EMERALD

Deliverable D3.3

Evidence assessment and Certification – Implementation - v1

Editor(s):	Nico Haas
Responsible Partner:	Fraunhofer AISEC
Status-Version:	Final – v1.0
Date:	31.10.2024
Type:	OTHER (SW)
Distribution level:	PU

Project Number:	101120688
Project Title:	EMERALD

Title of Deliverable:	D3.3 Evidence assessment and Certification – Implementation- v1
Due Date of Delivery to the EC	31.10.2024

Workpackage responsible for the Deliverable:	WP3 - Evidence assessment and Certification
Editor(s):	Fraunhofer AISEC
Contributor(s):	Nico Haas, Angelika Schneider (FHG) Cristina Regueiro, Iñaki Etxaniz (TECNALIA) Marinella Petrocchi (CNR)
Reviewer(s):	Jordi Guijarro (ONS) Cristina Martínez Martínez (TECNALIA) Juncal Alonso Ibarra (TECNALIA)
Approved by:	All Partners
Recommended/mandatory readers:	WP1, WP2, WP4, WP5, WP6

Abstract:	Interim versions of the implementation of the WP3 components.
Keyword List:	Implementation, Evidence Assessment, Assessment Evaluation, Certification, Control Metric Mapping, Trustworthiness, Orchestration
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	05.09.2024	Added Table of Contents	Nico Haas (FHG)
v0.2	14.10.2024	Added content	Nico Haas, Angelika Schneider (FHG) Cristina Regueiro, Iñaki Etxaniz (TECNALIA) Marinella Petrocchi (CNR)
v0.3	20.10.2024	QA Review	Jordi Guijarro (ONS)
v0.4	24.10.2024	Address internal QA review comments	Nico Haas (FHG)
v0.5	29.10.2024	Final review	Cristina Martínez /Juncal Alonso (TECNALIA)
v0.6	30.10.2024	Integration of the final review	Nico Haas (FHG)
v1.0	31.10.2024	Submitted to the EC	Cristina Martínez /Juncal Alonso (TECNALIA)

Table of contents

List of tables

List of figures

Terms and Abbreviations

AI	Artificial Intelligence
AI-SEC	AI Security Evidence Collector
API	Application Programming Interface
CaaS	Certification-as-a-Service
CI/CD	Continuous Integration /Continuous Development
CLI	Command Line Interface
CRUD	Create, Read, Update and Delete
CSV	Comma-Separated Values
DB	Database
DoA	Description of Action
EAP	Early Adopters Programme
EBSI	European Blockchain Service Infrastructure
EC	European Blockchain Services Infrastructure
EUCS	European Cybersecurity Certification Scheme for Cloud Services
EVM	Ethereum Virtual Machine
gRPC	Google Remote Procedure Call
GUI	Graphical User Interface
JPA	Java Persistence API
JWT	Java Web Token
KPI	Key Performance Indicator
KR	Key Result
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
GUI	Graphical User Interface
IP	Internet Protocol
MARI	Mapping Assistant for Regulations with Intelligence
NDCG	Normalised Discounted Cumulative Gain
MVC	Model, View, Controller
NLP	Natural Language Processing
OPA	Open Policy Agent
OSCAL	Open Security Controls Assessment Language
OSS	Open-Source Software
Protobuf	Protocol Buffers
RBAC	Role-Based Access Control
RCM	Repository of Controls and Metrics
Rego	Policy query language of OPA
REST	Representational State Transfer
SDLC	Software Development Life Cycle
SQL	Structured Query Language
SSI	Self-Sovereign Identity System
TWS	Trustworthiness System
UI	User Interface
URL	Uniform Resource Locator
WSGI	Web Server Gateway Interface

Executive Summary

This deliverable, the first version of Evidence Assessment and Certification – Implementation, provides an initial report on the implementation details of the WP3 components within the EMERALD framework. The goal of WP3 is to serve as the central integration point for evidence collection and knowledge extraction tools, contributing to the development of a Certification-as-a-Service (CaaS) framework for continuous certification of harmonized cybersecurity schemes by assessing the provided evidence to make appropriate certificate decisions. In particular, WP3 and its deliverables address the key results CERTGRAPH (KR2) by implementing the evidence store as a graph database, OPTIMA (KR3) by providing the optimal set of metrics for a given control of a security scheme, MULTICERT (KR4) by providing certification decision for multiple schemes, and INTEROP (KR7) by providing an interoperability layer for trustworthy systems, assessment results, and catalogue data. These key results are measured using the key performance indicators (KPIs) defined in the Description of Action (DoA) [1], which are outlined below.

WP3 enables continuous certification decisions based on a constantly changing certification target. This deliverable informs about the development and implementation of WP3 components, including the *Clouditor-Orchestrator*, *Clouditor-Assessment*, *Clouditor-Evidence Store*, *Clouditor-Evaluation*, *Mapping Assistant for Regulations with Intelligence* (MARI), *Repository of Controls and Metrics* (RCM), and *Trustworthiness System* (TWS).

We first demonstrate the place of the WP3 components in the EMERALD framework. To do this, we show an overview of all components, in which both the languages used, and the connection protocols are visualized. Finally, the main part of this document delves into each component's implementation, delivery, usage, and associated documentation.

Providing certificate decisions by meeting the ambitious objectives set in EMERALD requires various tools to work cohesively together: assessing evidence coming from the WP2 evidence collection tools (KPI 4.1); storing evidence in a graph-based database to enable sophisticated assessment of evidence distributed across various layers of a cloud service (KPI 2.1); the RCM component to store catalogues and metrics in an interoperable way (KPIs 7.1 and 7.2), the MARI component to provide metrics that are suitable for a given (set of) security schemes (KPIs 3.1 and 3.2), and the TWS component to improve the auditor's trust in the evidence (KPIs 7.1 and 7.2). To implement these components in a manner that ensures cohesive operation, they must be carefully designed and integrated. The main contributions of this deliverable to the project are therefore to focus on the implementation details of each WP3 component, ensuring that they are effectively realized within the whole framework.

The structure of the WP3 deliverables closely resembles the software development life cycle (SDLC) approach. After the first WP3 deliverable (D3.1 “Evidence Assessment and Certification – Concepts-v1” M09 [2]), this deliverable describes the initial implementation (D3.3 “Evidence Assessment and Certification – Implementation-v1” M12) and outlines the next steps, which include further integration (D3.5 [3]). This cycle is then repeated with the final versions of concepts (D3.2 [4]), implementation (D3.4 [5]), and integration (D3.6 [6]), ensuring continuous improvement and refinement of the components (also considering changes occurring in other work packages).

1 Introduction

1.1 About this deliverable

The EMERALD project aims to pave the way towards Certification-as-a-Service (CaaS) for continuous certification of harmonized cybersecurity schemes, such as the EUCS [7]. It addresses the critical need for enhanced transparency, accountability, and trustworthiness in European cloud services. The project focuses on developing robust evidence management components and providing a proof of concept for AI certification schemes.

Within this context, WP3 plays a pivotal role by serving as the central integration point for evidence collection and knowledge extraction tools developed in WP2, while also acting as the interface for auditors and pilots who can interact with it via the UI. The main goal of WP3 is to contribute to the CaaS framework by assessing the provided evidence to make appropriate certification decisions.

This deliverable serves as the initial report on the implementation details of the WP3 components within the EMERALD project. The main goal is to document the implementation of each WP3 component, providing detailed functional and technical descriptions, delivery and usage instructions, and associated documentation.

In summary, this document aims to provide a thorough understanding of the initial implementation, setting groundwork for the upcoming integration within the EMERALD framework.

1.2 Document structure

This document is structured to provide a comprehensive overview of the WP3 components' implementation. The rest of this document is structured as follows:

Section 2 presents an overview of the WP3 Architecture. It offers a concise look at the various components of WP3, focusing on the technical details of each component (e.g. which programming language it is written in) and the communication between them (e.g. used protocols such as gRPC).

Sections 3 to 9 contain the main contribution of the document. Each section delves into each component's implementation as well as its delivery and usage. The implementation covers the functional description (overall purpose of the component), technical description, prototype architecture (diagram and description), (sub) components description and technical specifications. Delivery and usage comprise package information (structure of the delivered package such as folders and files), installation instructions, instructions for use, licensing information and information about the download of the software.

Finally, Section 10 reports the conclusions.

In the *APPENDIX A: Examination of Graph DB Engines*, further information on the examination of Graph DB Engines is provided.

2 Evidence assessment and integration components in the EMERALD architecture

Before we look at the connection of the WP3 components to each other and to other EMERALD components, we give a brief description of each component (for more information, see D3.1 [2]):

- ***Clouditor-Orchestrator*** (described in Section 3) is responsible for managing the overall certification workflow and coordinating interactions between various components within the EMERALD framework.
- ***Clouditor-Assessment*** (described in Section 4) assesses evidence based on predefined metrics and generates assessment results that inform compliance decisions.
- ***Clouditor-Evidence Store*** (described in Section 5) securely stores and manages evidence collected from various sources, organizing it within a graph-based database.
- ***Mapping Assistant for Regulations with Intelligence* (*MARI*)** (described in Section 6) maps security controls of different security schemes to suitable metrics and controls from different schemes, optimizing the compliance assessment process.
- ***Clouditor-Evaluation*** (described in Section 7) aggregates assessment results to determine the compliance of cloud services with specific security controls.
- ***Repository of Controls and Metrics* (*RCM*)** (described in Section 8) serves as a smart catalogue of metrics and controls, facilitating the reuse and composition of elements within the certification process.
- ***Trustworthiness System* (*TWS*)** (described in section 9) enhances the integrity and transparency of the certification process by ensuring the trustworthiness of evidence and assessment results through blockchain technology.

Figure 1 illustrates the connections of WP3 components, including the protocols used for transporting information as well as the programming languages for the respective (sub) components. Besides, the figure represents the relationship of the *Evidence assessment and integration components* developed in the context of WP3 and the other components in EMERALD.

The components *Clouditor-Orchestrator*, *Clouditor-Assessment*, *Clouditor-Evidence Store* and *Clouditor-Evaluation* are part of the *Clouditor* toolbox[1]. Serving as a reference implementation for research, the *Clouditor* is developed to provide continuous cloud assurance, i.e. to evaluate cloud-based applications. For this purpose, it leverages the usage of semantic evidence, paving the way towards "continuous certification which is scheme- and vendor-independent" [8]. All *Clouditor* components are developed with the programming language Go[2]. Go is a statically typed, compiled programming language designed for simplicity, efficiency, and reliability. It features built-in support for concurrent programming, making it highly suitable for developing scalable and high-performance applications. We use Go in the *Clouditor* because of its efficient concurrency model, ease of deployment, and strong performance characteristics. Additionally, its robust standard library and tools for building services align well with the microservice architecture of the EMERALD framework. The *Clouditor* is designed and implemented in such a way that it can be easily imported as a library for further development, e.g. in EMERALD.

[1] https://github.com/clouditor/clouditor

[2] https://go.dev/

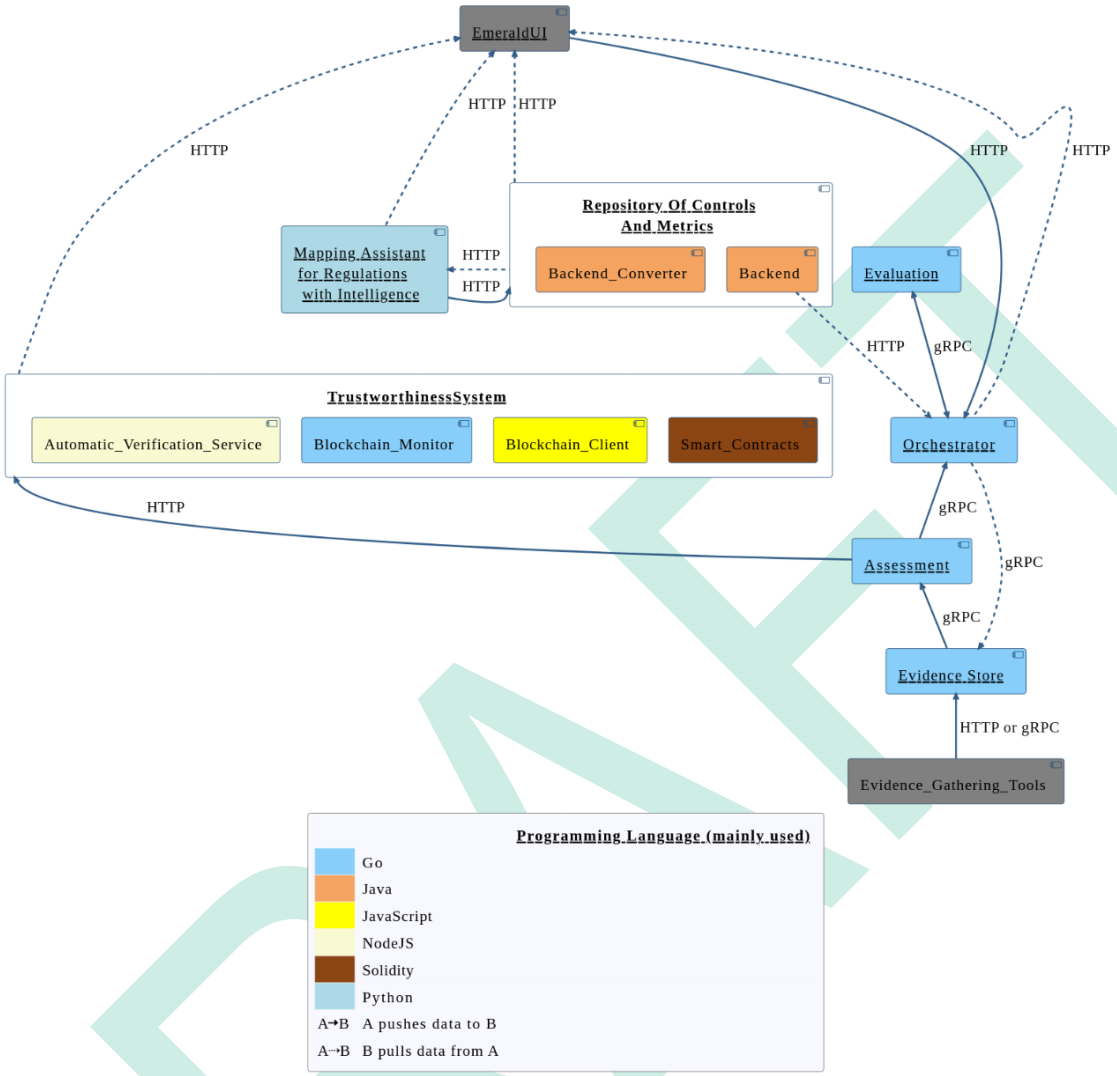

Figure 1. Overview of the EMERALD components with special focus on Evidence Assessment and Certification components

3 Clouditor-Orchestrator

The *Clouditor-Orchestrator* (in the following *Orchestrator*) is the central component orchestrating the certification process and connecting multiple components of the EMERALD framework. This component also handles the final certification decision, determining whether a cloud service is compliant with a given security scheme or not.

3.1 Implementation

The *Orchestrator* component is based on the respective microservice of *Clouditor* and was already used in MEDINA[3] to manage and provide an interface for its components [9]. It will be further developed by leveraging the functionality of the *Life-Cycle Manager* component in MEDINA to provide the final certificate decision [10].

3.1.1 Functional description

The *Orchestrator* is responsible for orchestrating the certification process and connecting various components. Its primary role is to manage the workflow of the certification process, ensuring that all necessary steps are executed in the correct sequence. The *Orchestrator* also interacts with other components, such as the *Clouditor-Assessment* and *Clouditor-Evidence Store*, to gather and process evidence required for certification decisions.

The motivation behind the *Orchestrator* is to provide a unified and efficient framework for managing the certification process of cloud services. By leveraging the functionalities of the *Life-Cycle Manager* component used in MEDINA, the *Orchestrator* aims to automate and streamline the certification workflow, reducing the complexity and time required for certification. This ensures that cloud services can be certified in a timely and consistent manner.

The *Orchestrator* introduces several key innovations to the certification process within the EMERALD framework:

1. Integration with Multiple Components: The *Orchestrator* effectively connects various components, such as the *Clouditor-Assessment*, *Clouditor-Evidence Store*, and *Mapping Assistant for Regulations with Intelligence* (MARI), facilitating seamless data flow and interaction.
2. Automated Certification Decisions: By leveraging the functionality of the *Life-Cycle Manager* component from MEDINA, the *Orchestrator* automates the final certification decision process, determining whether a cloud service is compliant with a security catalogue.
3. Enhanced Workflow Management: The *Orchestrator* manages the entire certification workflow, ensuring that all necessary steps are executed in the correct sequence and that evidence is collected and processed efficiently.
4. Scalability and Flexibility: Built using a microservice architecture, the *Orchestrator* can be easily scaled and adapted to handle varying workloads and integrate new components as needed.

Table 1 outlines the functional requirements proposed by the current version of the *Orchestrator*, as documented in D3.1 [2], and updates the status of their implementation in the current prototype in M12.

[3] https://medina-project.eu/

Table 1. Orchestrator functional requirements.

Req. ID	Description	Priority	Milestone	Progress
ORCH.01	**Final certificate decision:** Since we do not have a dedicated life-cycle manager component in EMERALD, the Orchestrator must take care of the final certificate decision. The decision is based on the input of the Evaluation component providing the Orchestrator with an evaluation result for each control.	Must	MS5 (M24)	10%
ORCH.02	**REST API Gateway for UI:** The Orchestrator should provide a REST API gateway for the UI that serves a central API endpoint for all information needed from the Orchestrator, Assessment, Evaluation and other Clouditor components.	Must	MS2 (M12)	100%
ORCH.03	**Role Based Access Control:** Since the UI wants to selectively disclose information to users and/or roles, we need a RBAC mechanism in our API endpoints, mainly in the Orchestrator.	Must	MS5 (M24)	25%
ORCH.04	**Manage Tools via API**: We need to manage external tools, such as evidence extractors in the Orchestrator.	Must	MS3 (M18)	0%
ORCH.05	**Provide an API for audit workflow:** We want to assign people to controls within an audit instance that have a particular task.	Must	MS6 (M30)	0%

3.1.1.1 Fitting into overall EMERALD Architecture

The *Orchestrator* plays a crucial role within the EMERALD framework by coordinating the interactions between various components involved in the certification process. It acts as the central hub, managing the flow of information and ensuring that each component performs its designated tasks in a cohesive manner.

The *Orchestrator* interfaces with the *Clouditor-Assessment* component to receive assessment results and with the *Clouditor-Evidence Store* to retrieve necessary evidence. It also interacts with the *Repository of Controls and Metrics* (RCM) to access relevant controls and metrics. Additionally, the *Orchestrator* communicates with the *Trustworthiness System* (TWS) to ensure the integrity and trustworthiness of the evidence and assessment results. To evaluate if a certification target is compliant with certain controls of a security scheme, the *Orchestrator* interacts with the *Clouditor-Evaluation*. For this purpose it sends the respective assessment results to the *Evaluation* component and in turn receives the evaluation results for the controls. Based on these evaluation results, the *Orchestrator* then makes the final certificate decision for a given certification target. Finally, the *Orchestrator* communicates with *AMOE* to send the target values for the metric configuration.

As a central component, the *Orchestrator* is connected to nearly all other components within the EMERALD framework. This extensive connectivity ensures that it can effectively manage and coordinate the certification process, as illustrated in Figure 2.

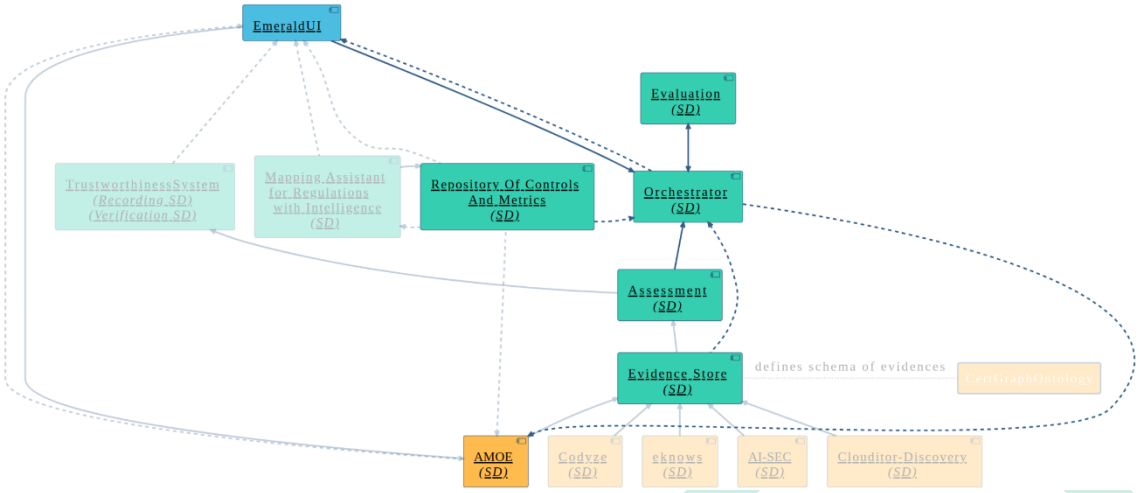

Figure 2. Role of the Orchestrator in the EMERALD framework

3.1.2 Technical description

The technical description of the *Orchestrator* provides an in-depth look at its architecture, components, and technical specifications. This section outlines how the *Orchestrator* is structured, as well as the specific technologies and methods used in its implementation. The following subsections detail the prototype architecture, components, and technical specifications of the *Orchestrator*.

3.1.2.1 Prototype architecture

The *Orchestrator* is designed as a microservice within the *Clouditor* tool, leveraging a modular architecture to ensure scalability and flexibility. The architecture of the *Orchestrator* includes several key elements:

- **API Gateway:** The *Orchestrator* provides REST and gRPC endpoints for interaction with other components and external systems. This gateway facilitates the communication between the *Orchestrator* and other parts of the EMERALD framework, enabling efficient data exchange.
- **Orchestration Module**: This module is responsible for managing the overall certification workflow, ensuring that each step in the process is executed in the correct sequence. It coordinates with other components such as the *Clouditor-Assessment* and *Clouditor-Evidence Store* to gather and process the necessary evidence.
- **Compliance Module**: This module evaluates the assessment results against predefined metrics and controls to make certification decisions. It ensures that all necessary evidence has been collected and assessed before making a final certification decision.

While the *Orchestrator* is not further divided into subcomponents, it is using different files representing different functionalities. Figure 3 illustrates the high-level architecture of the *Orchestrator*, including the REST-Gateway and Authorization which are available to every Clouditor service as well as the internal components (the different functionalities starting with *Assessment Results* from the top to the *Compliance* module at the bottom).

