Deliverable D3.6

Tools and techniques for collecting evidence of technical and organisational measures – v3

Editor(s):	Hrvoje Ratkajec
Responsible Partner:	XLAB
Status-Version:	v1.0
Date:	05.05.2023
Distribution level (CO, PU):	PU

Project Number:	952633
Project Title:	MEDINA

Title of Deliverable:	Tools and techniques for collecting evidence of technical and organisational measures – v3
Due Date of Delivery to the EC	30.04.2023

Workpackage responsible for the Deliverable:	WP3 – Tools to gather evidences for high-assurance cybersecurity certification
Editor(s):	Hrvoje Ratkajec (XLAB)
Contributor(s):	Immanuel Kunz, Florian Wendland (FhG), Franz Deimling (Fabasoft)
Reviewer(s):	Björn Fanta (Fabasoft), Olivia Kagerer (Fabasoft) Cristina Martinez (TECNALIA)
Approved by:	All Partners
Recommended/mandatory readers:	WP3, WP4, WP5, and WP6

Abstract:	This deliverable presents tools and techniques for the evidence collection of technical measures, such as security assessment of virtual machines, containers and server less functions or based on the analysis of information and data flows as well as organisational measures through the use of machine-learning and NLP. This is the third and final iteration of the tool integration, based on a refinement of the technical architecture and reflects the feedback from the validation at use cases. This deliverable is the result of Task 3.2, Task 3.3 and Task 3.4.
Keyword List:	Evidence gathering, Security assessment, Technical measures, Organisational measures, Components implementation, Clouditor, Codyze, Wazuh, Vulnerability Assessment Tools, Cloud Property Graph, LLVM Extensions of the Code Property Graph, Assessment and Management of Organisational Evidence
Licensing information:	This work is licensed under Creative Commons Attribution-ShareAlike 3.0 Unported (CC BY-SA 3.0) http://creativecommons.org/licenses/by-sa/3.0/
Disclaimer	This document reflects only the author's views and neither Agency nor the Commission are responsible for any use that may be made of the information contained therein

Document Description

Version	Date	Modifications Introduced	
		Modification Reason	Modified by
v0.1	08/03/2023	First draft version - ToC	Hrvoje Ratkajec (XLAB)
v0.2	17/03/2023	Added new section 4.2, added and updated the contents in section 2 and 3.1	Immanuel Kunz (FhG)
v0.3	20/03/2023	Added and updated the contents in sections 3.1, 4.1 and 4.3	Immanuel Kunz (FhG)
v0.4	22/03/2023	Added contents to sections 3.2 and 3.3.	Hrvoje Ratkajec (XLAB)
v0.5	29/03/2023	Updated contents in section 5 and in Appendix E	Franz Deimling (Fabasoft)
v0.6	03/04/2023	Added component cards to sections 3.1, 3.2, 3.3, 4.3 and 5, added Appendix F	Hrvoje Ratkajec (XLAB)
v0.7	06/04/2023	Review of changes from other contributors, updated Executive Summary and Conclusions, formatting	Hrvoje Ratkajec (XLAB)
v0.8	20/04/2023	Internal review contents added	Björn Fanta (Fabasoft), Olivia Kagerer (Fabasoft)
v0.9	03/04/2023	Addressing comments from the internal review	Hrvoje Ratkajec (XLAB), Immanuel Kunz, Florian Wendland (FhG), Franz Deimling (Fabasoft)
v1.0	05/05/2023	Ready for submission	Cristina Martínez (TECNALIA)

