/eucs-cloud-service-scheme>. [Accessed July 2024].

APPENDIX: Release 1.0.9 of Architecture and Data Modelling

In order to allow the readers of this document to consult the documentation and data model themselves, the current version of the files have been archived in a zip file. The contents are images of the different data models, as well as a webpage to aid in navigation. The 1.0.9 release version of the interactive documentation is available here:  [D1.1 Appendix Release 1.0.9 of Architecture and Data Modelling](#)

To open the interactive documentation locally, you need to extract the zip file. Then navigate to the “architecture_and_data_model” folder and open the index.html file in a common web browser.