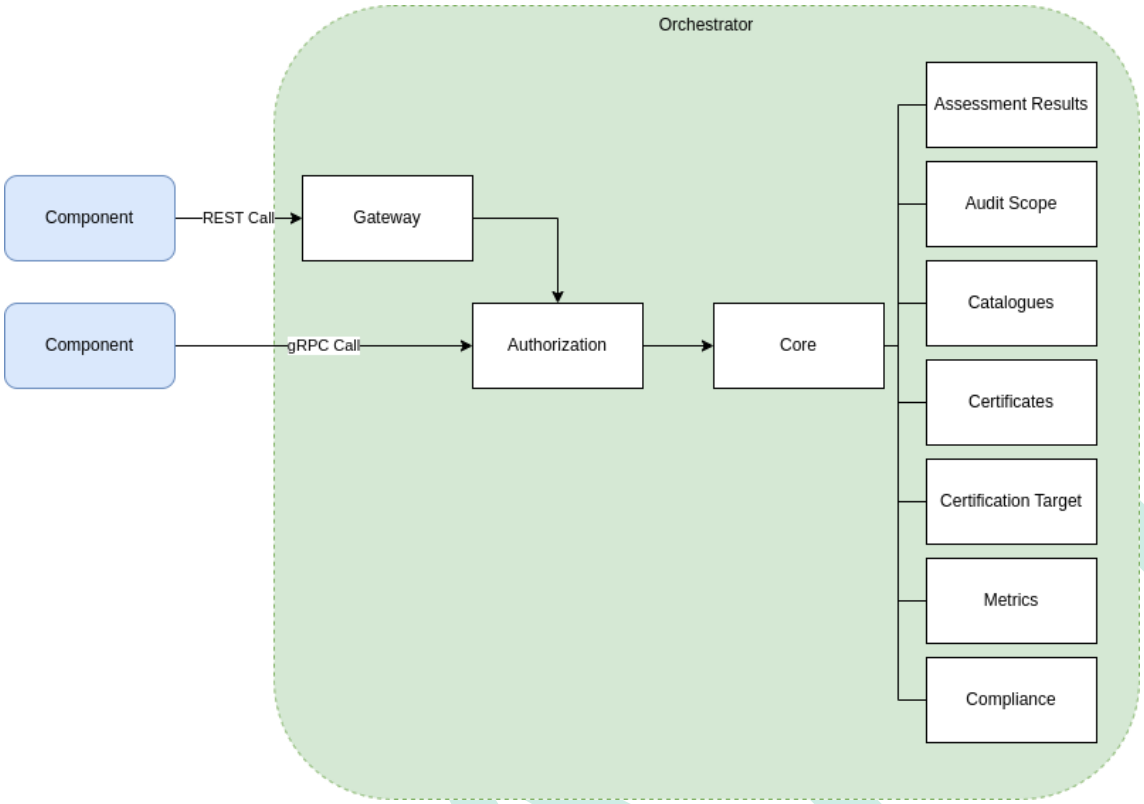

Figure 3. The prototype architecture of the Orchestrator

3.1.2.1.1 Components description

The architecture of the *Orchestrator* includes several key elements and functionalities:

- Clouditor's REST Gateway and Authorization: The *Orchestrator* uses Clouditor's REST Gateway to allow REST calls from components not able to communicate via gRPC. This ensures compatibility and flexibility in communication protocols. Additionally, the Orchestrator leverages Clouditor's Authorization implementation to provide state-of-the-art authentication mechanisms, such as OAuth2, which is used in EMERALD.

- Functionalities:
    a. **Assessment Results**: Responsible for handling assessment results, e.g. coming from the *Assessment* component. It provides CRUD (Create, Read, Update, Delete) operations for assessment results.
    b. **Audit Scope**: Manages CRUD operations for audit scopes (previously known as target of evaluation), which define a (part of a) cloud service and the (parts of) the respective certification schemes to check against.
    c. **Catalogues:** Handles general CRUD operations for catalogues as well as more specific ones, such as retrieving a specific control of a catalogue. In the context of EMERALD, this functionality is mainly used to get catalogues from the Repository of Controls and Metrics (RCM).
    d. **Certificates**: Manages CRUD operations for certificates against which a certification target is checked, including state history. It also offers an operation to list all current certifications without state history.
    e. **Certification Target**: Manages CRUD operations for certification targets (previously known as cloud services).

f. **Metrics:** Manages CRUD operations for metrics (and metric configurations), such as loading metrics locally or via network. In the context of EMERALD, metrics are received from the RCM.
g. **Compliance:** Based on evaluation results provided by the *Clouditor-Evaluation*, this module provides the final certificate decision.
h. **Orchestrator:** The main class that can instantiate an *Orchestrator* instance capable of running the operations described above.

3.1.2.2 Technical specifications

The *Clouditor-Orchestrator* is implemented using Go, providing efficient concurrency support and ease of deployment in microservice architectures. The main communication protocols used are REST and gRPC, ensuring high-performance interaction with other components.

- Programming Language: Go
- Communication Protocols: REST API and gRPC (including Protobuf)
- Postgres database for storing Assessment Results and Evaluation Results
- Security: OAuth and Role-Based Access

3.2 Delivery and usage

This section describes the information needed for the installation and use of the *Orchestrator.* Besides, it also details the licensing information and related packages and repositories.

3.2.1 Package information

Table 2 shows the *Orchestrator*-relevant package structure in the *Clouditor* repository and Table 3 shows the package structure in the EMERALD framework, where the *Orchestrator* parts of the *Clouditor* tool are used as dependencies.

Table 2. Package structure of Clouditor with Orchestrator-relevant parts

Folder	Description
`api/orchestrator/`	This folder contains code needed for the communication with this component. It mainly consists of auto-generated Protobuf and gRPC files.
`cli/commands/service/orchestrator`	This folder contains the Clouditor CLI based source code files
`cmd/orchestrator/`	This folder contains the main file.
`openapi/orchestrator/`	This folder contains the auto-generated OpenAPI files for the *Orchestrator*.
`rest/`	This folder contains the REST gateway implementation.
`service/orchestrator/`	This folder contains the source code for the Orchestrator microservice.

Table 3. Package structure of the Orchestrator used in EMERALD

Folder	Description
`modules/`	This folder contains the source code for the EMERALD-specific parts of the *Orchestrator*, e.g. the *Compliance* module for making certificate decisions.

`cmd/orchestrator`	This folder contains the main file.
`./ (Root directory)`	Beside the folders mentioned above, the root directory also contains a workflow file needed for continuous integration and deployment (`.gitlab-ci.yml`), *a README* file, a Go file for launching the EMERALD Orchestrator and Go module files for handling dependencies (`go.mod` and `go.sum`).

3.2.2 Installation

In EMERALD, we use GitLab's CI/CD pipeline for continuous integration and deployment. For this purpose, there is a workflow file at each components root level (`.gitlab-ci.yml`).

For running the *Orchestrator* locally, there is a docker file ("*Dockerfile*") located at the components root level. For building and running the *Orchestrator*, use the following commands:

```
docker build -t clouditor-orchestrator .
docker run -d -p 8080:8080 clouditor-orchestrator
```

3.2.3 Instructions for use

Within the EMERALD project, the *EMERALD UI* is used to access and manage the workflow in the framework. In the case of the *Orchestrator*, the UI interacts with it by using the components API endpoints, e.g. to list certification targets.

Currently, the EMERALD UI is work in progress, but some clickable mock ups have been designed in D4.3 [11], e.g. the audit scope overview in Figure 4.

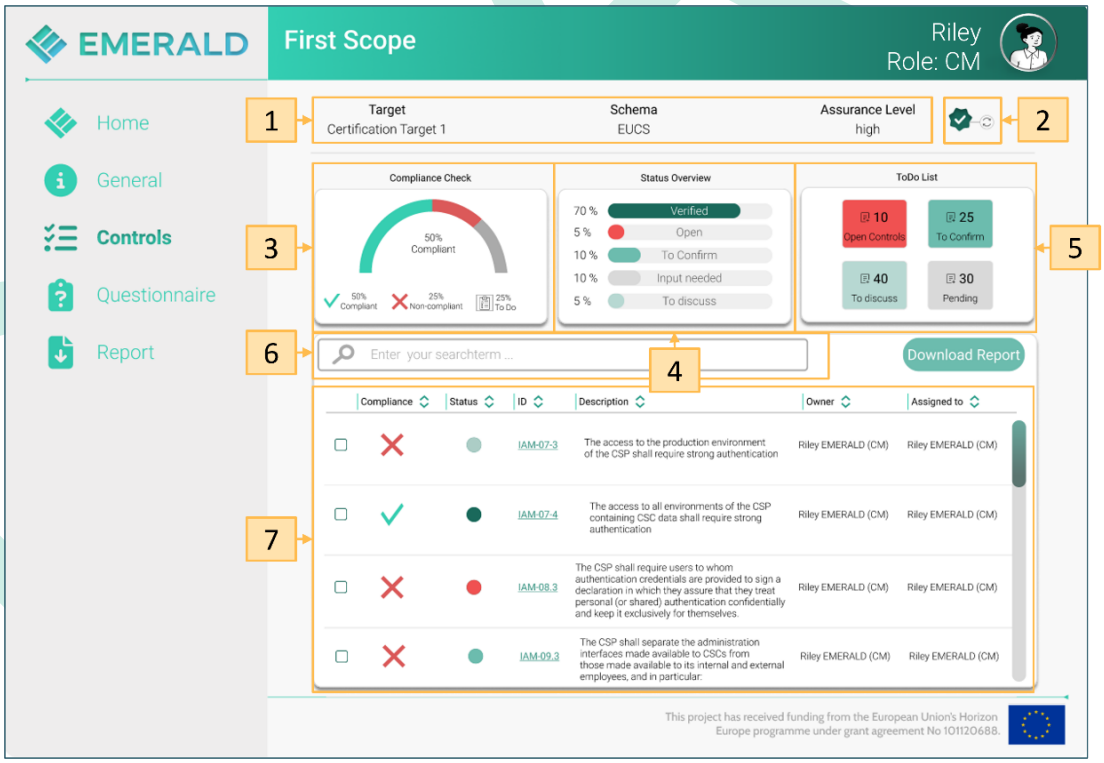

Figure 4. View of an audit scope from D4.3 [11]

3.2.4 Licensing information

The *Clouditor-Orchestrator* is offered under Apache 2.0 license. The license files and more detailed information can be found in the GitLab repository.

3.2.5 Download

The Clouditor source code can be found in the Clouditor GitHub repository[4]. The development of the Orchestrator can be found in the public EMERALD GitLab repository[5].

[4] https://github.com/clouditor/clouditor
[5] https://git.code.tecnalia.com/emerald/public/components/orchestrator

4 Clouditor-Assessment

The *Clouditor-Assessment* (in the following *Assessment*) component is responsible for evaluating evidence based on predefined metrics within the EMERALD framework. In the following, we provide information about the implementation as well as the delivery and usage of this component.

4.1 Implementation

The *Assessment* component is based on the respective microservice of *Clouditor* and was already used in MEDINA [9]. It will be further developed in EMERALD to handle multiple pieces of evidence that reflect resources on different layers.

4.1.1 Functional description

The *Assessment* is responsible for assessing evidence based on predefined metrics within the EMERALD framework. It processes the collected evidence to determine compliance with specific security requirements. The assessment results generated by this component are inspired by but decoupled from the actual controls of security catalogues. These results are then used by the *Clouditor-Evaluation* component (described in Section 7) to determine compliance with the relevant controls.

The *Assessment* evaluates evidence collected from various sources, ensuring that the evidence meets the predefined metrics necessary for compliance. This component operates within the broader context of the EMERALD framework, where it plays a critical role in the certification process.

The motivation behind the *Assessment* is to automate the evaluation of compliance, reducing manual effort and increasing the accuracy of assessments. By providing a systematic approach to evidence assessment, it ensures consistent and reliable results.

The main innovations of the *Assessment* component include:

- Automated Assessment: Utilizes predefined metrics to automatically evaluate evidence, streamlining the certification process.
- Scalability: Capable of handling multiple pieces of evidence that reflect resources on different layers, providing a comprehensive assessment.
- Interoperability: Integrates seamlessly with other components of the EMERALD framework, ensuring efficient data exchange and processing.

Table 4 outlines the functional requirements satisfied by the current version of the *Assessment*, as documented in D3.1 [2], and updates the status of their implementation in the current prototype in M12.

Table 4. Assessment functional requirements

Req. ID	Description	Priority	Milestone	Progress
ASSESS.01	**Assessment based on evidence:** The assessment should assess evidence based on the knowledge graph.	Must	MS6 (M30)	15%
ASSESS.02	**Assessment rules for 80% of the defined metrics:** Assessment rules must exist for 80% of the metrics defined in KP4.1.	Must	MS6 (M30)	15%
ASSESS.03	**Display cause of assessment result:** We want to know why an assessment result fails or passes.	Must	MS6 (M30)	0%

4.1.1.1 Fitting into overall EMERALD Architecture

The connection of the *Assessment* to other components in the EMERALD framework can be seen in Figure 5. Initially, the *Assessment* receives the metrics which include the rules for assessing evidence. These metrics originate from the *RCM* but are transferred via the central *Orchestration* component.

Evidence collected from the various collectors is sent to and stored in the *Evidence Store*. The *Evidence Store* then forwards this evidence to the *Assessment* component. Here the evidence (single ones or combined around different layers) is assessed using the metrics the *Assessment* received in the beginning.

Then both the evidence as well as the assessment results are sent to the *TWS* to ensure integrity and, therefore, enhance the trustworthiness of the whole process. The assessment results are also sent to the *Orchestrator* which first stores them in a database. The *Orchestrator* can then use the respective assessment results to evaluate these (using the *Evaluation* component) and, in the end, to make the final certificate decision.

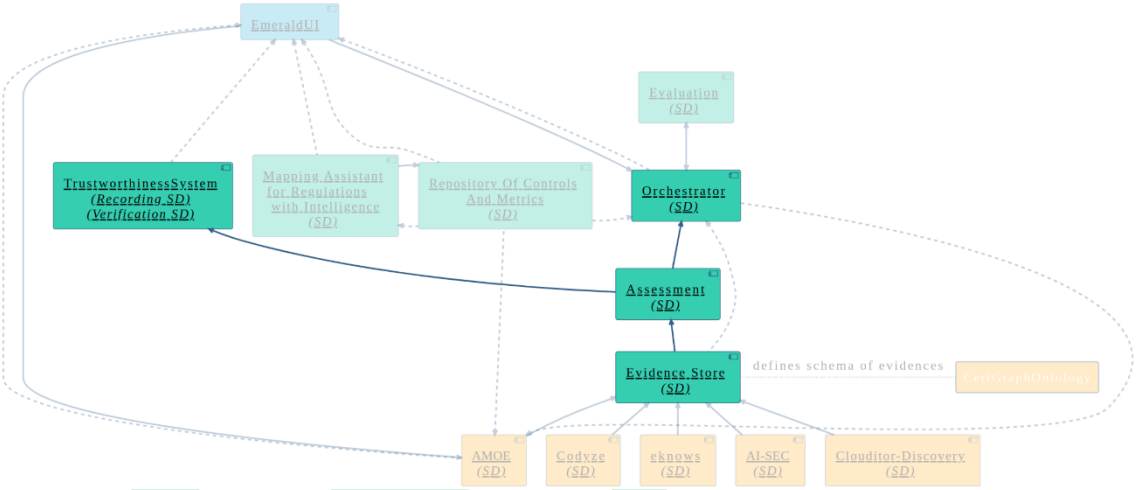

Figure 5. Role of the Assessment in the EMERALD framework

4.1.2 Technical description

The technical description of the *Assessment* provides an in-depth look at its architecture, components, and technical specifications. This section outlines the structure of the *Assessment* component, as well as the specific technologies and methods used in its implementation. The following subsections detail the prototype architecture, components, and technical specifications.

4.1.2.1 Prototype architecture

The architecture for the *Assessment* component is shown in Figure 6. It comprises the core component (`assessment.go` in the code structure) which is mainly responsible for providing and handling API requests as well as for instantiating an assessment component. For assessing incoming evidence, the *Assessment* leverages the Clouditor-internal library *Policies* which applies predefined rules to get an assessment result.

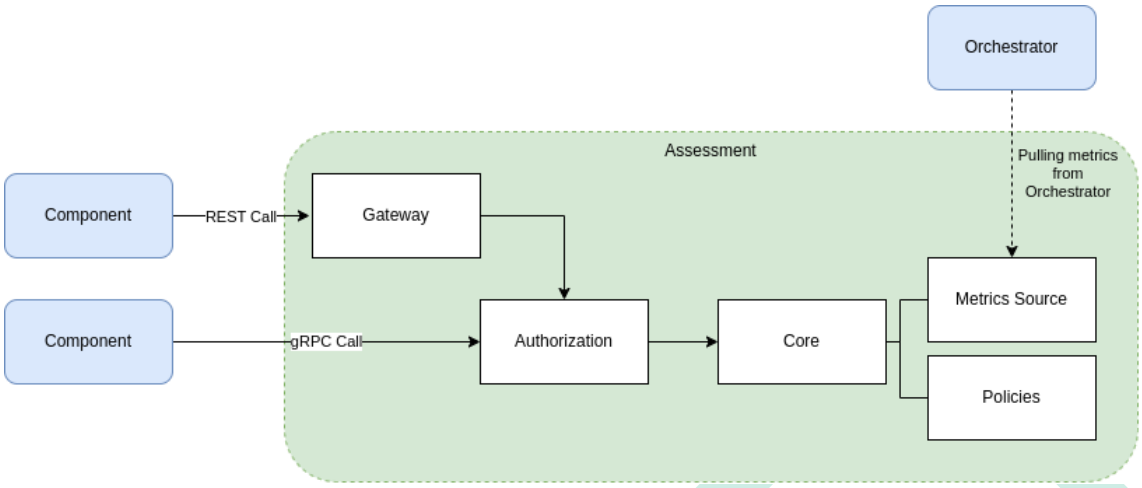

Figure 6. The prototype architecture of the Assessment

4.1.2.1.1 Components description

The *Assessment* is mainly composed by the following components:

- Clouditor's REST Gateway and Authorization: The *Assessment* uses Clouditor's REST Gateway to allow REST calls for components not able to communicate via gRPC. This ensures compatibility and flexibility in communication protocols. Additionally, the *Orchestrator* leverages Clouditor's Authorization implementation to provide state-of-the-art authentication mechanisms, such as OAuth2, which is used in EMERALD.

- The Core (`assessment.go`*):*
  - Function *NewService* for instantiating a new assessment.
  - Metric-corresponding methods (e.g. *MetricConfiguration*) that implements the *MetricsSource* interface to define where metric information is coming from. In the case of EMERALD, information about metrics is obtained from the *Orchestrator* (but originated from the *RCM*).
  - API-corresponding functions that implement the assessment service interface (for all relevant API endpoints see below), e.g. *AssessEvidences* that opens a stream to receive multiple instances in one connection. When assessment results are created, they are forwarded to the *Orchestrator* component and, together with evidence to the *TWS* to ensure that the data is not tampered with in the future.

- *Policies*:
  - Used by the *Core* (in *AssessEvidence* and *AssessEvidence*) to assess incoming evidence with pre-defined metrics (i.e. rules). The assessment initially loads the metrics from the *Orchestrator*.
  - Currently, we use the OPA policy engine to assess the evidence via the metrics that are written in the OPA policy language Rego. Since we are planning to move to a graph-based evidence structure (see *Evidence Store* in Section 5), we will also consider alternative approaches that may fit better the requirement to assess multiple evidence located in different layers (layers are, e.g., code, configuration and documents). Note that the evidence structure is defined by the ontology [12]. The Rego rules must be aligned with the ontology to work correctly. A concise example is given below.

In the following, all API endpoints are listed accompanied by a short explanation (all API endpoints are defined in the Clouditor repository[6]):

- *AssessEvidence*: Assesses the pieces of evidence included in the AssessEvidenceRequest. This endpoint is exposed via gRPC as well as REST.
- *AssessEvidences*: Assesses stream of evidence by opening a continued connection with the component that calls this endpoint. Because the connecting component does not have to open a new connection for each single evidence, multiple evidence can be assessed much faster. In particular. in the case of EMERALD, this can result in a major boost because complex distributed systems can lead to thousands of pieces of evidence that have to be assessed (e.g. when checking configurations of cloud systems). This endpoint is only served via gRPC and not at REST.

Example for the usage of Rego

Generally, using Rego comprises an input (here: evidence), a rule (policy) that is applied on the input, and a configuration to set custom values for the rule[7]. In the following, we show a simple example of MEDINA, for further reference see D3.5 of MEDINA [13].

Figure 7 depicts a simple Rego policy which checks the encryption algorithm used in an at-rest encryption (e.g. a block storage in a cloud). Figure 8 shows the configuration stating that the encryption algorithm should be at least 256 bits. Figure 9 shows an exemplary snippet of an evidence (configuration of a block storage).

```
default compliant = false

compliant {
    data.operator == ">="
    input.atRestEncryption.algorithm >= data.target_value
}

compliant {
    data.operator == "=="
    input.atRestEncryption.algorithm == data.target_value
```

Figure 7. Example of a Rego policy [13]

```
{
    "operator": ">=",
    "target_value": 256
}
```

Figure 8. Example of a Rego configuration [13]

[6] https://github.com/clouditor/clouditor/blob/main/api/assessment/assessment.proto
[7] https://www.openpolicyagent.org/docs/latest/

```
{
    "atRestEncryption": {
        "algorithm": 256,
        "enabled": true,
        "keyManager": "Microsoft.Storage"
    },
    "name": "storage12"
}
```


Figure 9. Example of an evidence [13]

In this example, the block storage has enabled encryption at rest and set the algorithm to have 256 bytes. Because the Rego policy and the configuration have defined that the evidence has to have at least 256 bytes, the assessment succeeds and outputs that this evidence is compliant (compliant is set to true).

4.1.2.2 Technical specifications

The *Clouditor-Assessment* is implemented using Go, providing efficient concurrency support and ease of deployment in microservice architectures. The main communication protocols used are REST and gRPC, ensuring high-performance interaction with other components.

- Programming Language: Go
- Communication Protocols: REST API and gRPC (including Protobuf)
- Security: OAuth and Role-Based Access

4.2 Delivery and usage

This section describes the information needed for the installation and use of the *Assessment.* Besides, it also details the licensing information and related packages and repositories.

4.2.1 Package information

Table 5 shows the *Assessment*-relevant package structure in the *Clouditor* repository and Table 6 shows the package structure in the EMERALD framework, where the *Assessment* parts of *Clouditor* are used as dependencies.