Table of contents

Terms and abbreviations 7

Executive Summary 9

1 Introduction 10

1.1 About this deliverable 10

1.2 Document Structure 10

1.3 Updates from D3.5 11

2 Evidence Management Tools High-level Architecture 13

3 Security Assessment of Cloud Infrastructure 15

3.1 Clouditor-Based Components 15

3.1.1 Implementation 15

3.1.2 Delivery and usage 29

3.1.3 Advancements within MEDINA 30

3.1.4 Limitations and future work 31

3.2 Wazuh 32

3.2.1 Implementation 32

3.2.2 Delivery and usage 41

3.2.3 Advancements within MEDINA 43

3.2.4 Limitations and future work 43

3.3 Vulnerability Assessment Tools 45

3.3.1 Implementation 45

3.3.2 Delivery and usage 50

3.3.3 Advancements within MEDINA 52

3.3.4 Limitations and future work 52

4 Security Assessment of Cloud Applications 54

4.1 Cloud Property Graph 54

4.1.1 Implementation 54

4.1.2 Delivery and usage 57

4.1.3 Advancements within MEDINA 57

4.1.4 Limitations and future work 58

4.2 LLVM Extensions of the Code Property Graph 59

4.2.1 Implementation 59

4.2.2 Delivery and usage 61

4.2.3 Advancements within MEDINA 61

4.2.4 Limitations and future work 61

4.3 Codyze 63

4.3.1 Implementation ... 63
4.3.2 Delivery and usage ... 69
4.3.3 Advancements within MEDINA ... 69
4.3.4 Limitations and future work ... 70
5 Assessment of Organisational Measures ... 71
5.1 Implementation ... 71
5.1.1 Functional description ... 71
5.1.2 Technical description ... 75
5.2 Delivery and Usage ... 77
5.2.1 Package ... 77
5.2.2 Installation ... 77
5.2.3 User Manual ... 78
5.2.4 Licensing ... 78
5.2.5 Download ... 78
5.3 Advancements within MEDINA ... 78
5.4 Limitations and future work ... 80
6 Conclusions ... 82
7 References ... 83
8 Appendix A: MEDINA Requirements Implementation Overview ... 85
9 Appendix B: Clouditor - Readme, Installation instructions and User manual ... 88
9.1 README ... 88
9.2 Installation instructions ... 89
9.3 User Manual ... 90
10 Appendix C: Codyze - Installation instructions and User manual ... 91
10.1 Installation instructions ... 91
10.2 User Manual ... 91
11 Appendix D: Cloud Property Graph - Installation instructions and User manual ... 92
11.1 Installation instructions ... 92
11.2 User Manual ... 92
12 Appendix E: AMOE User Manual ... 93
13 Appendix F: Wazuh and VAT Evidence Collector - Readme and installation instructions .. 98
13.1 Readme ... 98
13.2 Installation instructions & use ... 99

List of tables

Table 1. Overview of deliverable updates with respect to D3.5 11
Table 2. Component card for the Cloud Evidence Collector 17
Table 3. Component card for the Security Assessment 18
Table 4. Component card for the Orchestrator 19
Table 5. Overview of the Cloud Evidence Collector's API functions 23
Table 6. Overview of the Security Assessment module's API functions 24
Table 7. Overview of the Orchestrator's API functions 24
Table 8. Overview of the Clouditor package structure 29
Table 9. Component card for Wazuh 34
Table 10. Component card for Wazuh and VAT Evidence Collector 35
Table 11. Overview of the Wazuh-deploy package structure 41
Table 12. Overview of the Wazuh and VAT Evidence Collector package structure 41
Table 13. The component card for the Vulnerability Assessment Tools 45
Table 14. Component card for Codyze 64
Table 15. Component card for the Assessment and Management of Organisational Evidence 72
Table 16. Overview of AMOE's source code package contents 77
Table 17. Overview of requirements satisfaction according to current implementation of the tools presented in this deliverable 85
Table 18. Requirements satisfied by each tool 86