Table 5. Assessment package structure

Folder	Description
`api/assessment/`	This folder contains code needed for the communication with this component. It mainly consists of auto-generated Protobuf and gRPC files.
`api/ontology`	This folder contains the ontology objects (evidence format) defined in certification graph in WP2.
`cli/commands/service/assessment`	This folder contains the Clouditor CLI based source code files
`cmd/assessment/`	This folder contains the main file.
`openapi/assessment/`	This folder contains the auto-generated OpenAPI files for the Assessment.
`policies/`	This folder contains the Go implementation for using the OPA engine and, therefore, Rego rules. It will also contain the Rego policy files per metric.

`rest/`	This folder contains the REST gateway implementation.
`service/assessment/`	This folder contains the source code for the Assessment microservice.

Table 6. Assessment package structure in the EMERALD framework

Folder	Description
`cmd/emerald-assessment`	This folder contains the main file.
`./ (Root directory)`	Beside the folders mentioned above, the root directory also contains a workflow file needed for continuous integration and deployment (`.gitlab-ci.yml`), *a README* file, a Go file for launching the EMERALD Assessment and Go module files for handling dependencies (`go.mod` and `go.sum`).

4.2.2 Installation

In EMERALD, we use GitLab's CI/CD pipeline for continuous integration and deployment. For this purpose, there is workflow file at each components root level (`.gitlab-ci.yml`).

For running the Assessment locally, there is a docker file ("*Dockerfile*") located at the components root level. For building and running the component, use the following commands:

```
docker build -t clouditor-assessment .
docker run -d -p 8080:8080 clouditor-assessment
```

4.2.3 Instructions for use

The *Assessment* is only used by internal components and is not exposed via the EMERALD UI. Information regarding the Asssessment (e.g. metrics and assessment results) are transferred via the Orchestrator.

To use the *Assessment*, run the Docker image and set the variable of the URL where the assessment results are sent to (in EMERALD, this is the URL of the *Orchestrator*). When the *Assessment* is running, it can receive metrics and metric configurations (in EMERALD this is done by the *Orchestrator* in the beginning). Now, the assessment is ready and evidence can be sent to it which will be assessed and the results sent to the URL which was set in the beginning.

4.2.4 Licensing information

The *Assessment* is offered under Apache 2.0 license. The license files and more detailed information can be found in the GitLab repository.

4.2.5 Download

The source code for the Assessment in the Clouditor toolbox can be found in the Clouditor GitHub repository[8]. The implementation of the Assessment component in EMERALD can be found in the public EMERALD GitLab repository[9].

[8] https://github.com/clouditor/clouditor
[9] https://git.code.tecnalia.com/emerald/public/components/assessment

5 Clouditor-Evidence Store

The *Clouditor-Evidence Store* (in the following *Evidence Store*) is responsible for receiving evidence from the evidence collectors (WP2), storing it in a graph-based representation (certification graph [12]) and forwarding it for further processing (e.g. assessment, see Section 4). In the following, we provide information about the implementation as well as the delivery and usage of the component.

5.1 Implementation

The *Evidence Store* component is a crucial part of the EMERALD framework, designed to efficiently store and manage evidence collected from various sources. It is based on the respective microservice of *Clouditor* and was already used in MEDINA [9]. It is planned to further develop it to utilize a graph-based database for storing the certification graph, allowing for efficient organization, retrieval, and updating of evidence, and enabling sophisticated assessment of evidence distributed across various layers of a cloud service.

5.1.1 Functional description

Although in MEDINA, the *Evidence Store* was already storing evidence from different layers, it was not designed to "connect" evidence from different layers (e.g. application code with respective infrastructure configurations in the cloud). Therefore, we are planning to move towards a certification graph, allowing more complex assessments on evidence (see Section 4) to reflect the multi-dimensional needs of controls in a security catalogue.

Table 7 outlines the functional requirements satisfied by the current version of the *Evidence Store*, as documented in D3.1 [2], and updates the status of their implementation in the current prototype in M12.

Table 7. Evidence Store functional requirements

Req. ID	Description	Priority	Milestone	Progress
ESTORE.01	**Storage of evidence as ontology entities in graph database:** The Evidence Store must store the evidence according to the schema defined by the knowledge graph. The preferred way to store this information is a graph database.	Must	MS3 (M18)	30%
ESTORE.02	**Allow Interaction with Third-Party Tools:** The Evidence store should be allowed to accept evidence from third-party tools, e.g., using a REST API. The evidence needs to be in the ontology format. Therefore, information about the ontology and data models must be available.	Must	MS8 (M34)	25%

5.1.1.1 Fitting into overall EMERALD Architecture

The *Evidence Store* is the central link between the Evidence Collectors (developed in WP2 [14], [15], [16], [17]) and the WP3 components for assessment, evaluation and the final certification decision, see Figure 10.

Different evidence collectors exist that gather evidence from different layers of the certification target and send them to the Evidence Store: *AMOE* collects document information [15], *Codyze* and *eknows* application code [14], *AI-SEC* ML systems [16], and *Clouditor-Discover* configuration information from cloud systems [17]. The *Evidence Store* stores this evidence in a database and forwards the respective evidence to the *Assessment* component for further processing.

The uniform evidence format adheres to the ontology defined in WP2 (see D2.1 [12]) and is therefore the basis for the "decoupling" WP2 and WP3 components, regardless of the layer in which the evidence is located.

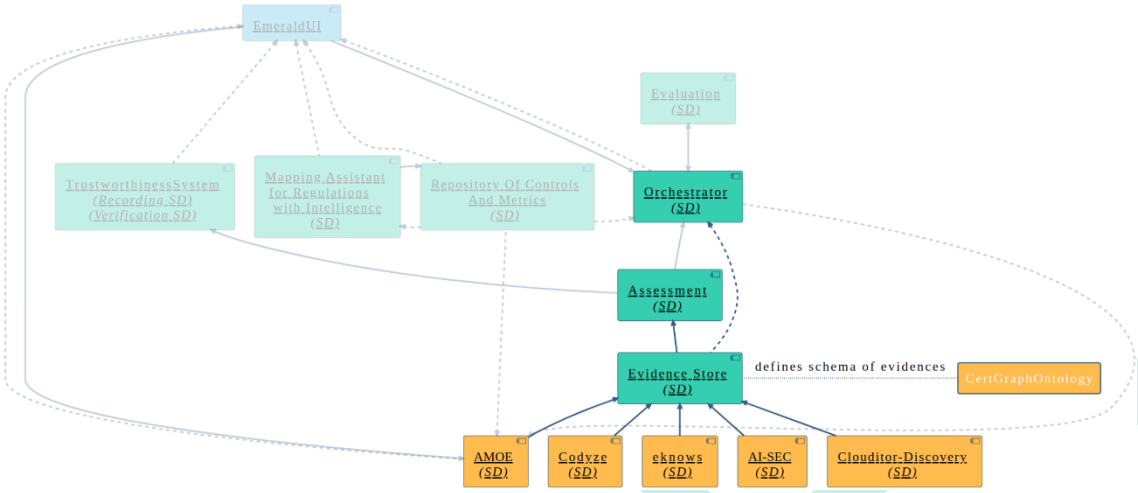

Figure 10. Role of the Evidence Store in the EMERALD framework

5.1.2 Technical description

The technical description of the *Evidence Store* provides an in-depth look at its architecture, components, and technical specifications. This section outlines the structure of the *Evidence Store*, as well as the specific technologies and methods used in its implementation. The following subsections detail the prototype architecture, components, and technical specifications.

Because the shift towards a certification graph is still work in progress, we present the previous approach in MEDINA as well as the two approaches we are testing currently: using a dedicated graph database vs. using regular a Postgres instance to represent a graph database for our needs in EMERALD.

5.1.2.1 Prototype architecture

The architecture of the *Evidence Store* is shown in Figure 11. It comprises the *Clouditor*'s REST gateway and authorization mechanism as well as two specific components. The first component is the *Core* (`evidence_store.go` in the code structure) which is mainly responsible for providing and handling API requests as well as for instantiating an evidence store component. The second component is *Persistence*, which is handling the storage of the Evidence, i.e. the intended graph database in the context of EMERALD.

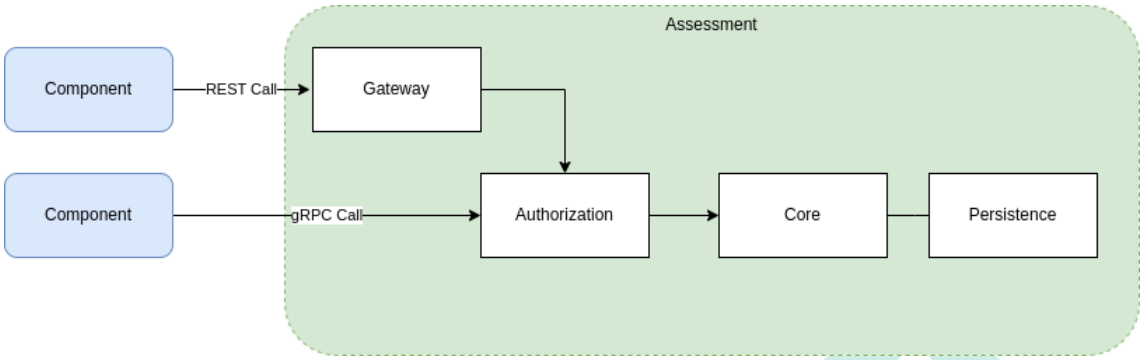

Figure 11. The prototype architecture of the Evidence Store

5.1.2.1.1 Components description

The architecture of the *Evidence Store* includes several key elements and functionalities:

- Clouditor's REST Gateway and Authorization: The *Evidence Store* uses Clouditor's REST Gateway to allow REST calls for components not able to communicate via gRPC. This ensures compatibility and flexibility in communication protocols. Additionally, the Orchestrator leverages Clouditor's Authorization implementation to provide state-of-the-art authentication mechanisms, such as OAuth2, which is used in EMERALD.

- The *Core* (`evidence_store.go`):
  - Function *NewService* for instantiating a new *Evidence Store* (can potentially be scaled to several instantiations which, e.g., access the same database)
  - API-corresponding functions that implement the *Evidence Store* Service interface (for all relevant API endpoints see below). The interface comprises CRUD functionalities, e.g. *StoreEvidence* to let an evidence collector of WP2 store evidence. For these purposes, the implemented methods access the persistence component which is described next.

- *Persistence*
  - The persistence component is implementing the persistence interface defined in Clouditor which requires to implement the following methods: *Create*, *Save*, *Update*, *Get*, *List*, *Count*, *Delete* and *Raw* (for raw DB-specific statements). The idea behind is to allow for the usage of different database (schemes) depending on the current needs. In the context of EMERALD, e.g., we want to move to a graph-based representation which reflects the evidence that is gathered on different layers on a certification target. One requirement is that it is open-source and has a suitable license, e.g. Apache 2, which allows the free use of this database. The second requirement is that it must have a native Go client or provide other mechanisms for seamless integration with Go.
  - As stated earlier this is work in progress. Therefore, we provide in the following brief descriptions of the database approach that is currently in place and two approaches we try to implement for efficient storage and usage of multi-layer evidence:
    - Current state of the art [18]: Offering an in-memory storage as well as a PostgreSQL connection.
    - Dedicated graph-based database: We examined different databases based on the requirements outlined above (see also *APPENDIX A: Examination of Graph DB* Engines for a detailed list) and decided to proceed with *Dgraph*[10],

[10] https://dgraph.io/

which we are currently testing. Dgraph is an open-source, distributed graph database that provides high performance and horizontal scalability. It is designed to handle complex queries and large-scale data with ease, making it well-suited for our needs within the EMERALD framework.

- A standard SQL-like (non-graph) database adapted for EMERALD: The second approach we are pursuing involves using a conventional database and customizing it to meet the requirements of EMERALD. This includes defining nodes and edges in the schema to enable the connection of evidence across different layers.

All API endpoints are listed below with a brief explanation:

- *StoreEvidence*: Stores a single piece of evidence into the evidence store. The endpoint is exposed via gRPC as well as REST.
- *StoreEvidences*: Stores a stream of evidence by opening a continued connection with the component that calls this endpoint. Because the connecting component does not have to open a new connection for each single evidence, multiple evidence can be stored much faster. In particular in the case of EMERALD, this can result in a major boost because complex distributed systems can lead to thousands of pieces of evidence that have to be stored. This endpoint is only exposed via gRPC and not at REST.
- *ListEvidences*: Returns all stored evidence. The endpoint is exposed via gRPC as well as REST.
- *GetEvidence*: Returns a particular stored piece of evidence. The endpoint is exposed via gRPC as well as REST.

5.1.2.2 Technical specifications

The *Evidence Store* is implemented using Go, providing efficient concurrency support and ease of deployment in microservice architectures. The main communication protocols used are REST and gRPC, ensuring high-performance interaction with other components.

- Programming Language: Go
- Communication Protocols: REST API and gRPC (including Protobuf)
- Database: PostgreSQL or Dgraph (work in progress, see above)
- Security: OAuth and Role-Based Access

5.2 Delivery and usage

This section describes the information needed for the installation and use of the *Evidence Store.* Besides, it also details the licensing information and related packages and repositories.

5.2.1 Package information

Table 8 shows the *Evidence Store*-relevant package structure in the *Clouditor* repository and Table 9 shows the package structure in the EMERALD framework, where the *Evidence Store* parts of the Clouditor are used as dependencies.

Table 8. Evidence Store relevant package structure

Folder	Description
`api/evidence/`	This folder contains code needed for the communication with this component, including the definition of an evidence as well as the API endpoints.

	It mainly consists of auto-generated Protobuf and gRPC files.
`api/ontology`	This folder contains the ontology objects (evidence format) defined in certification graph in WP2.
`cli/commands/service/evidence`	This folder contains the Clouditor CLI based source code files
`cmd/evidence_store/`	This folder contains the main file.
`openapi/evidence/`	This folder contains the auto-generated OpenAPI files for the Evidence Store.
`rest/`	This folder contains the REST gateway implementation.
`service/evidence/`	This folder contains the source code for the Evidence Store microservice.

Table 9. Evidence Store package structure in EMERALD framework

Folder	Description
`cmd/emerald-evidence-store`	This folder contains the main file.
`./ (Root directory)`	Beside the folders mentioned above, the root directory also contains a workflow file needed for continuous integration and deployment (`.gitlab-ci.yml`), *a README* file, a Go file for launching the EMERALD *Evidence Store* and Go module files for handling dependencies (`go.mod` and `go.sum`).

5.2.2 Installation

In EMERALD, we use GitLab's CI/CD pipeline for continuous integration and deployment. For this purpose, there is workflow file at each components root level (`.gitlab-ci.yml`).

For running the *Evidence Store* locally, there is a docker file ("*Dockerfile*") located at the components root level. For building and running the component, use the following commands:

```
docker build -t clouditor-evidence-store .
docker run -d -p 8080:8080 clouditor- evidence-store
```

5.2.3 Instructions for use

The *Evidence Store* is only used by internal components and is not exposed via the *EMERALD UI*. Other evidence information, e.g. resources to show a resource graph, is received by the *EMERALD UI* from the *Orchestrator*. Figure 12 shows a digital mock-up for the resource graph, as presented in D4.3 [11].

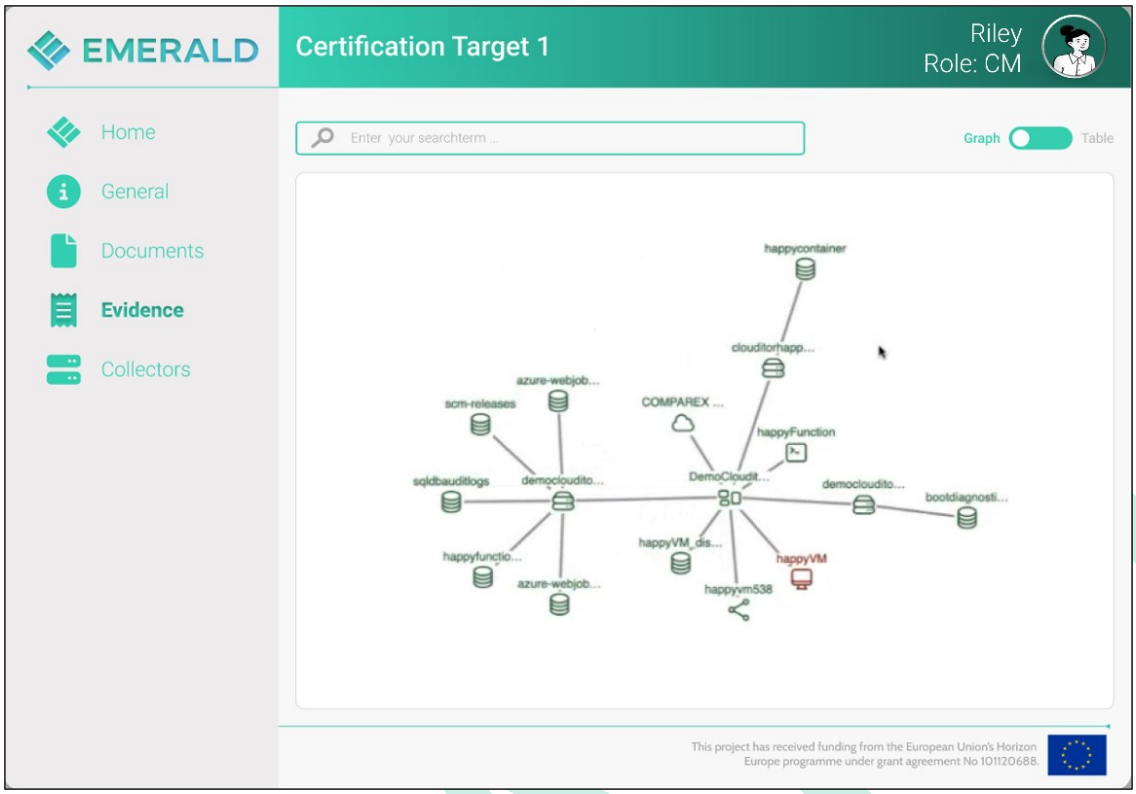

Figure 12. Digital mock-up for a resource graph (D4.3 [11])

To use the *Evidence Store*, run the Docker image and set the variable of the URL where the evidence is sent to (in EMERALD, this is the URL of the *Orchestrator*).

5.2.4 Licensing information

The *Evidence Store* is offered under Apache 2.0 license. The license files and more detailed information can be found in the GitLab repository.

5.2.5 Download

Thesource code for the Evidence Store in the Clouditor toolbox can be found in the Clouditor GitHub repository[11]. The implementation of the Evidence Store in EMERALD can be found in the public EMERALD GitLab repository[12].

[11] https://github.com/clouditor/clouditor
[12] https://git.code.tecnalia.com/emerald/public/components/evidence-store

6 Mapping Assistant for Regulations with Intelligence (MARI)

MARI is an intelligent system capable of selecting the optimal set of metrics to associate with one or more certification schemes. The metrics can be measured to evaluate the cloud system's compliance with the certification scheme. Another main feature of *MARI* is the ability to associate controls from different certification schemes.

The *MARI* component is based on the *Metric Recommender*[13] tool developed in MEDINA [19].

6.1 Implementation

The objective of *MARI* is to experiment with Deep Learning and NLP tools for automatic associations between:

- a security control and one or more security metrics;
- security controls coming from different certification schemes.

6.1.1 Functional description

Data input to *MARI* are the controls and metrics stored in the *Repository of Controls and Metrics* (described in Section 8). After selecting a set of controls and metrics, *MARI* starts the elaboration of the association either between controls or between a control and one or more metrics. The results are visualised in the *EMERALD UI*.

Table 10 shows a collection of functional requirements (from deliverable D1.3 [20]) related to the component, together with a description of how and to what extent these requirements are implemented at time of writing.

Table 10. MARI functional requirements.

Req. ID	Description	Priority	Milestone	Progress
MARI.01	**AI-based:** MARI is a tool based on state-of-the-art artificial intelligence, e.g., uses a transformer-based architecture	Must	MS6 (M30)	100%
MARI.02	**Automatic association:** MARI takes as input cloud security controls written in natural language, metrics that validate those controls, again written in natural language, and automatically returns as output the association control/metric(s) and the association control/control.	Must	MS6 (M30)	50%
MARI.03	**Performance Evaluation:** The performance of MARI should improve on the performance of the Metric Recommender of EMERALD's predecessor project, MEDINA. We can assume that we measure the performance of MARI with the same metrics used for the Metric Recommender, namely precision@k and NDCG (Normalised Discounted Cumulative Gain).	Must	MS6 (M30)	70%
MARI.04	**Usage and Visualization:** MARI should be invoked through EMERALD's built-in interface, and MARI results can be visualized through the same interface.	Must	MS6 (M30)	15%
MARI.05	**Strategies:** MARI can act according to specific strategies, such as considering only technical controls, or organizational controls, or controls of a certain category, or controls whose implementation	Must	MS6 (M30)	15%

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

	costs less in terms of human resources, etc. The strategies will be defined during the project.			

Regarding innovation, in contrast to its predecessor, the MEDINA *Metric Recommender* [19], *MARI* will support a wider range of controls from multiple certification schemes. A key enhancement is *MARI*'s ability to automatically map controls across different certification schemes, a task that previously required manual effort in the MEDINA project. Users will also have the option to define specific strategies for associating metrics with controls and for mapping controls to each other. Additionally, AI-based tools utilising transformer-based architectures are employed, enhancing both the performance and accuracy of metric-to-control associations compared to the original *Metric Recommender*.

To evaluate the performance, the metric we consider at the time of writing is the NDCG[14]. Experimenting with the approach based on sentence transformers, with 70 requirements and 162 metrics considered, we obtained NDCG@10 = 0.640, improving the performance of the MEDINA Metric Recommender by 0.146 points.

6.1.1.1 Fitting into overall EMERALD Architecture

Figure 13 shows the integration of *MARI* within the overall EMERALD architecture. *MARI* is expected to interact with the *Repository of Controls and Metrics*, which is its main source of data since it contains the controls and the metrics descriptions and the metadata. Moreover, *MARI* interacts also with the *Emerald UI*, through which it is possible to visualize the associations. It important to note that the interactions with other components have not yet been defined in detail. We therefore expect that there may be changes, and the next version of this deliverable (D3.4 [5]), due in month 24, will detail MARI's interactions with other components.