List of figures

Figure 1. WP3 Architecture and directly related components (source: D3.3 [2]) 14
Figure 2. Overview of the Clouditor architecture 16
Figure 3. Sample policies written in Rego: They compare a given encryption algorithm to a given target value (see next figures), depending on a given operator 28
Figure 4. Sample data that is provided by the central Catalogue of Controls and Metrics values 28
Figure 5. A sample excerpt of an evidence 28
Figure 6. High-level Wazuh's architecture 39
Figure 7. High-level schema of Wazuh, VAT, and related components 40
Figure 8. Internal architecture schema of the Vulnerability Assessment Tools 49
Figure 9. An excerpt from the graph generated by the Cloud Property Graph 55
Figure 10. Codyze architecture 67
Figure 11. Main architecture for AMOE (keyword-based extraction method) 71
Figure 12. AMOE prototype architecture 75
Figure 13. AMOE file upload dialog 93
Figure 14. AMOE landing page after file upload 93
Figure 15. AMOE evidence extraction progress 94
Figure 16. AMOE assessment status overview per document 94
Figure 17. AMOE overview of extracted evidence and meta data linked to the uploaded file 95
Figure 18. AMOE view of organisational evidence 96
Figure 19. Show processed evidence (HTML view) 97

Terms and abbreviations

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AWS	Amazon Web Services
BSD	Berkeley Software Distribution
BSI	Bundesamt für Sicherheit in der Informationstechnik
CI/CD	Continuous Integration / Continuous Deployment
CLI	Command Line Interface
CloudPG	Cloud Property Graph
CPG	Code Property Graph
CPU	Central Processing Unit
CSA or EU CSA	EU Cybersecurity Act
CSP	Cloud Service Provider
CVE	Common Vulnerabilities and Exposures
DB	Data Base
DSL	Domain Specific Language
DLT	Distributed Ledger Technologies
EC	European Commission
ELK	ElasticSearch, Logstash, Kibana
EUCS	European Cybersecurity Certification Scheme for Cloud Services
GA	Grant Agreement to the project
GDPR	General Data Protection Regulation
GNU GPL	GNU General Public License
GPL	General Public License
gRPC	Google Remote Procedure Call
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HIPAA	Health Insurance Portability and Accountability Act
HTTP	HyperText Markup Language
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
JNI	Java Native Interface
JSON	JavaScript Object Notation
K8S	Kubernetes
KPI	Key Performance Indicator
LLVM	Low Level Virtual Machine
LLVM-IR	Low Level Virtual Machine Intermediate Representation
LSP	Language Server Protocol
Nmap	Network Mapper
OASIS	Organization for the Advancement of Structured Information
OPA	Open Policy Agent
OS	Operating System
OWASP	Open Web Application Security Project
PaaS	Platform as a Service
PCI DSS	Payment Card Industry Data Security Standard
PoC	Proof of Concept
RAM	Random Access Memory
REST	Representational State Transfer

RPC	Remote Procedure Calls
SARIF	Static Analysis Results Interchange Format
SAST	Static Application Security Testing
SSL	Secure Sockets Layer
TLS	Transport Layer Security
UI	User Interface
URL	Uniform Resource Locator
YAML	Yet Another Markup Language
XML	Extensible Markup Language
VAT	Vulnerability Assessment Tools

Executive Summary

This deliverable presents the third and final version of the design, architecture, and implementation state of the MEDINA evidence gathering components which have been developed in the scope of Task 3.2, Task 3.3, and Task 3.4. It describes the components that produce evidence based on the assessment of cloud infrastructure (*Clouditor, Wazuh, VAT*), assessment of cloud applications source code (*Cloud Property Graph* and its *LLVM extension* and *Codyze*), and assessment of organisational measures (*AMOE*), providing an overview of how these components relate and interact between each other and the rest of the MEDINA framework.

For each component, this document describes its purpose and scope, the coverage of MEDINA requirements, the component's internal architecture and its subcomponents, the external architecture and relation to other components, the implementation state at the point of producing this deliverable, and technical details of the component including the programming languages and frameworks used, information about the packaging and installation of the component, and licensing. It is also mentioned which EUCS [1] requirements the respective evidence gathering tool should cover.

The components presented in this document currently satisfy all their functional requirements and are fully integrated with other components of the MEDINA framework. An overview of requirement fulfilment is presented in *Appendix A: MEDINA Requirements Implementation Overview.*

1 Introduction

Any automated compliance monitoring must start with the gathering of evidence upon which the analysis and decisions about the certification state can be made. In the case of MEDINA, this is done with components that first assess cloud infrastructure, resources and processes and then produce evidence about their status for further compliance check.

This document presents the technical details about the implementation of evidence gathering tools. The evidence gathering methodology, integration of evidence gathering tools into MEDINA framework (Task 3.1) and maintaining the trustworthiness of evidence (Task 3.5) are presented in more detail in deliverable D3.3 [2] that has been released in parallel with D3.6. The deliverable D3.3 also contains description of the design of evidence gathering components, an overview of the related state of the art as well as a more detailed analysis of the coverage of EUCS requirements by each of all MEDINA evidence gathering tools. The external architecture of the components and the relationship with all other MEDINA tools is further described in the scope of the overall MEDINA architecture in D5.2 [3], which also lists all MEDINA functional and technical requirements, elicited in WP5.

This document is the successor of D3.5 [4], which presented the intermediate (in month 24) versions of the same components mentioned above. D3.6 follows the same structure and keeps some of the content of the previous version to keep the document self-contained and easier to follow.

1.1 About this deliverable

This is a report on the final version of the design, implementation, and integration of the *MEDINA Evidence Management Tools*. It is the third of three iterations (following D3.5 [4]) of the deliverable resulting from:

- Task 3.2, implementing the tools for assessing the security performance of cloud workloads and providing evidence about fulfilment of technical measures related to the operational cloud infrastructure.
- Task 3.3, implementing tools for assessing and collecting evidence about the security implications of cloud applications used and their data flows through analysis of the application source code.
- Task 3.4, implementing a component for the assessment of organisational measures based on the analysis of CSP’s documentation about their policies and processes.

1.2 Document Structure

This document is organised in the following sections:

1. *Introduction* (section 1) provides the context of the results reported in this document, their scope and structure, and mentions the relationship to other work in the MEDINA project, as well as the modifications of this document compared to its second version, D3.5 [4].
2. *Evidence Management Tools High-level Architecture* (section 2) gives an overview of the components described in this document, and presents the architecture and relations between them.
3. *Security Assessment of Cloud Infrastructure* (section 3) reports on the design and implementation of *Clouditor, Wazuh,* and *Vulnerability Assessment Tools*. The goal of these components is to provide evidence about conformity to technical measures regarding the cloud infrastructure and its configuration.

4. *Security Assessment of Cloud Applications* (section 4) reports on the design and implementation of *Cloud Property Graph* and *Codyze* components for cloud application source code analysis and provision of related technical evidence.
5. *Assessment of Organisational Measures* (section 5) gives a report on the design and implementation of the component for extracting evidence of organisational measures from policy documents (*AMOE*).
6. *Conclusion* (section 6) summarizes and briefly comments on the reported results.

Appendix A (section 8) gives an overview on how components cover the EUCS requirements.

Appendices B - F (sections 9-13) contain the readme, installation instructions and user manuals for the components described in the document.

1.3 Updates from D3.5

It should be noted that this document keeps some content that was included in D3.5 [4] and has not changed since then. Such material is kept in this deliverable to make it self-contained and easier to follow. For simpler tracking of progress and updates with regards to the previous deliverable version (D3.5), Table 1 shows a brief overview of the changes and additions to each of the document sections.