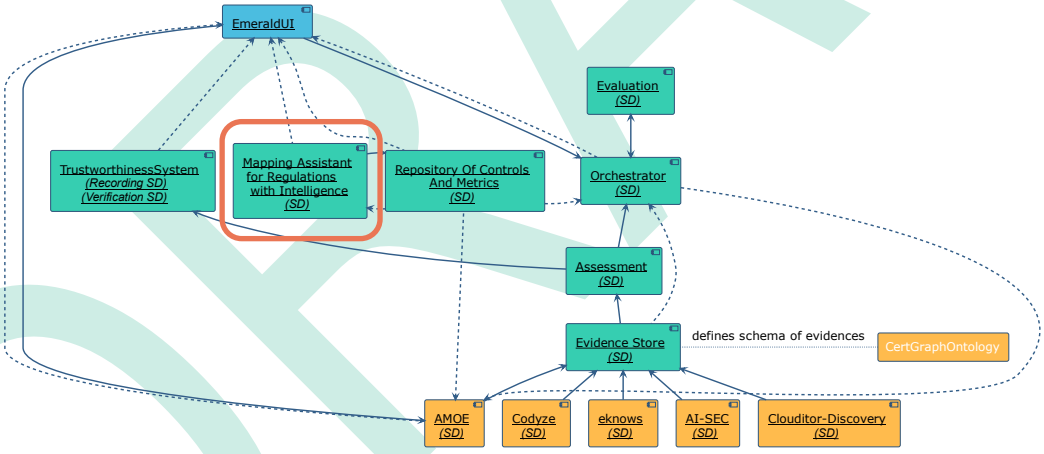

Figure 13. Fitting MARI with other components in EMERALD architecture

6.1.2 Technical description

The *MARI* component is written in Python 3.10.12 and organized as a Python notebook, an interactive computational environment that helps to manipulate and analyse data using Python. It contains all the content from a web application session, including computation inputs and outputs, mathematical functions, images, and explanatory text, making work more transparent, understandable, and reproducible.

[14] https://towardsdatascience.com/evaluation-metrics-for-recommendation-systems-an-overview-71290690ecba

Figure 14 shows the expected architecture of *MARI*, as well as the interactions with the *RCM* and *EMERALD UI* components. We remind the reader that this is a draft of the architecture and interactions, and changes may apply during the project lifetime due to better arrangements.

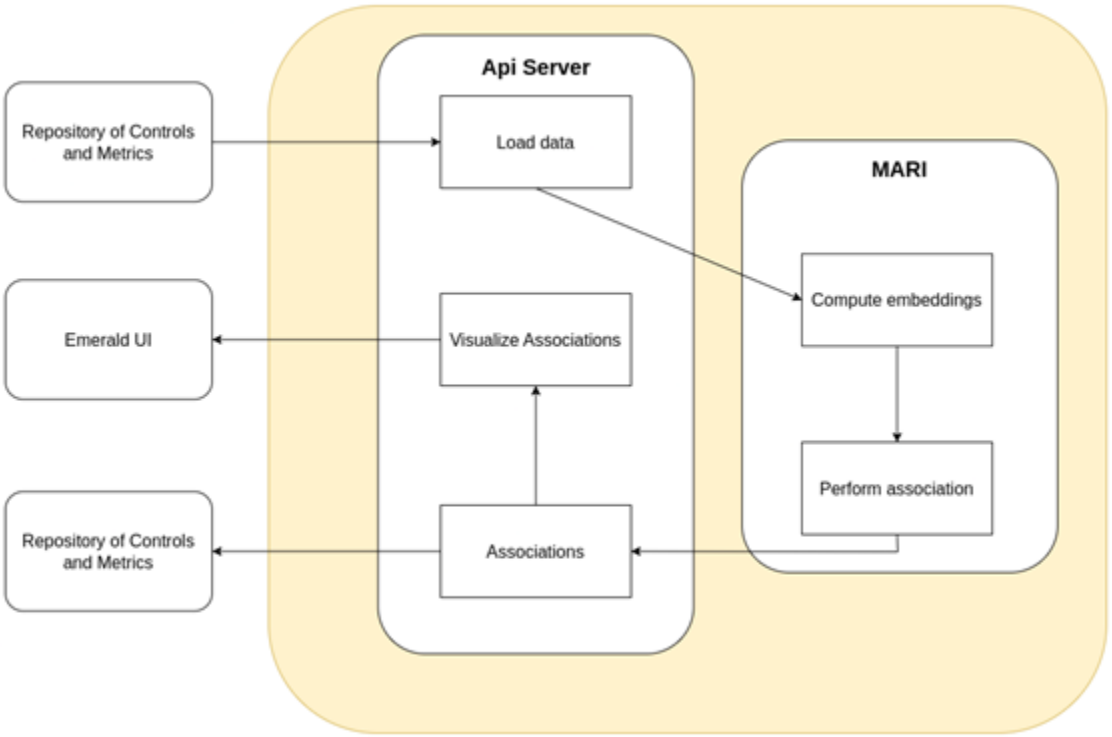

Figure 14. Overview of the MARI architecture and interactions

The *MARI* modules are organized into two components according to the functionalities they offer:

- **API Server**: It serves as the API interface for various EMERALD components, facilitating communication and interaction. In addition to this role, it also plays a crucial part in coordinating all operations and managing the connections with other components, ensuring seamless integration and functionality across the system.
- **MARI:** Given one control and a set of metrics, it provides the association between the control and related metrics. Given two certification schemes -1 and 2-, it provides the association between one control from scheme 1 and one control from scheme 2, if any.

6.1.2.1 Prototype architecture

The architecture of *MARI* includes subcomponents that map controls to metrics (and controls to controls). As the evaluation process analyses this information, it generates tailored recommendations. The output is represented by the controls and associated metrics (and the controls and associated controls). For each control, all of the most relevant metrics (controls) are displayed, in descending order. Arrows in the diagram indicate the flow of information, showing how data moves to the analytical component and ultimately to the output. Figure 15 shows one use case: the controls-metrics association.

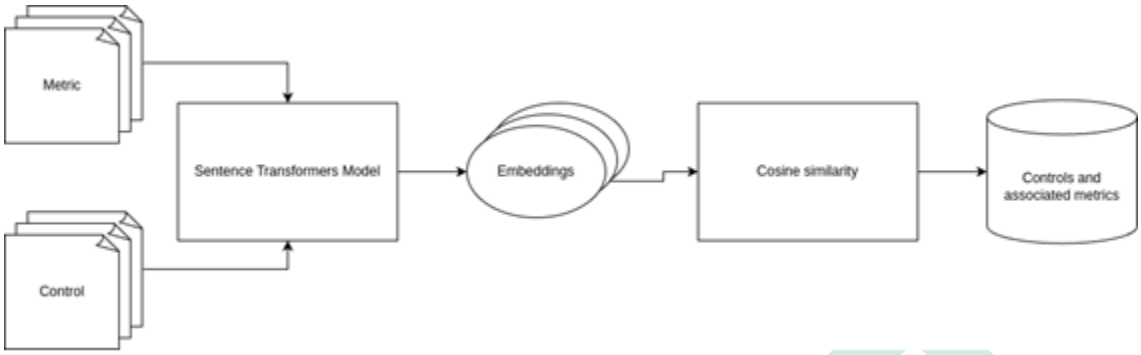

Figure 15. Detailed architecture of MARI component

Controls and metrics are stored in two separate CSV files. Both files are processed using the model *multi-qa-mpnet-base-dot-v1*, which generates embeddings for all the items. The *cosine_similarity* function from the *scikit-learn* library creates associations for each control[7]. The resulting associations are ranked by relevance and saved in a CSV file. The first column contains the ID of the control, while the subsequent columns list the associated metrics, ordered by their relevance. In the near future, we will also experiment with the control-control associations.

6.1.2.2 Technical specifications

The *MARI* component responsible for creating the associations is developed using Python version 3.10.12. Below is a selection of the main libraries utilised in the project. A complete list of all dependencies can be accessed through the GitLab repository.

- pandas
- sentence-transformers → *multi-qa-mpnet-base-dot-v1*
- sklearn
- numpy
- matplotlib
- numpy

The *multi-qa-mpnet-base-dot-v1* model is a sentence transformer designed for semantic search tasks, mapping sentences and paragraphs to a 768-dimensional dense vector space. It was trained on 215 million question-answer pairs from diverse sources such as WikiAnswers, Stack Exchange, MS MARCO, and more. The model intends to encode both queries and passages for efficient retrieval of relevant documents in a dense vector space. For longer texts, truncation occurs, which may affect accuracy, but this is not our case since the text of the metrics and controls does not reach long dimensions[15].

At the time of writing, MARI's developers are experimenting with the following sentence transformer models:

- all-mpnet-base-v2[16]
- multi-qa-mpnet-base-dot-v1[17]
- all-distilroberta-v1[18]

[15] Hugging Face, multi-qa-distilbert-cos-v1, https://huggingface.co/sentence-transformers/multi-qa-distilbert-cos-v1, 2024. Online; accessed 12 Sept 2024

[16] Hugging Face, all-mpnet-base-v2, https://huggingface.co/sentence-transformers/all-mpnet-base-v2, 2024. Online; accessed 12 Sept 2024.

[17] Hugging Face, multi-qa-mpnet-base-dot-v1, https://huggingface.co/sentence-transformers/multi-qa-mpnet-base-dot-v1, 2024. Online; accessed 12 Sept 2024

[18] Hugging Face, all-distilroberta-v1, https://huggingface.co/sentence-transformers/

- all-MiniLM-L12-v2[19]
- multi-qa-distilbert-cos-v1[20]

all-distilroberta-v1, 2024. Online; accessed 12 Sept 2024

[19] Hugging Face, all-MiniLM-L12-v2, https://huggingface.co/sentence-transformers/
all-MiniLM-L12-v2, 2024. Online; accessed 12 Sept 2024.

[20] Hugging Face, multi-qa-distilbert-cos-v1, https://huggingface.co/sentence-transformers/
multi-qa-distilbert-cos-v1, 2024. Online; accessed 12 Sept 2024

6.2 Delivery and usage

This section describes the information needed for the installation and use of *MARI.* Besides, it also details the licensing information and related packages and repositories.

6.2.1 Package information

The code in the repository follows the structure outlined in Table 11. There is no Docker configuration or API access in the current version. The code that generates the associations can be run through the Jupyter Notebook file *code.ipynb*, which takes two CSV files containing metrics and controls as input and returns a file in the same format with the associations between them, using a model based on sentence transformers.

The integration functionalities with the other components, as well as the API and the use of Docker will be developed in the course of the project.

Table 11. MARI package information

Folder/file	Description
`code.ipynb`	Jupyter notebook with the code for associating controls and metrics
`dataset/metrics.csv`	CSV file with the metrics data (input)
`dataset/controls.csv`	CSV file with the control data (input)
`dataset/recommendations.csv`	CSV file with association data (output)
`libraries.txt`	List of Python libraries and their versions
`README.md`	Project description and setup instructions

6.2.2 Installation

The project requires Python 3.10.12, which can be downloaded from the official Python website[10]. The installation can be verified from the terminal by running the following command:

```
python --version
```

The output should confirm that Python 3.10.12 has been correctly installed.

It is often beneficial to create a virtual environment for the project. A virtual environment keeps the project's dependencies isolated from the system's main Python installation. The command is:

```
python -m venv myenv
```

To activate the environment:

For Windows, the command is:

```
myenv\Scripts\activate
```

For macOS or Linux:

```
source myenv/bin/activate
```

The project relies on several Python libraries, all of which are listed in the `libraries.txt` file. By navigating to the project directory and using the following command, all necessary libraries can be installed with:

```
pip install -r libraries.txt
```

Once the setup is complete, the Jupyter Notebook file *"code.ipynb"* can be opened. This notebook contains the code required for the associations and can be accessed through Jupyter Notebook or JupyterLab by running the following command:

```
jupyter notebook code.ipynb
```

All these steps are also explained in the readme file of the EMERALD public Gitlab repository[21].

6.2.3 Instructions for use

To use the code in the file *code.ipynb,* the two CSV files containing the metrics and the controls must be placed in the dataset folder (these files are included in the current version of the code). After this, executing all the cells in the notebook generates the associations. The results are saved in a specific file named *recommendations.csv*, formatted as shown in Figure 16. This file will contain a list of controls associated with the metrics (*Requirement ID and Recommended Metrics*), ordered by relevance.

Controls ID	Recommended Metrics
OPS-05.3H	['OPS-04.1H', 'HR-03.4H', 'OPS-10.1H', 'OPS-12.1
OPS-07.2H	['OPS-07.2H', 'OPS-06.1H', 'OPS-06.1H', 'OPS-10
OPS-13.1H	['OPS-10.1H', 'IM-04.3H', 'OPS-13.3H', 'IM-07.1H',
OPS-18.6H	['OPS-18.6H', 'ISP-02.1H', 'ISP-02.1H', 'OPS-18.6I

Figure 16. Output file preview of the associations between controls and metrics

Within the EMERALD project, the *EMERALD UI* is used to access and manage the workflow in the framework. In the case of *MARI*, the UI interacts with it by using the components API endpoints. Currently, the *EMERALD UI* is work in progress, but some clickable mock ups have been designed in D4.3 [11], e.g. to present the results of the mapping of metrics to controls, as shown in Figure 17, or to present the mapping of controls between two security schemes, as shown in Figure 18).

[21] https://git.code.tecnalia.com/emerald/public/components/mari

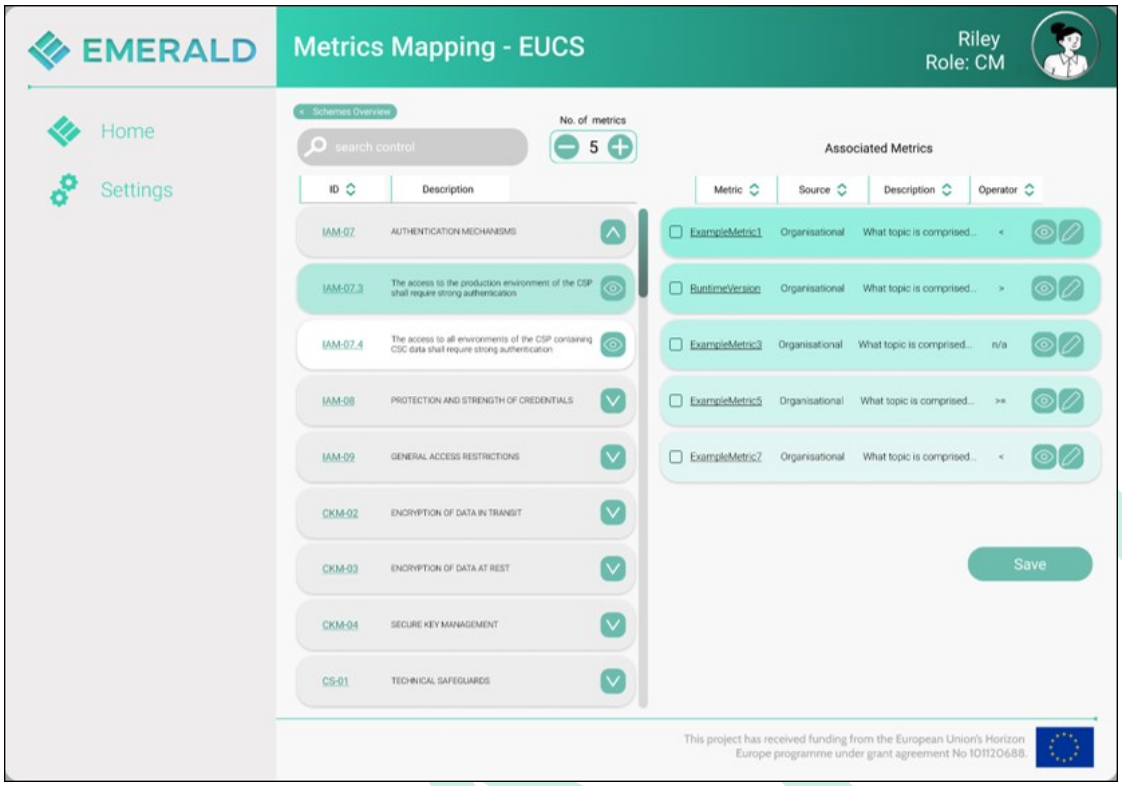

Figure 17. MARI - Overview of controls and mapped metrics (D4.3 [11])

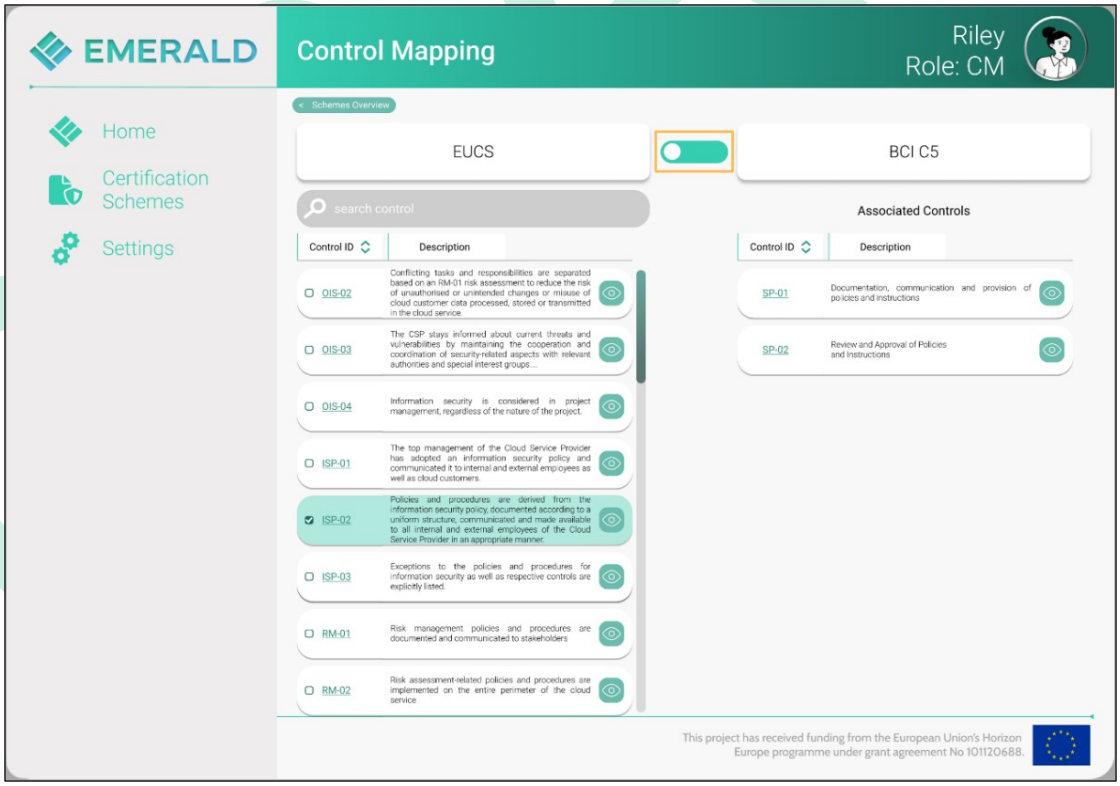

Figure 18. MARI - Mapping controls from EUCS to BSI C5 (D4.3 [11])

6.2.4 Licensing information

The *MARI* component is open source, under the Apache License 2.0.

6.2.5 Download

The source code can be found in public EMERALD repository[22].

[22] https://git.code.tecnalia.com/emerald/public/components/mari

7 Clouditor-Evaluation

The *Clouditor-Evaluation* component (in the following *Evaluation*) is responsible for evaluating (multiple) assessment result(s) to show the compliance for specific controls of a security catalogue.

7.1 Implementation

This component is based on the respective microservice of Clouditor. It is inspired by the work that was done in MEDINA [10].

7.1.1 Functional description

For given controls of a security catalogue and assessment results the *Evaluation* decides if the certification target is compliant with respect to a respective control. The *Orchestrator* starts the *Evaluation* component (see API endpoint *StartEvaluation* in Section 7.1.2.1.1) with information of the Certification Target ID and the Catalogue ID as well as an interval. This information is used to retrieve the respective assessment results and the controls from the *Orchestrator* and aggregate them into an evaluation result. The evaluation is carried out periodically, the time span depends on the interval that has been set (if not set, the default value of 5 minutes will be used).

Table 12 outlines the functional requirements satisfied by the current version of the *Evaluation*, as documented in D3.1 [2], and updates the status of their implementation in the current prototype in M12.

Table 12. Evaluation Functional Requirements

Req. ID	Description	Priority	Milestone	Progress
EVAL.01	**Display cause of failing evaluation result:** We want to know why the evaluation result fails or passes. Therefore, it should contain a list of assessment results that cause the evaluation status to be non-compliant.	Must	MS6 (M30)	100%
EVAL.02	**Evaluation based on assessment results:** The evaluation should assess the result based on all the required assessment results stored in the database.	Must	MS6 (M30)	100%

7.1.1.1 Fitting into overall EMERALD Architecture

The only connection of the *Evaluation* to another component in the EMERALD framework is to the *Orchestrator*, see Figure 19. Its only purpose is the evaluation of multiple assessment results for a given control. The decoupling from the *Orchestrator* allows to scale up fast and easy if there is a need for higher performance, e.g. because there are so many assessment results and many controls, we need evaluation for.

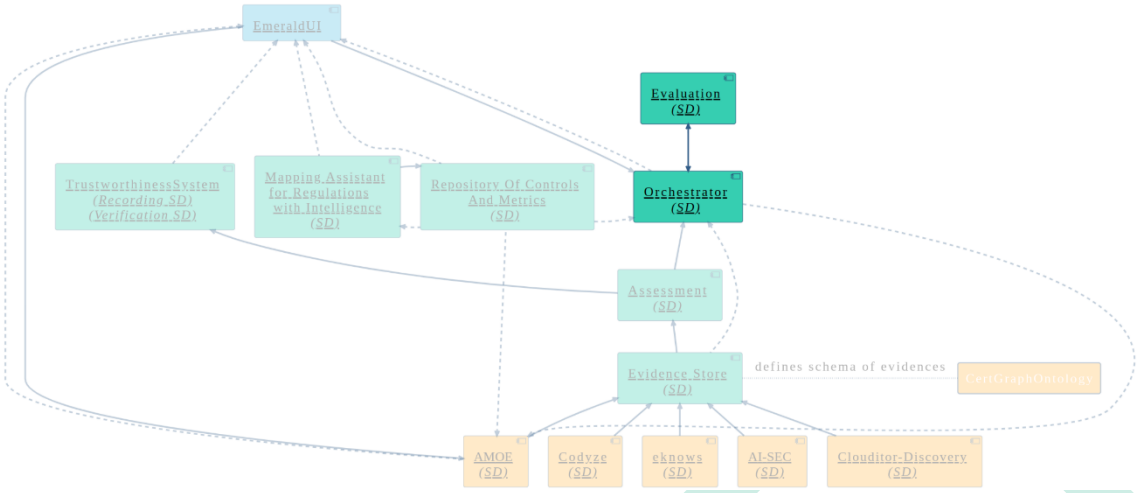

Figure 19. Role of the Evaluation in the EMERALD framework

7.1.2 Technical description

The technical description of the *Evaluation* provides an in-depth look at its architecture, components, and technical specifications. This section outlines the structure of the *Evaluation*, as well as the specific technologies and methods used in its implementation. The following subsections detail the prototype architecture, components, and technical specifications.

7.1.2.1 Prototype architecture

The architecture of the *Evaluation* is shown in Figure 20. The *Evaluation* can provide REST and gRPC endpoints for connectivity. In the EMERALD framework, the *Evaluation* only connects to the *Orchestrator*, which utilizes the gRPC endpoints.

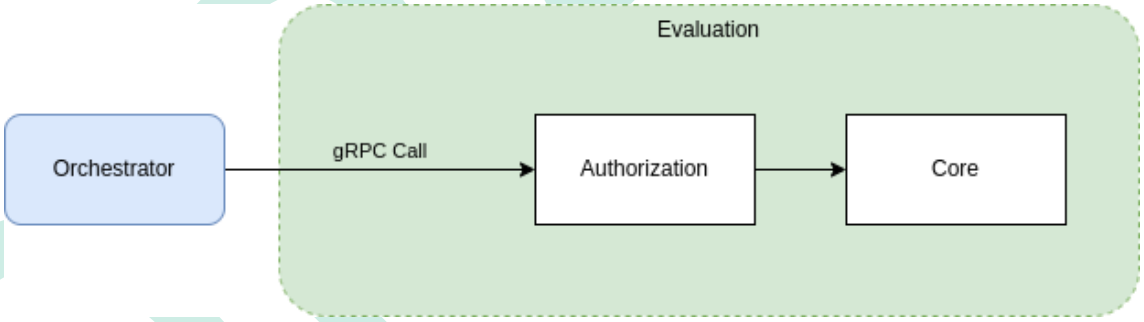

Figure 20. The prototype architecture of the Evaluation component

7.1.2.1.1 Components description

The architecture of the *Evaluation* includes the following key elements and functionalities:

- Clouditor's *Authorization*: The *Evaluation* uses Clouditor's Authorization implementation to provide state-of-the-art authentication mechanisms, such as OAuth2, which is used in EMERALD.

- The *Core* (*evaluation.go):*
  - Function *NewService* for instantiating a new *Evaluation* (can potentially be scaled to several instantiations to handle more evaluation requests).
  - API-corresponding functions that implement the *Evaluation* Service interface (for all relevant API endpoints see below), e.g. *StartEvaluation* for starting the

evaluation for a specific audit scope (combination of certification target and catalogue).

In the following, all API endpoints are listed accompanied by a short explanation:

- *StartEvaluation*: Evaluates periodically all assessment results of a certification target based on the given catalogue. It is also possible to set the interval, which specifies the time between repeated executions. Exposed via gRPC as well as REST.
- *StopEvaluation*: Stops the evaluation for the given certification target and catalogue. Exposed via gRPC as well as REST.
- *ListEvaluationResults*: List all evaluation results that the caller (user) can access. Filtering options are available for the endpoint, allowing users to filter by certification target and catalog. Exposed via gRPC as well as REST.
- *CreateEvaluationResult*: Only manually created evaluation results can be generated through this endpoint. Exposed via gRPC as well as REST.

7.1.2.2 Technical specifications

The *Evaluation* is implemented using Go, providing efficient concurrency support and ease of deployment in microservice architectures. The main communication protocols used are REST and gRPC, ensuring high-performance interaction with other components.

- Programming Language: Go
- Communication Protocols: REST API and gRPC (including Protobuf)
- Security: OAuth and Rol-Based Access

7.2 Delivery and usage

This section describes the information needed for the installation and use of the *Evaluation.* Besides, it also details the licensing information and related packages and repositories.

7.2.1 Package information

Table 13 shows the *Evaluation*-relevant package structure in the Clouditor repository and *Table 14* shows the package structure in the EMERALD framework, where the *Evaluation* parts of the Clouditor are used as dependencies.

Table 13. Evaluation relevant package structure

Folder	Description
`api/evaluation/`	This folder contains code needed for the communication with this component. It mainly consists of auto-generated Protobuf and gRPC files.
`cmd/evaluation/`	This folder contains the main file.
`openapi/evaluation/`	This folder contains the auto-generated OpenAPI files for the Evaluation.
`rest/`	This folder contains the REST gateway implementation.
`service/evaluation/`	This folder contains the source code for the *Evaluation* microservice.

Table 14. Evaluation package structure in the EMERALD framework

Folder	Description
`cmd/emerald-evaluation`	This folder contains the main file.
`./ (Root directory)`	Beside the folders mentioned above, the root directory also contains a workflow file needed for continuous integration and deployment (`.gitlab-ci.yml`), *a README* file, a Go file for launching the EMERALD Assessment and Go module files for handling dependencies (`go.mod` and `go.sum`)

7.2.2 Installation

In EMERALD, we use GitLab's CI/CD pipeline for continuous integration and deployment. For this purpose, there is workflow file at each components root level (`.gitlab-ci.yml`).

For running the *Evaluation* locally, there is a docker file ("*Dockerfile*") located at the components root level. For building and running the component, use the following commands:

```
docker build -t clouditor-evaluation .
docker run -d -p 8080:8080 clouditor- evaluation
```

7.2.3 Instructions for use

The *Evaluation* is only used by one internal component, the *Orchestrator*. It is not exposed via the *EMERALD UI*. But information of this component, the evaluation results, are stored in the *Orchestrator* which are presented in the *EMERALD UI* eventually.

To use the *Evaluation* component, simply run the Docker image and wait for information sent by the *Orchestrator*, e.g. which assessment results are needed for a given control. The *Orchestrator* is continuously leveraging the *Evaluation* (or multiple instances of it) to calculate evaluation results for a given set of controls.

7.2.4 Licensing information

The *Evaluation* is offered under Apache 2.0 license. The license files and more detailed information can be found in the GitLab repository.

7.2.5 Download

The source code for the Evaluation component in the Clouditor toolbox can be found in the Clouditor GitHub repository[23]. The implementation of the Evaluation in EMERALD can be found in the public EMERALD repository[24].

[23] https://github.com/clouditor/clouditor
[24] https://git.code.tecnalia.com/emerald/public/components/evaluation

8 Repository of Controls and Metrics (RCM)

The *Repository of Control and Metrics* (RCM) is a smart catalogue of controls and metrics that provides a central point in the EMERALD frameworks where Compliance managers and Auditors can obtain all the information related to security certification schemes (i.e., controls, security requirements, assurance levels, etc.), that is, everything that can be considered "static" information that appears in the scheme. The *RCM* supports multi-scheme and multi-level certification and incorporates the definition of the metrics used in EMERALD to assess evidence.

The *RCM* will also provide mechanisms to update the catalogues and maintain a versioning system and will foster the interoperability using OSCAL[25] as exchange format. This feature will allow importing and exporting catalogues to facilitate the reuse and composition of the catalogue elements. In addition, the *RCM* will manage other information, such as the mapping of controls and metrics provided by the MARI component, the guidelines (e.g., guidelines for EUCS requirements are already included) and a self-assessment questionnaire to assess compliance for the EUCS [7].

8.1 Implementation

The implementation of the *RCM* component is based in the MEDINA component *Catalogue of Controls and Metrics*[26] [21]. This foundational tool was the baseline for elaborating the specifications of the *RCM*, that are documented in D3.1 [2]. This section presents the functional description and technical description of the component.

8.1.1 Functional description

The *Repository of Controls and Metrics* (RCM) provides a central point in the EMERALD framework where the **certification schemes are stored and managed**. It consists of a repository capable of containing different certification schemes, including the information of each scheme categorized by classes (e.g., categories, controls, assurance levels, etc.) and supporting multi-scheme and multi-level certification. The RCM also incorporates the definition of the **metrics** used in EMERALD to assess evidence.

The *RCM* provides the user with an automated tool where a Compliance Manager or an Auditor can select a security scheme and obtain all related information and guidance by navigating though the scheme via the user interface, instead of having to read a paper document. This is what we can refer as the "static" information of the standard.

The *RCM* will provide mechanisms to update the catalogues and maintain a versioning system and will foster the interoperability using OSCAL as exchange format. This feature will allow importing and exporting catalogues into/from the *RCM*. In addition, the RCM will manage other information, such as:

- The mappings generated by the MARI component, which provide information on related controls from different security schemes.
- The control implementation guidelines (guidelines for EUCS requirements are already included).
- A self-assessment questionnaire, that allows the user to assess the compliance against the EUCS security scheme. It includes all levels of certifications (Basic, Substantial and High), with several questions to check the fulfilment at sub-control level ("requirements" in EUCS). It also allows the user to enter comments, textual references

[25] OSCAL: Open Security Controls Assessment Language, https://pages.nist.gov/OSCAL/
[26] https://git.code.tecnalia.com/medina/public/catalogue-of-controls

to locate the evidence supporting the answers, and non-conformities for each requirement that is not fulfilled.

Table 15 shows the functional requirements satisfied by the current version of the *RCM*, as documented in D3.1 [2], and updates the status of their implementation in the current prototype in M12.

Table 15. RCM Functional Requirements

Req. ID	Description	Priority	Milestone	Progress
RCM.01	**Multi-schema support**: The repository should contain at least an additional security scheme, apart from the EUCS that is the scheme implemented in MEDINA Catalogue and is inherited in EMERALD	Must	MS2 (M12)	90%
RCM.02	**Accessible by the rest of components**: The repository content should be made accessible to the rest of EMERALD components via API	Must	MS2 (M12)	100%
RCM.03	**Include metrics for all schemes supported**: The repository should include metrics that could be used to assess the compliance with one or more certification schemes	Must	MS2 (M12)	30%
RCM.04	**Mapping of schemes**: The repository should support the mapping of the certification schemes contained. The scheme-to-scheme mapping will be provided by the MARI tool and stored in the repository. The rationale for the mapping decision will also be stored.	Should	MS5 (M24)	10%
RCM.05	**Import/export of security schemes in OSCAL**: The repository is able to import a new scheme defined in the OSCAL language (this feature can also be used to update an existing scheme). The repository is able to export any available scheme in OSCAL format.	Must	MS6 (M30)	40%
RCM.06	**Import/export of security schemes in CSV format**: The repository can export a scheme to a CSV file, and import a CSV file with the same format as a new scheme	Could	MS2 (M12)	60%
RCM.07	**Support for personalized catalogues**: The Repository has to offer the user the possibility to create a personalized catalogue of controls. These controls can be taken from the same or from different security schemes.	Must	MS6 (M30)	0%
RCM.08	**Support updating/versioning of schemes**: The Repository has to maintain a versioning system of the schemes it contains, so that if a new version is uploaded, it is able to detect the change and notify the user that a new version is available	Should	MS6 (M30)	10%

8.1.1.1 Fitting into overall EMERALD Architecture

The *RCM* is one of the components of the EMERALD architecture. Figure 21 shows the interaction with other tools in the EMERALD framework.

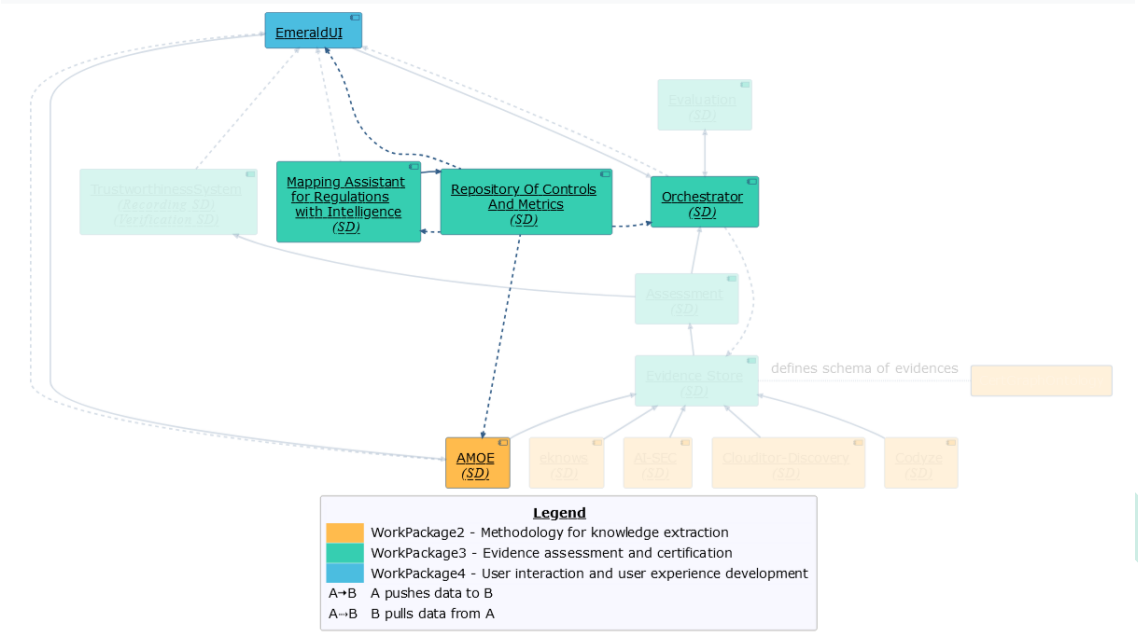

Figure 21. Fitting of the RCM with other components in the EMERALD architecture

The main interactions of the *RCM* with the EMERALD components are as follows:

- ***Clouditor-Orchestrator*** retrieves the information about the schemes and the metrics from the *RCM*. It stores this information internally for efficiency reasons. This information is used to configure the extractors and organize the evidence.
- ***Mapping Assistant for Regulations with Intelligence* (MARI)** gets the same information about the schemes and the metrics from the *RCM*, and processes it to obtain the required mappings, that are then sent back to the *RCM* in order to store them for further use.
- ***EMERALD UI*** uses the *RCM* API-REST to call services depending on the user request. The information required is then translated into internal queries to the *RCM* Database, which returns the data. This data is then packed in JSON format in the REST call and sent to the *EMERALD UI* for displaying. Other types of interaction occur when the user wants to change the content of the repository (for example, introducing a new scheme), or fill in the self-assessment questionnaire.
- ***AMOE*** is a knowledge extractor from documents. It obtains from the *RCM* the definition of the security metrics that are used to evaluate evidence in policy documents.

8.1.2 Technical description

This section describes the technical specification of the *RCM* component. First, we present the main architecture of the prototype, including all its sub-components. Next, the technical specifications of the developed system are presented. The section finishes with the description of the published APIs.

8.1.2.1 Prototype architecture

The *RCM* architecture is built using a microservices architecture[27], which divides the front end and back end to make it easier to scale for increasing user counts and to withstand infrastructure

27 https://www.jhipster.tech/tech-stack/

problems. Additionally, this approach gets everything ready for the phases of exploitation and sustainability.

The *RCM* comprises five main sub-components, as illustrated in Figure 22. They are briefly outlined as follows:

- **Frontend**: This sub-component serves as the *RCM*'s graphical user interface, enabling users to filter the view and choose the specific information they wish to review from the existing schemes (e.g., controls of a certain scheme, metrics related to specific controls, etc). This sub-component will be developed as part of the *EMERALD UI* and will interact with the backend through the API. The component will also provide a second "internal" frontend, for development and management purposes, that makes it possible to use the *RCM* alone.
- **Backend**: This is the central sub-component of the *RCM*, responsible for implementing the APIs to manage the scheme data based on the user-defined filters via the UI or API calls. In a general microservices architecture, it can consist of multiple specialized applications, each containing a few related entities and business rules.
- **Converter Backend,** which is dedicated to scheme conversions to/from OSCAL, and other possible import/export functionalities.
- **Registry,** which is an internal sub-component provided by the framework that facilitates the creation of a microservices architecture component that interconnects the other sub-components and enables their communication.
- Furthermore, data persistence is facilitated by a SQL **database** (MySQL) connected to the backend.

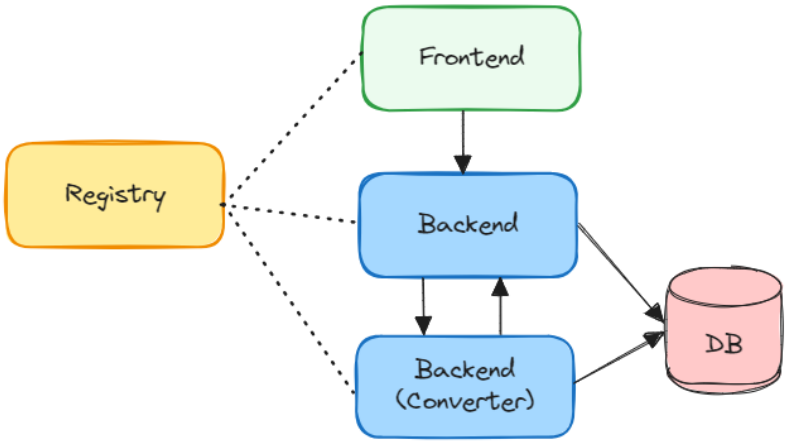

Figure 22. Architecture of the Repository of Controls and Metrics (RCM)

8.1.2.1.1 Components description

The components of the *RCM* are detailed in the following paragraphs.

Frontend

It is the graphical user interface of the *RCM*. Its main purpose is to serve as the interface for the user to interact with the *RCM*. It will be constructed based on two main foci (i) the information contained in the *RCM*, that has to be made available to the user; and (ii) the user needs to work, through the EMERALD workflow, with that information. The result is a set of screens that will serve primarily to present the information contained in the *RCM* to the user, and a set of commands (menu items or buttons) to produce actions to Create/View/Edit the elements on the screen.

Regarding the View actions, the frontend should allow the users filtering and ordering the set of information they want to view. For example, controls from a certain framework, or metrics related to a control.

Regarding the Create/Edit actions, the user is quite limited with the *RCM*, because its main objective is to store existing norms or standards that, logically, should not be modified by the user. However, some actions are permitted like the creation of particular sets of controls to create a personalised user-owned schema.

Backend

It is the core sub-component of the Repository. It performs the actual discovery of the data entities like controls, metrics, etc. from the security schemes stored in the database. The backend is a passive sub-component that waits the call to the API endpoints. Whenever the backend receives a call, it performs the following consecutive steps:

1. Receives the API calls from the Frontend.
2. Transforms the call them into SQL queries.
3. The queries are executed against the MySQL database.
4. The results (if any) are packed into the JSON data schemas.
5. The information is returned to the caller, with a code indicating the success/error of the call.

The possible filters established by the user through the UI/API are translated directly to queries to produce the desired results. The backend makes use of a MySQL database containing the data of the stored schemas and the defined metrics.

Converter Backend

This is a new sub-component that deals with conversion facilities, not available in the MEDINA Catalogue, that have been introduced in EMERALD. The Converter Backend mission is to "translate" any scheme stored in the RCM to a standard language. The OSCAL language has been selected as the main conversion language, but conversion from/to CSV files will also be implemented.

The Converter is a stand-alone backend, which will be deployed as a separate container, and is coded in Python language. The operation is typical of a backend: when it receives an export order from the frontend, it will access the database, extract the scheme, apply the conversion logic and provide as output the same information but adapted to the schema of the OSCAL interchange language. When dealing with an import order, the logic will translate the OSCAL/CSV entry into the internal schema of the database and provide as output a new security scheme available in the *RCM*.

Registry

The Registry in this version is Consul, provided by HashiCorp[28]. It has to be setup before the rest of the subcomponents, as it stores internal information about the Frontend and Backend and performs checks to control that the whole framework is up and running. It guarantees the security of the *RCM* component.

[28] https://www.consul.io/

8.1.2.2 Technical specifications

On the server side, the *Backend* and the *Registry* will use Maven, Spring MVC REST for the API, Spring Data JPA, Netflix OSS[29] and Python - Flask REST API[30].

The internal *Frontend* will make use of Angular, Webpack, Yeoman, JavaScript, and Bootstrap technologies on the client side.

The API endpoints of the *RCM* are briefly described in the following:

- *Schema*: Retrieves the information about a certification schema (categories, controls, sub-controls, metrics, etc) as needed. It includes filters to extract only the required data, as well as search functionalities. Exposed as a REST API.
- *Mapping*: Sets a control mapping among schemes, where controls of two different schemes are considered equivalent and thus linked. It will be called by the *MARI* component. Exposed as a REST API.
- *Import-export*: Manages import/export of security schemes in OSCAL. It will accept as import a scheme written OSCAL, following a pre-defined template that must be defined. The export functionality will provide as output the schema selected by the user from the *RCM* in the mentioned OSCAL format. Exposed as a REST API.

8.2 Delivery and usage

This section describes the main packages of the *RCM*. Some instructions for installation and use are presented and the section finishes with information about licencing and download.

8.2.1 Package information

The *RCM* contains the following main packages:

- The **Backend** package[31], responsible for implementing the logic of the RCM and manage the persistence of the data used and generated.
- The **Converter** package[32], responsible for performing the conversions between the different supported catalogue formats.
- The **Development Frontend** package[33], responsible for provisioning the web interface to use the functionalities of the RCM.

Besides, the *RCM* requires some additional packages (side services) from state of the practice to fulfil its features (see Figure 23):

- **Consul** package, responsible for providing the configuration to the *RCM* main components and enabling the discovery of the backend services from the frontend services.
- **MariaDB**[34] package, responsible for providing the persistence of the data required and generated by the *RCM*.
- **Gateway** package, responsible for providing routing for the requests received in the HTTPS port to the frontend package and other side services. Besides, it is also responsible of providing and maintaining the HTTPS certificates negotiated with online

29 https://www.jhipster.tech/microservices-architecture/

30 https://flask.palletsprojects.com/en/3.0.x/

31 https://git.code.tecnalia.com/emerald/public/components/rcm/backend

32 https://git.code.tecnalia.com/emerald/public/components/rcm/converter

33 https://git.code.tecnalia.com/emerald/public/components/rcm/frontend

34 https://mariadb.com/

certificate authorities such as let's encrypt[35]. For this purpose, RCM uses state of the practice components:

- Traefik[36] in the local development environment.
- Nginx[37] ingress + Certmanager in the integration environment.

- **Keycloak[38]** package, responsible of provisioning of the identification service for the usage of the RCM service.