Table 1. Overview of deliverable updates with respect to D3.5

Section	Changes
2	Minor updates to the description of the *Cloud Property Graph* and LLMV extensions
3.1	• Minor updates to the general description • Updated requirements implementation status • Updated info about the database in the section "Technical specifications" • Added description of new features in the section "Advancements within MEDINA" • Updated info about *Security Assessment* and the *Orchestrator* in the section "Limitations and future work"
3.2	• Updated requirements implementation status • Updated info about *Wazuh* and *VAT Evidence Collector* packages in the section "Package information" • Updated description of covered metrics in the section "Limitations and future work"
3.3	• Updated description of requirements/metrics covered and the custom script functionality in the section "Functional description" • Updated requirements implementation status • Updated description of advances in the section "Advancements within MEDINA" • Updated description of the custom script implementation status in section "Limitations and future work"
4.1	• Updated functional description • Update section on relevant requirements and their implementation state
4.2	• Added new section "LLVM Extensions of the Code Property Graph"
4.3	• Moved the *Codyze* description from section 4.2 to 4.3 • Update section on relevant requirements and their implementation state • Updated section on limitation and future work reflecting progress

5	- Updated descriptions regarding installation and download
6	- Updated description to reflect the final status of evidence gathering components
Appendix A	Contents were updated according to the current (final) implementation state of the components. Added info about requirements covered by the LLVM Extensions of the *Code Property Graph*.
Appendix B	Footnote corrections in the section containing a README, user manual and installation instructions for *Clouditor*.
Appendix C	Updated installation instructions and footnote corrections in the section containing the user manual and installation instructions for *Codyze*.
Appendix D	Minor text corrections.
Appendix E	Figure numbers and minor text corrections in the section containing the user manual for *AMOE*.
Appendix F	New section containing readme and installation instructions for *Wazuh and VAT Evidence Collector*.

2 Evidence Management Tools High-level Architecture

This section gives a brief overview of the high-level architecture of MEDINA WP3 components, which result in the *MEDINA Evidence Management Tools*. These components gather evidence about CSP's fulfilment of technical and organisational measures, perform initial processing of the evidence, and transmit it to other MEDINA components. The overall architecture of the MEDINA framework is presented in more detail in D5.2 [3].

Figure 1 shows the architecture and data workflow among WP3 and other related components in the MEDINA framework. The tools for collecting evidence about technical measures are represented at the bottom part of the figure. They are connected to the infrastructure under evaluation either through an interface of the underlying cloud provider or installed directly in the CSP's (virtual) machines. These components are further described in section 3:

- *Clouditor* (section 3.1) collects evidence about the secure configuration of cloud resources.
- *Wazuh* (section 3.2) is installed in the CSP's cloud infrastructure and monitors the security state of the individual machines.
- *Vulnerability Assessment Tools* (section 3.3) are also installed in the CSP's infrastructure and can periodically scan the configured servers and networks for vulnerabilities, or run user-provided custom scripts for monitoring specialized metrics and producing evidence based on the output.

Technical evidence, obtained from the analysis of the source code of cloud applications, is gathered by *Codyze* (section 4.3). *The Cloud Property Graph* (section 4.1) and the *LLVM extensions* (section 4.2) can also gather evidence based on the analysed source code or binaries, but they are not included in the architectural diagram since they are novel research approaches that are not integrated with the other components (see explanation in section 4). Evidence about technical measures can also be collected by custom CSP-native components.

AMOE, the component for organisational evidence gathering and processing (section 5) analyses various documents and policies of the CSP and based on this produces evidence about the CSP's compliance to organisational requirements of the certification framework.

The tools for processing the evidence into assessment results and gathering them are represented in the middle of Figure 1. The evidence produced by all the above-mentioned components must be transformed into security assessment results with the information whether the addressed metric measured on the particular evaluation resource (e.g., virtual machine, cloud computing resource, storage, process, policy) is compliant or not. The assessment results can be either produced by the evidence collection components internally and sent directly to the *Orchestrator*, or by the specialized *Security Assessment* component (both described in section 3.1).