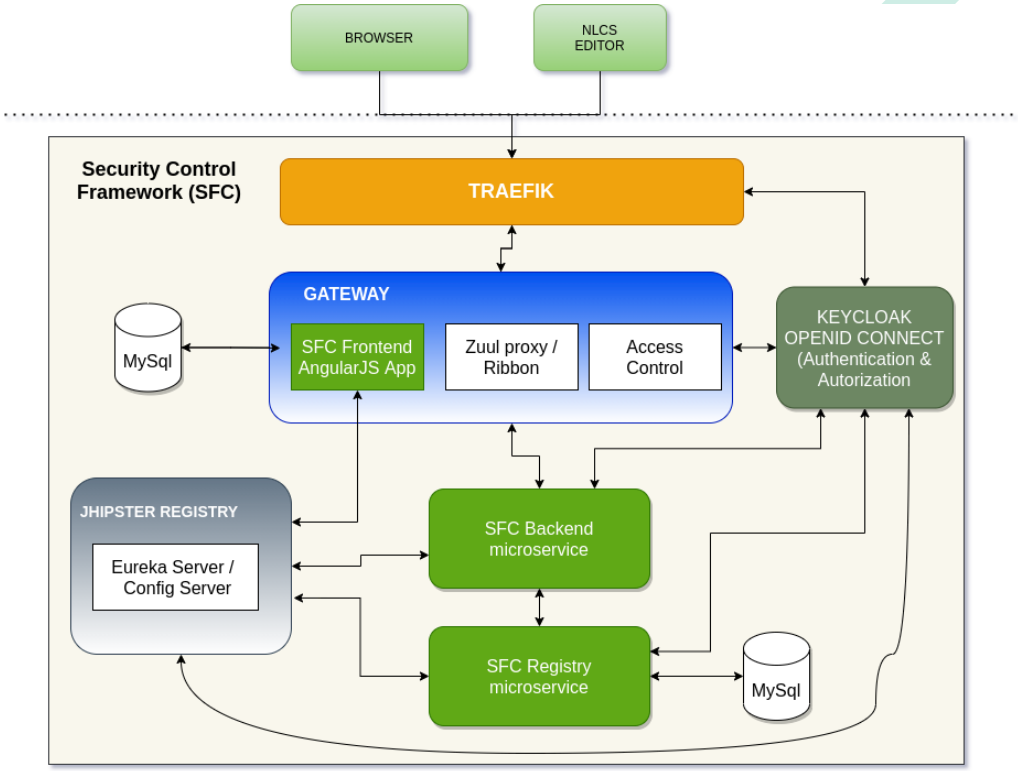

Figure 23. Repository components (green boxes) and auxiliary elements

The usage of state of the practice components requires the implementation of configurations components to prepare those generic services with the required information for the *RCM*:

- **mariadb-setup-dbs** package, responsible of creating the databases required by the *RCM* main subcomponents.
- **mariadb-setup-db** package, responsible of adding the initial information to the databases to have some initial content in the *RCM* main subcomponents.
- **consul-config-loader** package, responsible of adding the configuration properties required by the *RCM* main subcomponents.
- **keycloak-setup-realms** package, responsible of creating the emerald realm in local development environments.
- **keycloak-setup-realm** package, responsible of adding the required configuration for the *RCM* on keycloak.

[35] https://letsencrypt.org/
[36] https://traefik.io/
[37] https://nginx.org/
[38] https://www.keycloak.org

The code in charge of starting and configuring these last sets of components (the ones that provide the side services and the ones that configure them) are contained in the composition repositories. The main composition repository is the CaaS Framework[39] that will be released later in the project. Besides, in the case of *RCM* we have a local composition for development purposes[40]. This composition relays in docker compose technology because it is easier to be used in an isolated way.

8.2.1.1 Backend

The Backend subcomponent is divided into several subpackages. Being composed of Java classes, each of these subpackages has its main purpose and context within the prototype as a whole. They are the following:

- **logging**: This package consists of the `LoggingAspect.java` class that defines the aspect for logging execution of Spring service and repository components.
- **client:** This package consists of the `UserFeignClientInterceptor.javaclass` that implements `RequestInterceptor.java`. This class checks and adds a JWT token to the request header.
- **config:** This package contains all the classes related to configuration purposes.
- **domain:** This package contains data model classes.
- **repository:** This package contains Spring Data SQL repository classes.
- **security:** This package contains Spring Security related classes for security management.
- **service:** This package contains backend services for CRUD operations and other requirements needed.
- **web:** This package contains classes to expose backend rest end points.

8.2.1.1 Converter

This package, as a REST API developed in Python – Flask, requires the installation of the following packages as dependencies:

- **cryptography (v3.3)**: This package provides cryptographic recipes and primitives.
- **Flask (v2.0.0)**: This package consists of a lightweight application framework to expose the component as a REST API.
- **Flask-JWT-Extended (v4.4.1)**: This package provides JSON Web Tokens support to the previous package.
- **jsonschema (v4.21.1)**: This package allows to validate JSON objects.
- **pymysql (v1.1.0)**: This package is used to manage the internal database.
- **python-dateutil (v2.9.0)**: This package consists of an extension of the basic datetime package.
- **pytz (v2024.1)**: This package allows accurate and cross platform time zone calculations.
- **Werkzeug (v2.2.2)**: This package is a comprehensive WSGI web application library.

8.2.1.2 Development Frontend

The Frontend subcomponent is divided into several subpackages, including web resources and Java packages.

The web is implemented with Angular technology and includes several packages:

[39] https://git.code.tecnalia.com/emerald/public/caas-framework
[40] https://git.code.tecnalia.com/emerald/public/components/rcm/docker-compose-rcm

- **admin**: This package includes admin focused features such as health-checks, logs, or metrics.
- **config**: This package manages the gathering of the config of the frontend to adapt to different environments.
- **core**: This package includes core functionalities such as authentication management or error managements.
- **entities**: This package is the most important part as it contains the frontends to manage the resources of the *RCM*.
- **home**: This package contains the welcome page related assets.
- **layout**: This package contains elements to customize the look and feel of the frontend to different environments.
- **Login**: This package contains login related assets.
- **shared**: This package contains utility functionalities to be used by the previous elements. It includes functions to manage dates, pagination, language, etc.

The java part provides logic to manage users, roles and access to the *RCM* backend assets. It is organized in a similar way to the backend.

- **logging**: This package consists of the `LoggingAspect.java` class that defines the aspect for logging execution of Spring service and repository components.
- **client**: This package consists of the `UserFeignClientInterceptor.java` class that implements `RequestInterceptor.java`. This class checks and adds a JWT token to the request header.
- **config**: This package contains all the classes related to configuration purposes.
- **domain**: This package contains data model classes.
- **repository**: This package contains Spring Data SQL repository classes.
- **security**: This package contains Spring Security related classes for security management.
- **service**: This package contains frontend services for CRUD operations and other requirements needed.
- **web**: This package contains classes to expose frontend rest end points.

8.2.2 Installation

The *RCM* has been developed to be used in a container-based environment. For the installation we consider two scenarios:

1. Development: This scenario is focussed on the coding and testing of the *RCM* on the developer side independently from the integration platform status.
2. Integration: This scenario is focussed on the provision of the CaaS Framework as a single solution.

8.2.2.1 Installation in the Development environment

This installation is defined by the use of docker compose technology, which provides several advantages:

- It can be run in the development computer. This reduces the need to acquire and configure external servers saving time and money.
- It allows to instantiate additional packages (MariaDB, keycloak, gateway …) allowing to identify and debug issues that will appear in the integration environment saving time and dependencies.
- It allows to prepare automatic configuration procedures for these additional services saving time during deployment and providing replicability.

- It allows to easily destroy and create the *RCM* allowing to simulate migration scenarios which are expected to happen during the life of the project.

These are the steps to install and execute the *RCM* in a development environment:

1) Fulfil these few requirements:

   - The computer should have docker and git installed.
   - The computer should have a traefik with certificates configured in a custom network called *traefik_network*
   - SERVER_HOST variable pointing to the name of the host that resolves the IP address of the machine. E.g., 192.168.56.5.nip.io
   - Alternatively, an ADMIN_PASSWORD variable to overwrite the default ADMIN_PASSWORD specified in the .env.

2) Clone the repository:

   git clone --recursive https://git.code.tecnalia.com/emerald/public/components/rcm/docker-compose-rcm

3) Later, to use the docker compose we have several options depending on what we require to do. The first purpose could be to check the *RCM* using the development frontend.

   docker compose -d

Once docker-compose is successfully deployed, and assuming the following value for SERVER_HOST (192.168.56.5.nip.io), we will be able to access the Repository services at:

```
https://rcm.192.168.56.5.nip.io Repository
```

Other services that are deployed to help in the development phase are a consul, MariaDB, keycloak.

```
https://adminer.192.168.56.5.nip.io MariaDB.
https://consul.192.168.56.5.nip.io Consul.
https://keycloak.192.168.56.5.nip.io keycloak.
```

8.2.2.2 Installation in the Integration environment

The installation in the integration environment implies the deployment of the CaaS framework, which involves three fundamental steps:

1) To install the development environment there are few requirements:

   - Create/adquire a Kubernetes cluster or obtain access to existing one.
   - Get the cloud.yaml to use kubctl

2) Clone the repository:

   ```
   git clone https://git.code.tecnalia.com/emerald/public/caas-framework
   ```

3) Later, use kubectl to start the CaaS framework that includes the *RCM*.

   ```
   kustomize build . | envsubst | kubectl apply -f-
   ```

Once docker-compose is successfully deployed, and assuming the following value for hostname (project.domain), we will be able to access the Repository services at:

```
https://rcm.project.domain Repository
```

Other services that are deployed to help in the development phase are a Consul, MariaDB, and keycloak.

```
https://adminer.project.domain MariaDB
https://consul.project.domain Consul
https://keycloak.project.domain keycloak
```

8.2.3 Instructions for use

A graphical user interface (GUI) is needed by the repository to access and edit the various entities stored in the database. For every primary entity, a CRUD API (Create/Retrieve/Update/Delete) has been created; however, the activities that may be performed can vary according to the role of the user.

The GUI lets the user interact with the security schemes by letting them utilize buttons, links, and filters, among other graphic components, on different screens. Typical tasks could be, for example, to show the metrics associated with a specific control; to select the controls of a specific category; or to view which controls in a scheme correspond to a particular assurance level.

The GUI that interacts with the *RCM* will be integrated in the *EMERALD UI* component. At the time of writing, it has not been yet developed, but some paper mock ups and digital mock ups have been designed in D4.3 [11] that allow access the functionality of the *RCM*. This section presents some of the available mock ups, showing the principal interactions with the *RCM* interface. Most of the screens are dedicated to navigation though a scheme for informative purposes. In a future, specific mock-ups will also be developed to handle the EUCS questionnaire.

The user accesses the *RCM* by clicking the "Certification Schemes" menu option in the in the EMERALD home page (see Figure 24).

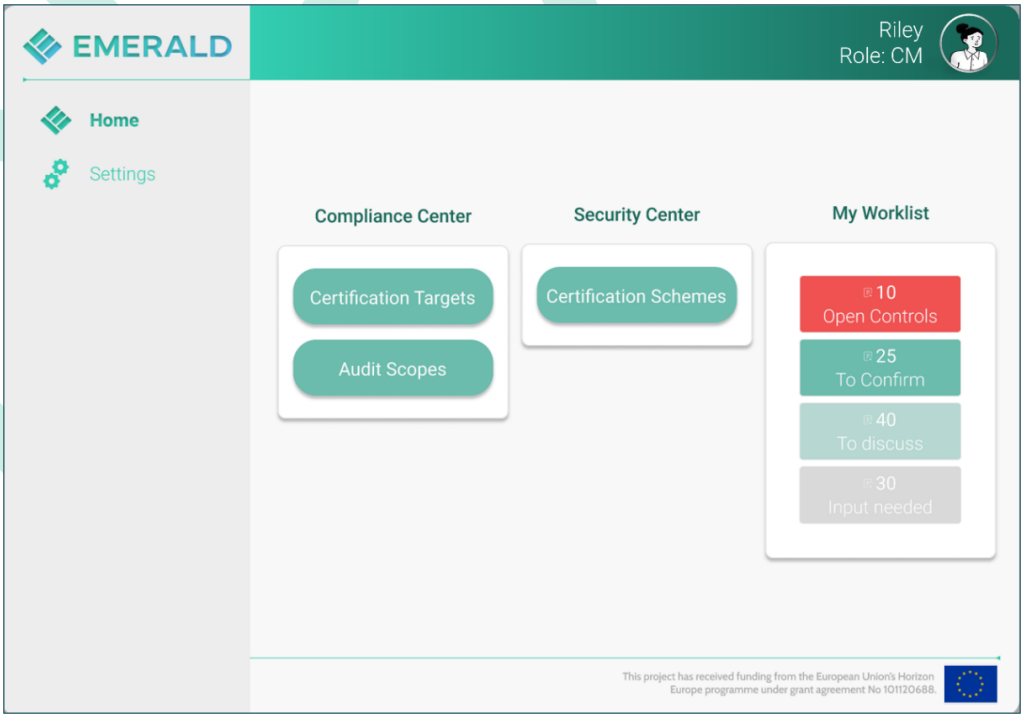

Figure 24. Home page of the EMERALD framework (D4.3 [11])

When entering the "Certification Schemes" area, a list of all available schemes is presented, as shown in Figure 25.

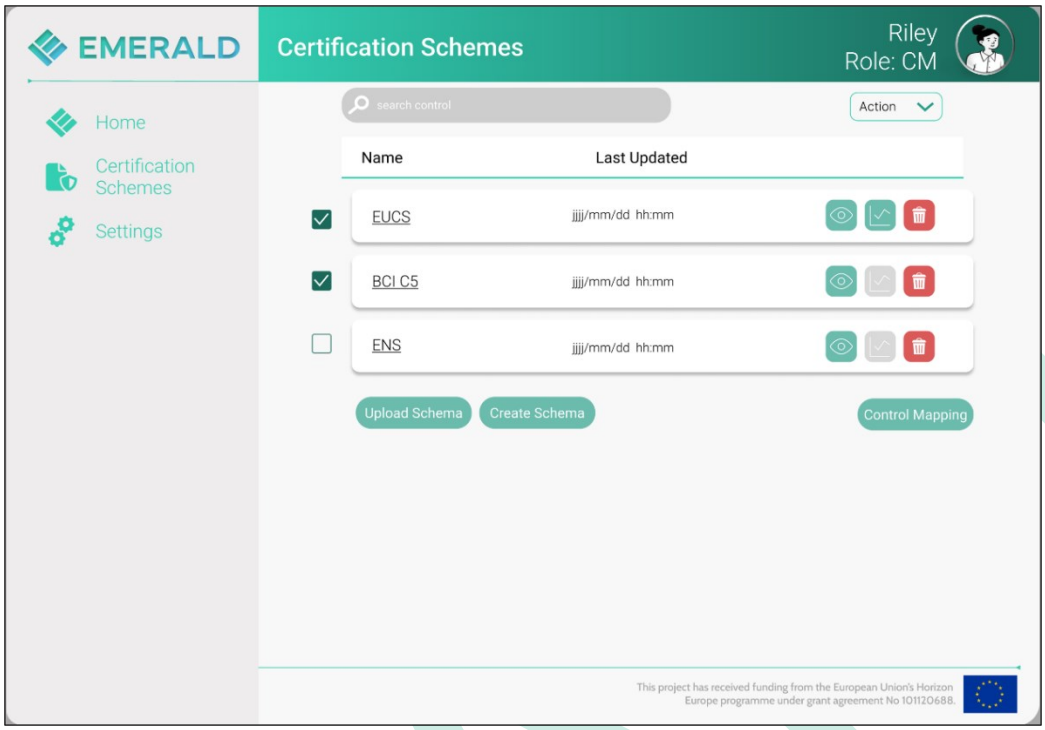

Figure 25. List of schemas page (D4.3 [11])

When clicking on the "Upload Scheme" button, a window opens that allows to upload a new scheme selecting CSV or OSCAL files (see Figure 26).

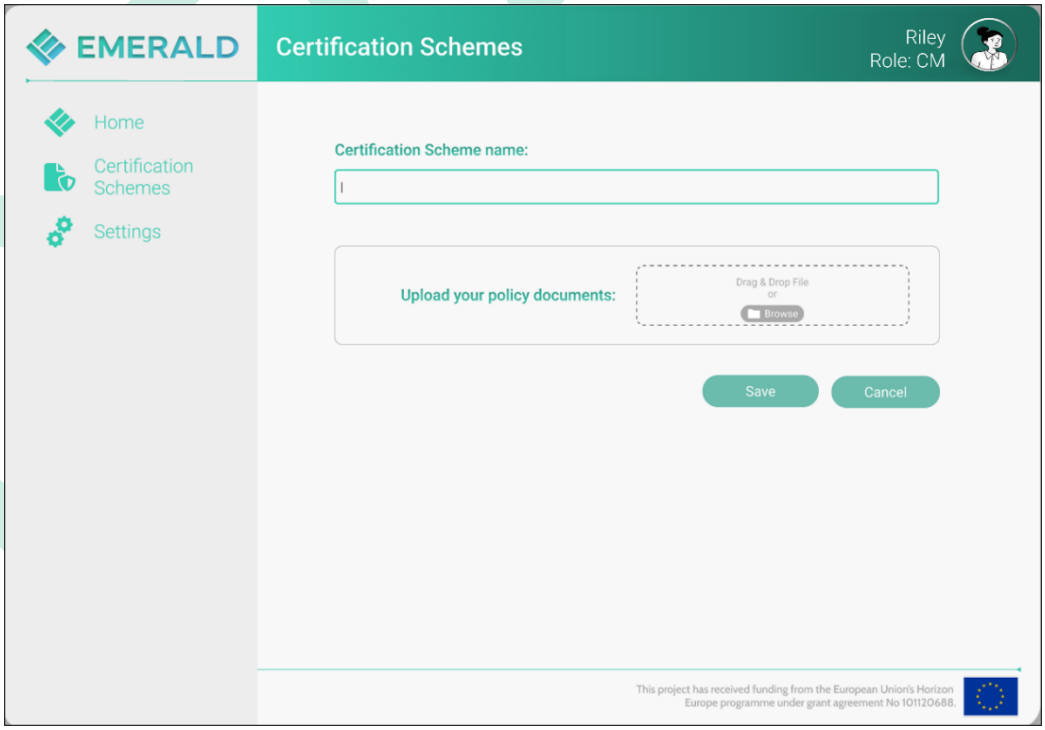

Figure 26. Upload new scheme page (D4.3 [11])

When clicking on the scheme title or on the "view" button in Figure 25, the details of the respective scheme are shown, as can be seen in Figure 27. Here, the name, code and description of each category is shown.

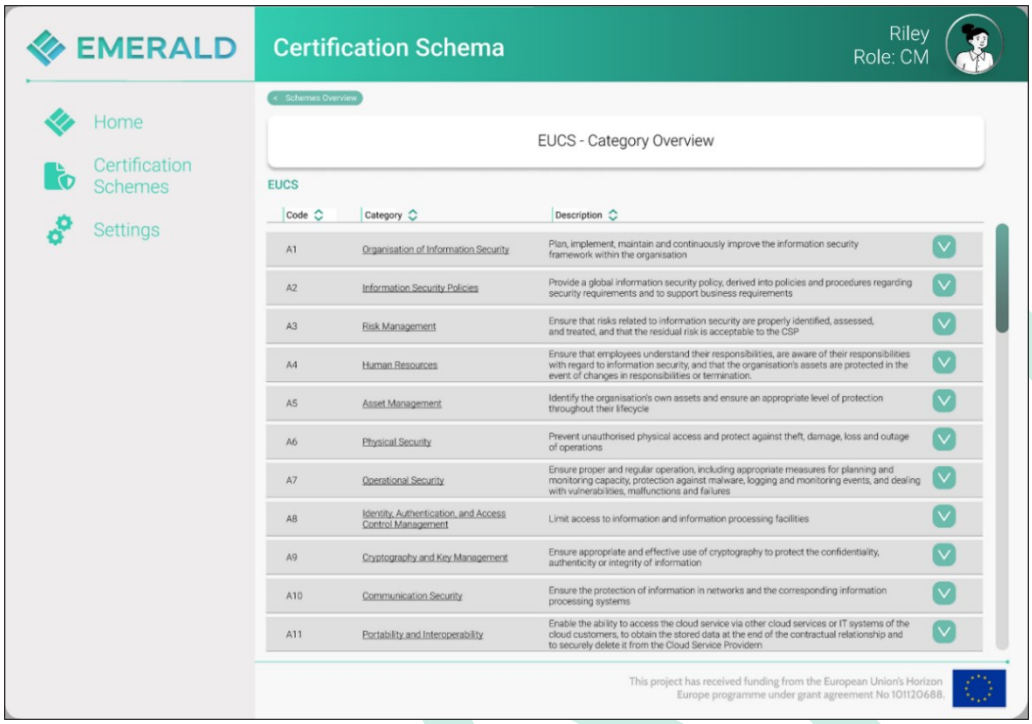

Figure 27. Browse scheme (EUCS categories) (D4.3 [11])

When clicking on one of the high-level categories, a sub-list with its sub-categories are open up below, in a hierarchical way, as presented in Figure 28.

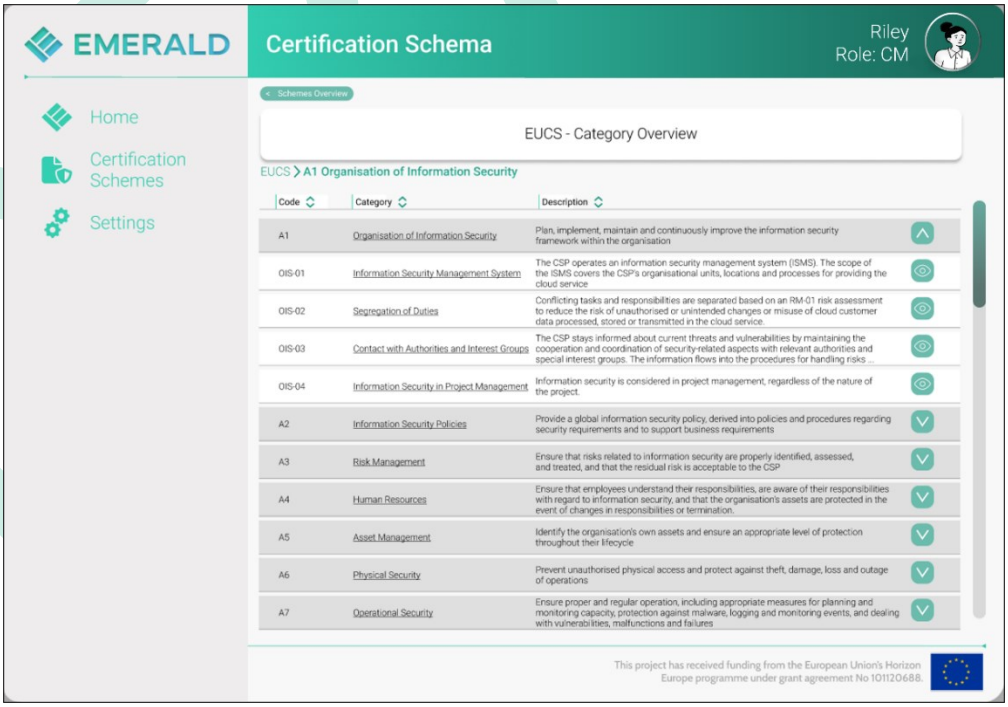

Figure 28. Browse sub-categories of the EUCS scheme (D4.3 [11])

When clicking on one of the categories, the controls that belong to the category are shown, as depicted in Figure 29. For each control, the code, description, assurance level and the assigned metrics are shown.