The *Security Assessment* component assesses the received evidence based on the target values coming from the certification specification and the CSP's configuration. For each evidence object, the *Security Assessment* outputs a security assessment result with the information whether the addressed metric measured on the particular evaluation resource (e.g., virtual machine, cloud computing resource, storage, process, policy) is compliant or not. The component is based on the respective component in the *Clouditor* framework.

The assessment results and associated evidence are all gathered by the *Orchestrator* (also based on *Clouditor*). It stores this data in the respective databases and makes it available to the other components, mostly part of WP4 and WP6 (e.g., *Continuous Certification Evaluation* [1], *Company Compliance Dashboard* [2]). The evidence and assessment results are also forwarded to the *MEDINA Evidence Trustworthiness Management system* [3], which uses Blockchain technologies to ensure the authenticity of data when retrieved at a later stage.

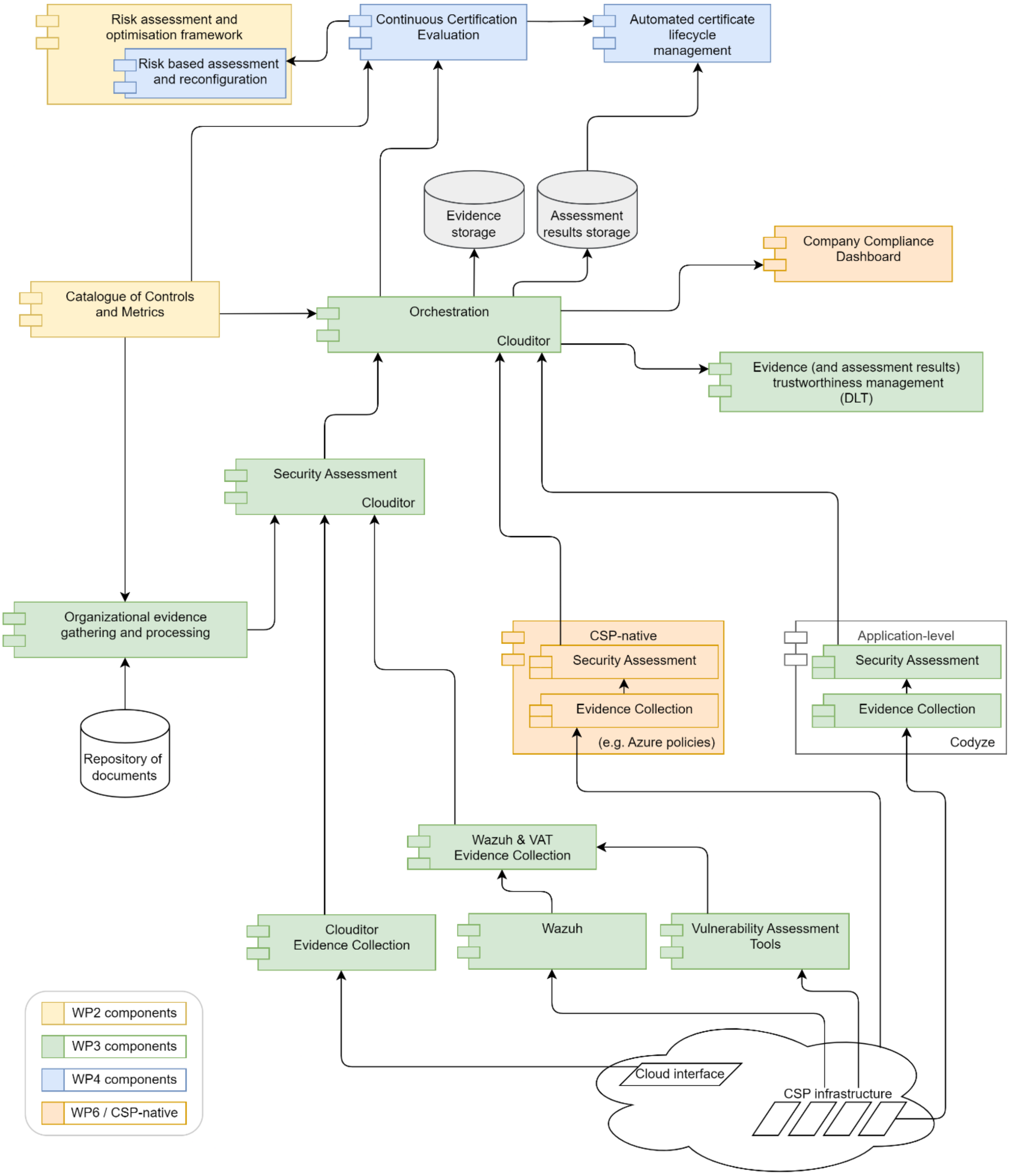

Risk assessment and optimisation framework
Risk based assessment and reconfiguration
Continuous Certification Evaluation
Automated certificate lifecycle management
Evidence storage
Assessment results storage
Company Compliance Dashboard
Catalogue of Controls and Metrics
Orchestration
Clouditor
Evidence (and assessment results) trustworthiness management (DLT)
Security Assessment
Clouditor
Organizational evidence gathering and processing
Repository of documents
CSP-native
Security Assessment
Evidence Collection
(e.g. Azure policies)
Application-level
Security Assessment
Evidence Collection
Codyze
Wazuh & VAT Evidence Collection
Clouditor Evidence Collection
Wazuh
Vulnerability Assessment Tools
Cloud interface
CSP infrastructure
WP2 components
WP3 components
WP4 components
WP6 / CSP-native

Figure 1. WP3 Architecture and directly related components (source: D3.3 [2])

[1] *CCE* is developed in the scope of WP4 and reported in D4.3 [54]

[2] *CCD* is developed in the scope of WP6 and reported in D6.3 [55]

[3] *MEDINA Evidence Trustworthiness Management* is developed in the scope of WP3 and reported in D3.3 [3]

3 Security Assessment of Cloud Infrastructure

This section describes the technical structure and implementation state of the components responsible for collecting evidence about the security performance of cloud workloads (e.g., cloud configuration, virtual machines and containers, or software running in them). The following subsections present the individual components, developed in the scope of Tasks 3.1 and 3.2.

3.1 Clouditor-Based Components

Clouditor is a monitoring tool for cloud systems that can automatically and continuously discover existing resources in a cloud system, query their configuration, assess the gathered information according to pre-defined metrics, and more. These metrics may be mapped to certification frameworks, such as the EUCS [1], to demonstrate compliance with the certification requirements.

Several components in the MEDINA framework are based on *Clouditor*, i.e., the *Cloud Evidence Collector*, the *Security Assessment*, and the *Orchestrator*.

3.1.1 Implementation

The following subsections provide functional and technical descriptions of the Clouditor-based components.

3.1.1.1 Functional description

There are three components that make up *Clouditor*: the *Cloud Evidence Collector*, which discovers cloud resources and creates MEDINA evidence for them; the *Security Assessment*, which uses metrics to assess evidence and creates assessment results; and the *Orchestrator*, which manages the evidence and assessment results flow, storage, and other utility functionalities.

The current implementation of *Clouditor* supports several cloud systems, i.e., Microsoft Azure, Amazon Web Services, and Kubernetes. The resource configurations in these platforms are checked by means of various metrics. Examples of resource configuration checks are the following:

- Secure transport encryption with TLS
- Secure TLS version
- Data at rest encryption in various storage resources
- Resource deployment in allowed regions

3.1.1.1.1 Fitting into overall MEDINA Architecture

The three microservices that make up *Clouditor* constitute central components in the MEDINA framework as described in the following. Figure 2 shows an overview of the current *Clouditor* architecture.

Cloud Evidence Collector

The *Cloud Evidence Collector* gathers evidence from cloud workloads. As such, it is one of the evidence collectors that can be integrated into the MEDINA framework.