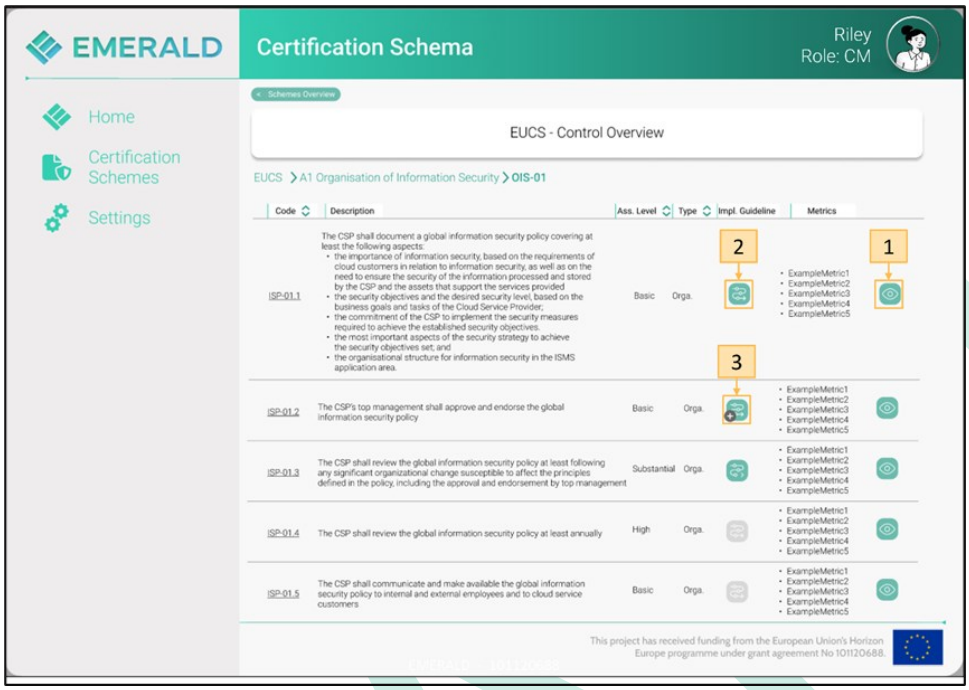

Figure 29. Controls of an EUCS scheme category (D4.3 [11])

8.2.4 Licensing information

The RCM component is offered under Apache 2.0 license. The license files and more detailed information can be found in the EMERALD Public GitLab repository[41].

8.2.5 Download

The code is available at the public GitLab repository of the EMERALD project[41].

[41] https://git.code.tecnalia.com/emerald/public/components/rcm

9 Trustworthiness System

The *Trustworthiness System* (TWS) component provides trustworthiness, fairness and transparency to the evidence and assessment results stored in EMERALD, as the integrity and authenticity of the information is guaranteed.

9.1 Implementation

The implementation of the *TWS* component is based in the MEDINA component *MEDINA Evidence Trustworthiness Management System* [18]. This foundational tool was the baseline for elaborating the specifications of the *TWS*, that are documented in D3.1 [2]. This section presents the functional description and technical description of this component.

9.1.1 Functional description

Blockchain technology is increasingly recognized as a reliable solution for providing trustworthiness, offering a transparent, secure, and cost-effective method for maintaining audit trails. By decentralizing data across a distributed network, Blockchain removes the need for a central authority to manage records. It also ensures the immutability of recorded information, as data is replicated across all the nodes of the Blockchain network. Additionally, each piece of information on the Blockchain is digitally signed by its creator, allowing for traceability.

Despite these important advantages—integrity, decentralization, and non-repudiation—Blockchain's adoption for audit trails remains still limited due to its user-unfriendly nature. Currently, interacting with Blockchain requires a client, which is not usually available on user systems. Therefore, a graphical and web-based tool is recommended to make Blockchain accessible and transparent for external users, such as auditors. This tool would provide a user-friendly interface to verify the integrity of the information recorded on the Blockchain, including evidence and assessment results. With proper authentication, anyone could access this information without needing a Blockchain client.

Thus, the *TWS* offers a secure way for EMERALD to maintain an audit trail of evidence and assessment results. It provides the following functionalities:

- It allows and facilitates the provision of required audit information from the *Assessment* component in EMERALD to the Blockchain.
- It ensures long-term information recording, leveraging Blockchain's inherent advantages like integrity, decentralization, and authenticity.
- It makes use of a general-purpose semi-public Blockchain infrastructure, which supports services in accordance with European regulations.
- It allows external users to access EMERALD audited information in a graphical and user-friendly way.

Table 16 shows the functional requirements satisfied by the current version of the *TWS*, as documented in D3.1 [2], and updates the status of their implementation in the current prototype in M12.

Table 16. TWS Functional Requirements

Req. ID	Description	Priority	Milestone	Progress
TWS.01	Provide a tool allowing the **verification of evidence integrity** without needing to store the evidence itself (for confidentiality reasons).	Must	MS2 (M12)	75% (*)

Req. ID	Description	Priority	Milestone	Progress
TWS.02	Provide a tool allowing the **verification of assessment results integrity** without needing to store the result itself (for confidentiality reasons).	Must	MS2 (M12)	75% (*)
TWS.03	The integrity validation of evidence and assessment results must be done **through REST API or graphical interface (EMERALD UI)**.	Must	MS5 (M24)	50%
TWS.04	The TWS must be **based on a real Blockchain network, with multiple nodes and multiple organizations** to guarantee suitable decentralization and governance of the Blockchain network.	Must	MS5 (M24)	5%

(*) Note that TWS.01 and TWS.02 requirements should have been fully completed by M12. However, due to a high delay on the feedback in the EBSI early adopters submission (see Section 9.1.2.1.1 for further details), although the current progress is almost completed, it is not totally completed as a new Blockchain ecosystem needs to be selected, and the functionalities of TWS.01 and TWS.02 also need to be adapted to the new Blockchain ecosystem.

9.1.1.1 Fitting into overall EMERALD Architecture

TWS is one of the components of the EMERALD architecture. Figure 30 shows the interaction with other components in the EMERALD framework.

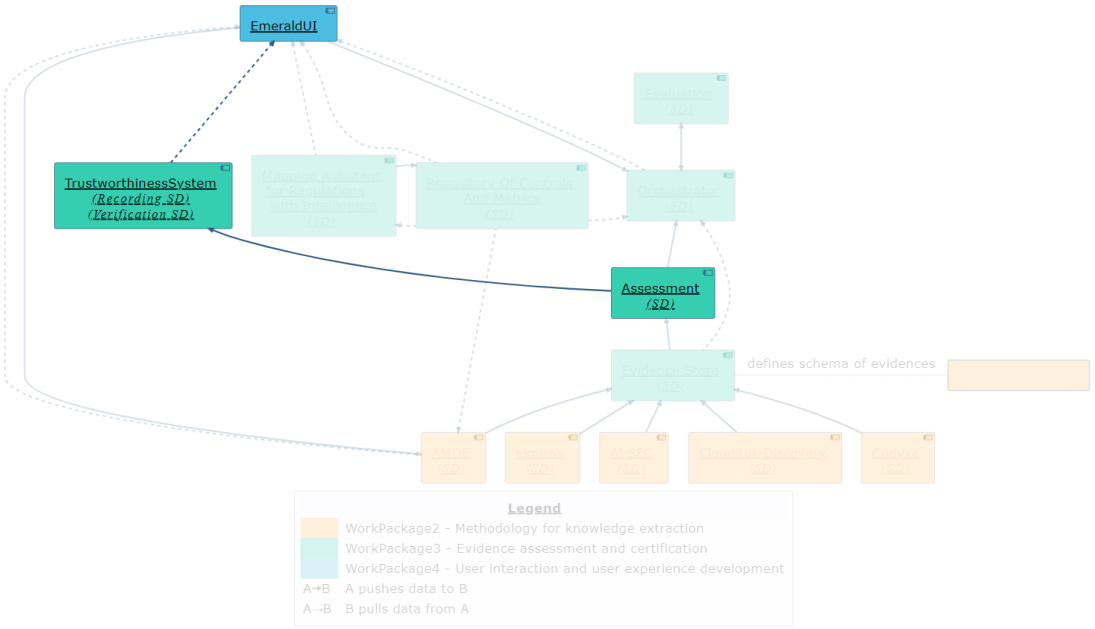

Figure 30. Fitting of the TWS with other components in EMERALD architecture

The main interactions of the *TWS* with the EMERALD components are as follows:

- ***Clouditor-Assessment*:** The interaction with this component occurs in two ways:
  - The *Assessment* component provides information related to evidence and assessment results to be recorded on the Blockchain.
  - The automatic verification service requests the current values of evidence and assessment results stored in EMERALD's internal evidence storage to validate their integrity against the information previously recorded on the Blockchain.
- **EmeraldUI:** The graphical interface of the *TWS* automatic verification service is integrated into the EMERALD UI, allowing auditors to easily verify the trustworthiness of evidence and assessment results, and determine their reliability.

9.1.2 Technical description

This section is dedicated to describing the technical specification of the *TWS* component. First, the main architecture of the prototype is presented, including all its sub-components. Next, the technical specifications of the developed system are described. The section finishes with the description of the published APIs.

9.1.2.1 Prototype architecture

Figure 31 shows the architecture of the Blockchain-based *TWS* [2].

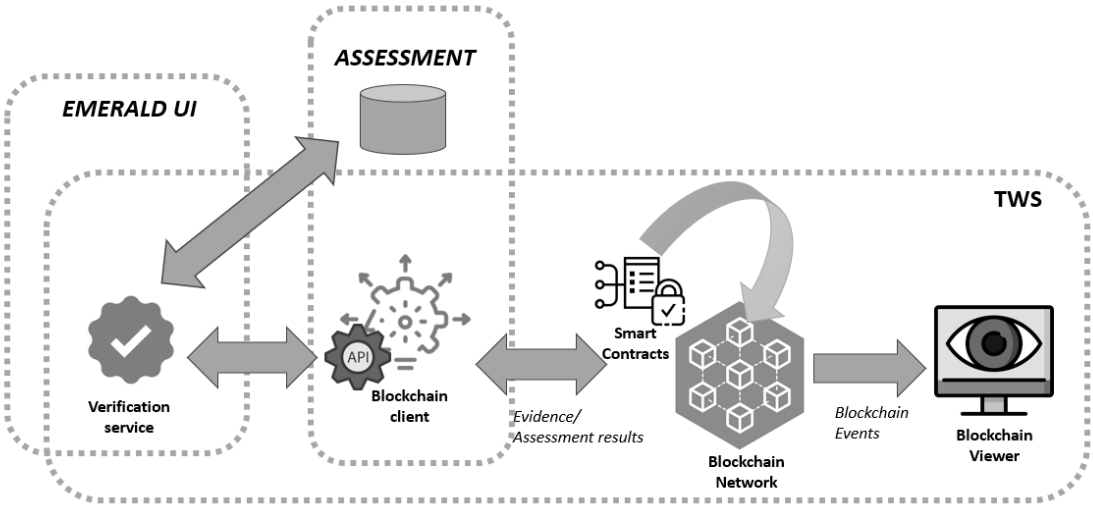

Figure 31. TWS architecture

The architecture is composed of five main elements:

- **Blockchain Network:** A general-purpose semi-public Blockchain infrastructure will be considered for the *TWS*. From MEDINA, a privately deployed Quorum network was considered as a proof of concept. However, EMERALD plans to make a production deployment in a real Blockchain infrastructure. Initially, the European Blockchain Service Infrastructure (EBSI) was considered. However, EBSI is currently in the process of transition to a new legal entity and the participation in the early adopters' programme that EMERALD requested at the beginning of the project is currently close (more details in Section 9.1.2.1.1). Other options are currently under analysis by the EMERALD partners.
- **Smart Contracts:** The auditing functionalities of the *TWS* are implemented through Smart Contracts to be deployed on the Blockchain network. These Smart Contracts handle the registration of data (evidence and assessment results) on the Blockchain and its consumption for integrity validation. They also generate Blockchain-based events to feed the Blockchain viewer and allow user-friendly access to the information recorded on the *TWS*.
- **Blockchain Client:** Each assessment component of the EMERALD architecture will have a Blockchain client to interact with the Blockchain and the Smart Contracts, managing wallets and generating transactions for registering and consumption operations.
- **Blockchain Viewer:** This subcomponent listens to Blockchain events from the Smart Contracts, normalizes and categorizes the details for display on a dashboard. It allows external users to access information recorded on the Blockchain without needing a Blockchain client.

- **Automatic Verification Service:** This tool automatically checks the validity of currently available evidence and assessment results in EMERALD in comparison to the information previously recorded on the Blockchain. It provides auditors a user-friendly interface integrated into the EMERALD UI for easy data integrity verification.

9.1.2.1.1 Components description

The following paragraphs describes in more detail the *TWS* components.

Blockchain Network

The *TWS* is based on a Quorum network, based on the implementation from MEDINA. However, different types of Blockchain networks are currently under consideration in order to be able to deploy the *TWS* in a general-purpose semi-public-permissioned Blockchain ecosystem.

The first idea was to deploy the *TWS* in the European Blockchain Services Infrastructure (EBSI) [22]. EBSI is a Blockchain network of distributed nodes across Europe. It is the first EU-wide Blockchain infrastructure, driven by the public sector to support cross-border applications such as traceability, verifiable credentials, trusted data exchange (as it could be the case of the *TWS*) or IP management across multiple domains. Additionally, EBSI considers Hyperledger Besu[42], which is an Ethereum client suitable for both public and private permissioned Blockchain network use cases. It is also EVM (Ethereum Virtual Machine) compatible, which signifies that EBSI can support the deployment of Smart Contracts written in Solidity[43], as it is the case of the *TWS*. For these reasons, EBSI was considered a suitable Blockchain ecosystem to deploy the *TWS*.

Although EBSI is still under development, it offers an early adopters' programme where some pilot projects can be launched. EMERALD made a request to take part in the early adopters' programme (contribution ID 64e2738a-0a70-4388-888f-ef108424edb3). However, in September 2024, the request was closed. The reason EBSI provided is literally the following one:

*Unfortunately, the **Early Adopters Programme (EAP) is now closed** and therefore we are not onboarding any projects to the programme at this moment. We will be in contact with you in case of any developments regarding the programme.*

*EBSI is evolving and in May 2024 a group of member states came together and created the **Europeum-EDIC**, a new legal entity which will enable EBSI to go into production and deepen the collaboration on Web3 technologies. EBSI is currently in the **process of transition** to this legal entity who will oversee the EAP. You can read more about the EDIC's goals and how they align with the programme here.*

As a result, other alternatives have recently started to be considered for the *TWS*. The two main requirements for the Blockchain ecosystem to be considered are:

- It should be a general-purpose public-permissioned ecosystem.
- It must be compatible with the current implementation of the *TWS*, based on Solidity. Moreover, it is highly recommended to consider Hyperledger Besu to keep

[42] https://besu.hyperledger.org/

[43] https://soliditylang.org/

compatibility with EBSI in case a European-wide deployment in such an ecosystem is possible in the future.

Alastria [23] is the option being further analysed. Alastria is the first Spanish and one of the world's largest public-permissioned multisector blockchain platforms, bringing together companies, academia, and public administration. It includes different Blockchain networks, including a solution based on Hyperledger Besu. More details will be provided in the next versions of the component.

Smart Contracts

The Smart Contracts deployed on the Blockchain network encompass the functionalities of *TWS* requiring high levels of security, including functionalities for the different users. There are two types of users for the *TWS*:

- Administrators: These users have the authority to authorize or de-authorize access to the *TWS*. They can also appoint new administrators or remove existing ones. By default, TECNALIA, as component owner, is the administrator of the *TWS*. However, when the component is deployed on the pilots' premises, they should take this role.
- Authorized users: They refer to the assessment components which are authorized by an existing administrator to use the *TWS*. What is collected by each assessment component can only be accessed by its owner. Although the Blockchain network is shared among several authorized users, each one can only access information related to his own registered assessment component.

Each user (administrators and authorized users) is identified in the *TWS* through a Blockchain address.

Administrators' Functionalities

Administrators have exclusive access to general information about the current status of the *TWS*. Only administrators can perform actions related to the user management.

- Register a new administrator.
- Remove an existing administrator.
- Check if a specific user is an administrator.
- Retrieve the total number of administrators in the system.
- Authorize a new user.
- De-authorize an existing user.
- Check if a specific user is authorized in the system.
- Retrieve the total number of authorized users in the system.
- Obtain all registered user IDs (administrators can only see the user ID, not the information provided by the user itself).

Authorized users' Functionalities

If users are not authorized, they will not be able to access the system due to restrictions enforced by the Smart Contracts design. To gain access, they must submit a "registration request" to notify administrators of their intent to use the TWS. This request should include the user identifier (Blockchain address). Upon receiving the notification, administrators will manually review the request to determine if the user should be authorized.

Once authorized, they can register assessment components, providing the following information:

- id: This is the internal ID used to identify the assessment component within *TWS*. It is automatically generated by the Smart Contract based on the authorized user unique Blockchain address.
- owner: This refers to the authorized user Blockchain address who has registered the assessment component. It is automatically obtained by the Smart Contract.
- timestamp: This indicates the timestamp in seconds since the epoch of the assessment component registration process. It is automatically obtained by the Smart Contract.

Once an assessment component is registered, the functionalities of the authorized user are:

- Retrieving the registered assessment component ID.
- Retrieving the associated assessment component owner (authorized user) ID.
- Retrieving the registered assessment component registration timestamp.
- Adding new evidence information following the trustworthy evidence data model shown in Figure 32. This data model is the first version that could be updated as required in the following versions of the *TWS*.
- Retrieving specific evidence information.
- Retrieving all added evidence IDs associated to a given assessment component.
- Adding new assessment result information following the trustworthy assessment result data model shown in Figure 32. This data model is the first version that could be updated as required in the following versions of the *TWS*.
- Retrieving specific assessment result information.
- Retrieving all added assessment result IDs associated to a given assessment component.
- Checking the integrity validity of a specific evidence.
- Checking the integrity validity of a specific assessment result.
- Checking the integrity validity of a specific assessment compliance result.

```
address id;
address owner;
uint256 creationTimestamp;

uint256[] evidencesIds;
struct Evid {
    uint256 id;
    bytes32 valueHash;
    uint256 toolId;
    uint256 resourceId;
    uint256 cspId;
    uint256 timestamp;
}
mapping(uint256 => Evid) public evid;

uint256[] assessmentsIds;
struct Assess {
    uint256 id;
    bytes32 securityAssessmentHash;
    bytes32 complianceHash;
    uint256[] associatedEvidencesId;
    uint256 metricId;
    uint256 timestamp;
}
mapping(uint256 => Assess) public assess;
```

Figure 32. TWS Data model

Events Generation

Every time an operation is executed in the Smart Contracts, a Blockchain-based event is generated to feed the Blockchain viewer. Initially, the events to be generated include (but are not limited to):

- Registration of new administrators.
- Removal of an existing administrator.

- Authorization of a new user in the system.
- De-authorization of an existing authorized user in the system.
- Registration of a new assessment component by its owner (authorized user).
- Registration of new evidence information.
- Registration of new assessment result information.

In this first version, there are no events related to reading/retrieving actions.

Blockchain client

The functionalities of the EMERALD TWS are implemented through Smart Contracts that need to be invoked by a Blockchain client. In the initial version, the main functionalities of the Blockchain client are as follows:

Blockchain Account Management

Each assessment component interacting with the *TWS* requires a Blockchain account. This account consists of a Blockchain address, which uniquely identifies the user within the Blockchain network, and an associated private key, known only by the assessment component and securely kept. The Blockchain account is securely managed by means of a "wallet" included in the Blockchain client, simplifying user interaction with the Blockchain network.

The functionalities available in the Blockchain client related to Blockchain accounts include:

- **Create a new Blockchain account**: Automatically generates a new Blockchain address and its associated private key.
- **Get the address associated with a specific private key**: Obtains the address from a private key; for validation purposes.
- **Add a specific Blockchain account to the Blockchain client wallet**: The private key is needed to store the Blockchain account in the internal wallet of the Blockchain client. Initially, only one account can be stored.
- **Get the Blockchain address added to the wallet**: Retrieves the Blockchain address information previously added to the Blockchain wallet for validation purposes.
- **Request authorization**: Ask the EMERALD administrators to provide rights for a specific Blockchain address (associated with a specific assessment component) to use the *TWS* (authorization functionality from administrators).

Blockchain Transactions Creation

The assessment component needs to generate Blockchain transactions to send them to the Blockchain and be understood by the Smart Contracts deployed on it. The Blockchain client automatically creates the required Blockchain transactions for executing all functionalities available in the EMERALD *TWS* Smart Contracts, using the Web3.js library internally.

API REST for External Interaction

The Blockchain client exposes an API REST to allow assessment components to easily interact with the Blockchain client for their account management as well as for the provision of evidence and assessment results to be recorded on the *TWS*.

Additionally, the automatic verification service described in Section 9.1.2.1.1 interacts with the Blockchain client API to obtain the evidence and assessment results recorded in the Blockchain, necessary for the automatic verification of the integrity of the current evidence and assessment results available at EMERALD at a given time.

API description

All API endpoints are listed below with a brief explanation:

- *Account:* This endpoint is related to the Blockchain account (address and private key) management (create, add account to wallet, get account from wallet, get address from private key).
- *Registration*: This endpoint refers to an authorization request in the TWS.
- *Admin*: This endpoint refers to the TWS management as administrator. Administrators can be created, updated, listed and removed; users can be authorized or deauthorized; the total number of administrators and authorized users can be listed; the registered assessment components can be listed.
- *Assessments*: This endpoint refers to the management of the assessment results proofs of integrity (register, get, check).
- *Evidence*: This endpoint refers to the management of the evidence proofs of integrity (register, get, check).

Blockchain viewer

The Blockchain viewer monitors Blockchain-based events generated by the Smart Contracts, providing notifications about new users in the system, as well as new evidence or assessment results recorded in the *TWS*. It offers a mechanism for external users (such as auditors or security engineers) to manually verify evidence and assessment results recorded on the Blockchain. Figure 33 illustrates the internal architecture of the Blockchain viewer.

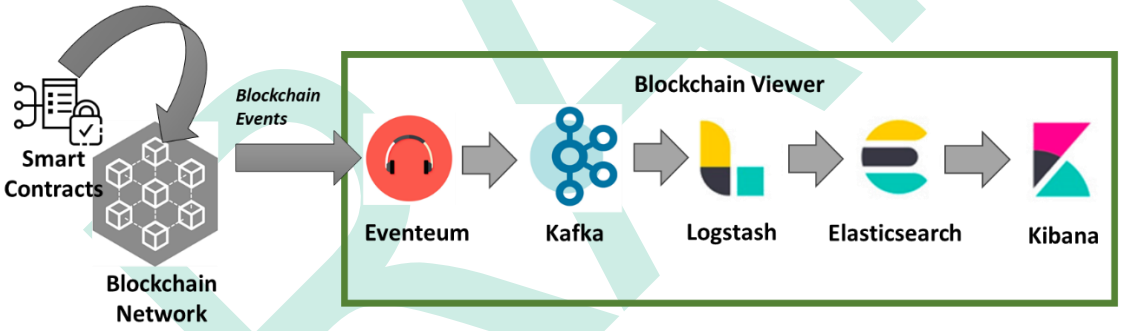

Figure 33. TWS Blockchain viewer architecture

The Blockchain viewer consists of five components:

- **Eventeum**[44]: This component bridges the Smart Contracts deployed in the Blockchain with the Blockchain Viewer. As explained in Section 9.1.2.1.1, Smart Contracts automatically generate Blockchain events that Eventeum listens to. To listen to the events, it is necessary to subscribe to events from specific Smart Contract addresses (each Smart Contract has a unique Blockchain address). Additionally, the format of the specific events must be indicated to Eventeum (event id, parameters order, parameters type).
- **Apache Kafka**[45]**:** This intermediate platform distributes Blockchain events between Eventeum and Logstash. Kafka uses message queues to provide asynchronous communication, meaning the sender (Eventeum) and the receiver (Logstash) do not need to interact with the message queue simultaneously.

[44] https://github.com/eventeum/eventeum

[45] https://kafka.apache.org/

- **Logstash**[46]: A log management tool used in the Blockchain viewer to collect all events received from Eventeum via Kafka queues. It normalizes these events into a common format before routing them to Elasticsearch for processing.
- **Elasticsearch**[47]: A distributed search and analysis engine that stores, indexes, and processes the information from the events. The information stored in Elasticsearch can be recreated from scratch in case of a security incident, ensuring a fully reliable source of information.
- **Kibana**[48]: Kibana is a graphical interface that displays information from Elasticsearch in real time through customized dashboards. Access to Kibana dashboards requires authentication. Different roles need to be created to access different types of information in the *TWS*. For example, administrators should have access to all registered evidence and assessment results from different assessment components. In contrast, each authorized assessment component should have limited access only to its associated evidence and assessment results to prevent information disclosure. Additionally, different dashboards will need to be created for the different roles. For example, Figure 34 and Figure 35 show two examples of dashboards for administrators and assessment components. These dashboards are just an example and will be updated for the next version of the *TWS* according to the EMERALD requirements.

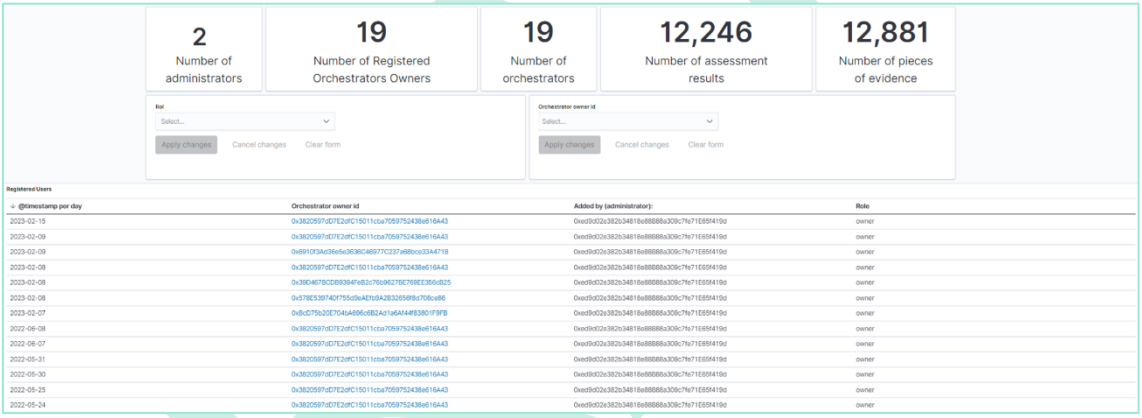

Figure 34. TWS Blockchain viewer dashboard for administrators

[46] https://www.elastic.co/es/logstash
[47] https://www.elastic.co/es/elasticsearch
[48] https://www.elastic.co/es/kibana

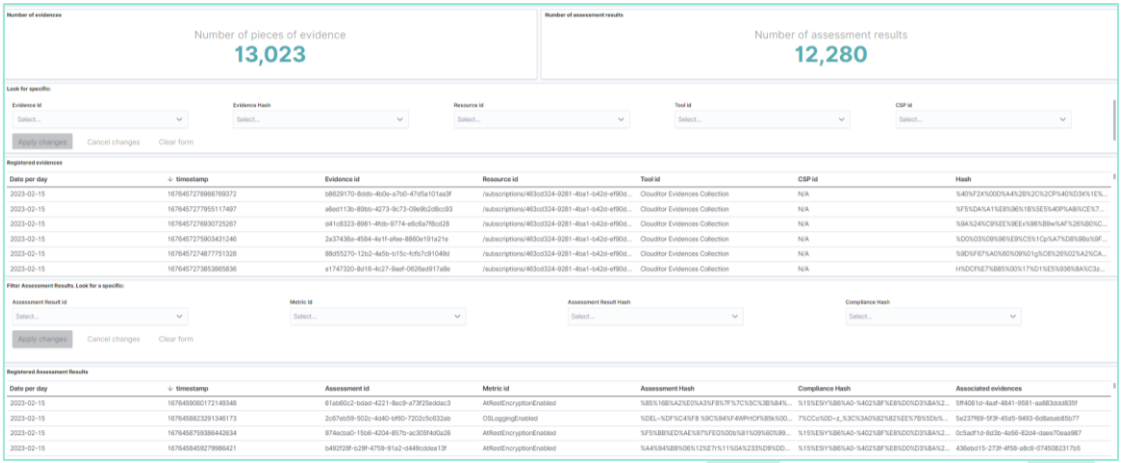

Figure 35. TWS Blockchain viewer dashboard for assessment components

Automatic verification service

Auditors require an automated method to verify the integrity of evidence and assessment results, in addition to the manual access to hashes and additional information offered through the Blockchain viewer. The automatic verification service meets this need by offering a graphical tool that validates evidence and assessment results from the EMERALD evidence storage against Blockchain records. Thus, this service ensures the integrity of evidence and assessment results. The verification service will be integrated in the EMERALD UI allowing automatic integrity verifications. More details about the way to be integrated are provided in D4.3 [11].

9.1.2.2 Technical specifications

The *TWS* is a software solution deployable at both Windows and Linux operating systems as long as they have hardware virtualization and docker support. It has been implemented using Solidity for the Smart Contracts, Javascript for the Blockchain client, React and Nodejs for the verification service and Go and Scripting for the Blockchain monitor.

9.2 Delivery and usage

This section describes the information needed for the installation and use of the *TWS.* Besides, it also details the licensing information and related packages and repositories.

9.2.1 Package information

The *TWS* is composed of the following packages:

- TWS.sol and Assessment.sol. These are the Smart Contracts that need to be deployed on the selected Blockchain network.
- One docker of the Blockchain client, to be deployed on the EMERALD infrastructure associated to the assessment component.
- Two dockers of the verification service (one for the backend, and one for the frontend), to be deployed on the EMERALD infrastructure.
- Eight dockers of the Blockchain viewer (Oauth2 proxy, eventeum, mongodb, zookeeper, kafka, Logstash, Elasticsearch, kibana) to be deployed on the EMERALD infrastructure (or as a service from other infrastructure).

9.2.2 Installation

The deployment of the Smart Contract will depend on the specific Blockchain network to be considered for the *TWS*.

For the rest of the services, as they have been dockerized, the installation is as follows:

- docker run {docker_image}, for the Blockchain client and the verification service.
- docker-compose up, for the Blockchain monitor (as a docker-compose.yml file has been created).

9.2.3 Instructions for use

Blockchain client

The *Assessment* component from EMERALD is the component that uses the Blockchain client to provide evidence and assessment results to the *TWS*. Once the Docker image is running (after following the installation steps in Section 9.2.2), the user needs to:

1. Generate and Add a Blockchain Account

Generate a Blockchain account and add it to the Blockchain wallet (inside the Blockchain client) through a POST request to the `/client/account` endpoint and a POST request to the `/client/wallet` endpoint of the Blockchain client API, respectively.

2. Request Authorization:

Request authorization for the user through a POST request to the `/client/registration` endpoint of the Blockchain client API. The administrators (initially, TECNALIA) will authorize the Blockchain account (authorized user).

3. Register the assessment component:

Once the user is authorized, register the assessment component through a POST request to the `/client/assessment` endpoint. From this point, all authorized users' functionalities from the *TWS* can be executed. Refer to Section 9.1.2.1.1 for available functionalities: providing evidence or assessment results, retrieving the list of registered evidence or assessment result IDs, retrieving information of a specific evidence or assessment result ID, and checking the integrity of specific evidence or assessment results.

Blockchain viewer

In EMERALD, the verification service will be integrated in the EMERALD UI. There are two steps in the use of the *TWS* verification service:

1. Set-up of the component

It is necessary to define when and how often the *TWS* should be automatically updated and/or on demand, as shown in Figure 36.

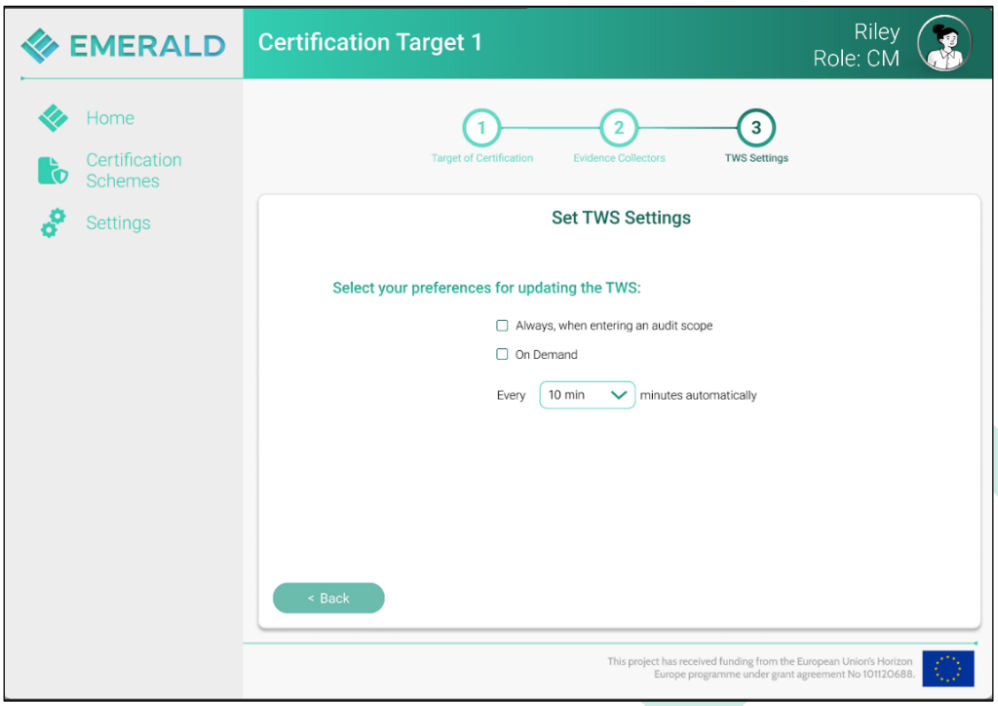

Figure 36. TWS set-up (D4.3 [11])

2. Integrity check

The status of the integrity check is always visible at the upper right side of the EMERALD UI.

- If the integrity check of all evidence and assessment results is ok, the TWS status symbol is presented in green as shown in Figure 37.

Figure 37. Correct integrity verification

- If the integrity check is not ok, the TWS status symbol is presented in red as shown in Figure 38. Additionally, it is possible to get a report with the details of the modified evidence as shown in Figure 39.

Figure 38. Incorrect integrity verification

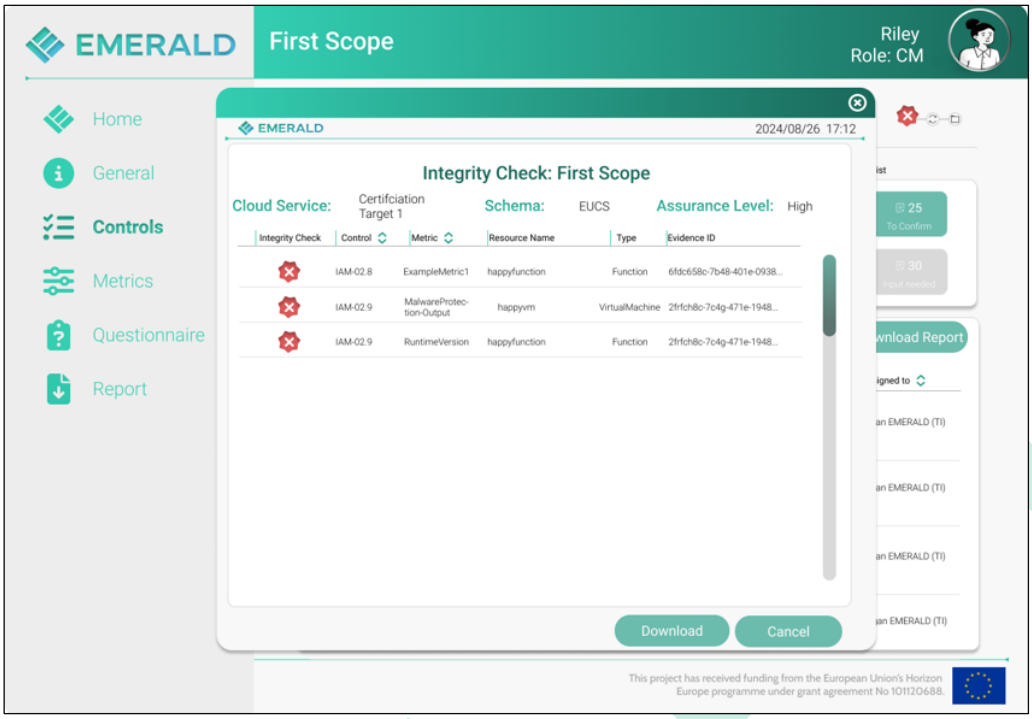

Figure 39. Integrity verification details (D4.3 [11])

More details on the graphical interface are provided in D4.3 [11].

9.2.4 Licensing information

Proprietary. Copyright by TECNALIA.

9.2.5 Download

This section is not applicable as TECNALIA owns a proprietary license, so no source code can be provided.

10 Conclusions

In this deliverable, we have provided a comprehensive overview of the initial implementation of the WP3 components within the EMERALD project. This includes detailed functional and technical descriptions, delivery and usage instructions, and associated documentation for each component: *Clouditor-Orchestrator*, *Clouditor-Assessment*, *Clouditor-Evidence Store*, *Mapping Assistant for Regulations with Intelligence* (MARI), *Clouditor-Evaluation*, *Repository of Controls and Metrics* (RCM), and *Trustworthiness System* (TWS).

The primary goal of this deliverable is to document the implementation of the WP3 components, ensuring that they are effectively integrated and operational within the EMERALD framework. By achieving this, we aim to facilitate the development of a Certification-as-a-Service (CaaS) framework for continuous certification of harmonized cybersecurity schemes.

The key contributions of this deliverable include the initial implementation of the WP3 components, addressing key results such as CERTGRAPH (KR2), OPTIMA (KR3), MULTICERT (KR4), and INTEROP (KR7). This progress is measured using the key performance indicators (KPIs) defined in the DoA [1].

Looking ahead, the next steps involve further development and refinement of the WP3 components. This includes the interim integration of the components within the overall EMERALD system (D3.5 [3], M15), thereby completing the first iteration of concepts, implementation and integration of WP3 components. Following this, the final versions of the concepts (D3.2 [4], M18), implementation (D3.4 [5], M24) and integration (D3.6 [6], M27) will be completed. These steps will ensure continuous improvement and alignment with the project's objectives, ultimately enhancing the robustness and effectiveness of the EMERALD framework.

11 References

[1] EMERALD Consortium, “EMERALD - Annex 1- Description of Action - GA101120688,” 2022.

[2] EMERALD Consortium, “D3.1 Evidence assessment and Certification–Concepts-v1,” 2024.

[3] EMERALD Consortium, “D3.5 Evidence assessment and Certification - Integration -v1,” 2025.

[4] EMERALD Consortium, “D3.2 Evidence assessment and Certification Concepts - v2,” 2025.

[5] EMERALD Consortium, “D3.4 Evidence Assessment and Certification - Implementation-v2,” 2025.

[6] EMERALD Consortium, “D3.6 Evidence assessment and Certification - Integration- v2,” 2026.

[7] ENISA, “EUCS - Cloud Services Scheme,” [Online]. Available: https://www.enisa.europa.eu/publications/eucs-cloud-service-scheme. [Accessed October 2024].

[8] B. e. al., A Semantic Evidence-based Approach to Continuous Cloud Service Certification, Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing, 2023.

[9] MEDINA Consortium, “D3.6 Tools and techniques for collecting evidence of technical and organisational measures - v3 (https://medina-project.eu/public-deliverables/),” 2023.

[10] MEDINA Consortium, “D4.3 Tools and techniques for the management and evaluation of cloud security certifications-v3 (https://medina-project.eu/public-deliverables/),” 2023.

[11] EMERALD Consortium, “D4.3 User interaction and user experience concept–v1,” 2024.

[12] EMERALD Consortium, “D2.1 Graph Ontology for Evidence Storage,” 2024.

[13] MEDINA Consortium, “D3.5 Tools and techniques for collecting evidence of technical and organisational measures-v2 (https://medina-project.eu/public-deliverables/),” 2022.

[14] EMERALD Consortium, “D2.2 Source Evidence Estractor -v1,” 2024.

[15] EMERALD Consortium, “D2.4 AMOE-v1,” 2024.

[16] EMERALD Consortium, “D2.6 ML model certification-v1,” 2024.

[17] EMERALD Consortium, “D2.8 Runtime evidence extractor - v1,” 2024.

[18] MEDINA Consortium, “D3.3 Tools and techniques for the management of trustworthy evidence-v3 (https://medina-project.eu/public-deliverables/),” 2023.

[19] MEDINA Consortium, “D2.3 Specification of the Cloud Security Certification Language-v3 (https://medina-project.eu/public-deliverables/),” 2023.

[20] EMERALD Consortium, “D1.3 EMERALD solution architecture-v1,” 2024.

[21] MEDINA Consortium, “D2.2 Continuously certifiable technical and organizational measures and catalogue of cloud security metrics-v2 (https://medina-project.eu/public-deliverables/),” 2023.

[22] European Comission, “European Blockchain Services Infrastructure (EBSI),” 2024. [Online]. Available: https://ec.europa.eu/digital-building-blocks/sites/display/EBSI/Home. [Accessed 10 2024].

[23] Alastria, “Where Blockchain happens,” 2024. [Online]. Available: https://alastria.io/en/home/. [Accessed 10 2024].

APPENDIX A: Examination of Graph DB Engines

The following graph databases are potential candidates for using in the EMERALD framework, providing Go compatibility and including appropriate licences (e.g. Apache 2.0):

ArangoDB (https://arangodb.com/)
Apache AGE (https://age.apache.org/)
Dgraph (https://dgraph.io/)
Memgraph (https://memgraph.com/)
Nebula (https://www.nebula-graph.io/)
Neo4j (https://neo4j.com/)
dylanpaulus (https://www.dylanpaulus.com/posts/postgres-is-a-graph-database/)
RedisGraph (https://redis.io/docs/stack/graph/)
SurrealDB (https://surrealdb.com/)
TinkerGraph (https://tinkerpop.apache.org/docs/current/reference/#tinkergraph-gremlin)
ArcadeDB (https://arcadedb.com/)

The following graph databases will be not considered due to limitations either in the Go compatibility or in the provided licence:

Aerospike (https://aerospike.com/)
Dedicated Cloud Services (e.g., https://azure.microsoft.com/services/cosmos-db or https://aws.amazon.com/neptune/)
Janus Graph (https://janusgraph.org/)
GraphDB (https://www.ontotext.com/)
OrientDB (https://orientdb.org/)
Stardog (https://www.stardog.com/)
TigerGraph (https://www.tigergraph.com/)
Virtuoso (https://virtuoso.openlinksw.com/)
Fauna (https://fauna.com/)
Giraph (https://giraph.apache.org/)
AllegroGraph (https://allegrograph.com/)
Blazegraph (https://blazegraph.com/)
TypeDB (https://typedb.com/)
Graph Engine (https://www.graphengine.io/)
InfiniteGraph (https://objectivity.com/infinitegraph/)
Fluree (https://flur.ee/)
AnzoGraph (https://cambridgesemantics.com/anzograph/)
AgensGraph (https://bitnine.net/agensgraph/)
TerminusDB (https://terminusdb.com/)
FlockDB (https://github.com/twitter-archive/flockdb)
HyperGraphDB (https://hypergraphdb.org/)
Ultipa (https://www.ultipa.com/)
Apache HugeGraph (https://hugegraph.apache.org/)
Bangdb (https://bangdb.com/)
GraphBase (https://graphbase.ai/)
gStore (https://en.gstore.cn/)
TinkerGraph (https://tinkerpop.apache.org/docs/current/reference/#tinkergraph-gremlin)
Sparksee (https://www.sparsity-technologies.com/)
VelocityDB (https://velocitydb.com/)

Transwarp StellarDB (https://www.transwarp.cn/en/product/stellardb)
Galaxybase (https://galaxybase.com/)
HGraphDB (https://github.com/rayokota/hgraphdb)
Kuzu (https://kuzudb.com/)