Security Assessment

The *Security Assessment* first retrieves metrics and target values from the *Orchestrator* and then assesses any incoming evidence accordingly. The *Security Assessment* can be integrated with various evidence collectors. In this way, CSPs can develop their own evidence collectors, and simply let them send evidence to the (*Clouditor*) *Security Assessment*.

Orchestrator

The *Orchestrator* is the central management component of MEDINA which manages database access, cloud services, a user interface, and more. If CSPs decide to implement a custom *Security Assessment*, they can integrate it with the *Orchestrator* according to the MEDINA data model.

The *Clouditor Security Assessment* currently processes evidence of the *Cloud Evidence Collector*, as well as evidence of other evidence collection tools, e.g., *Wazuh*. The *Orchestrator* processes the results of the *Clouditor Security Assessment* as well as results of other security assessment tools, e.g., *Codyze*.

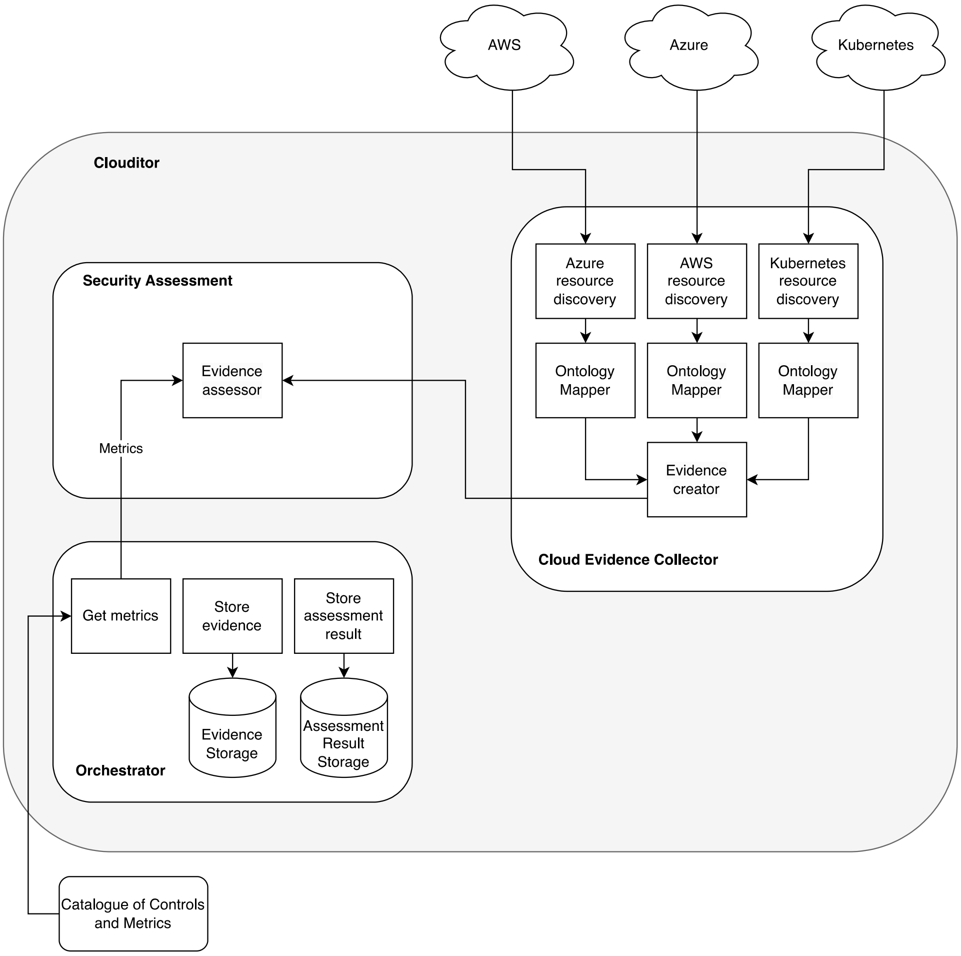
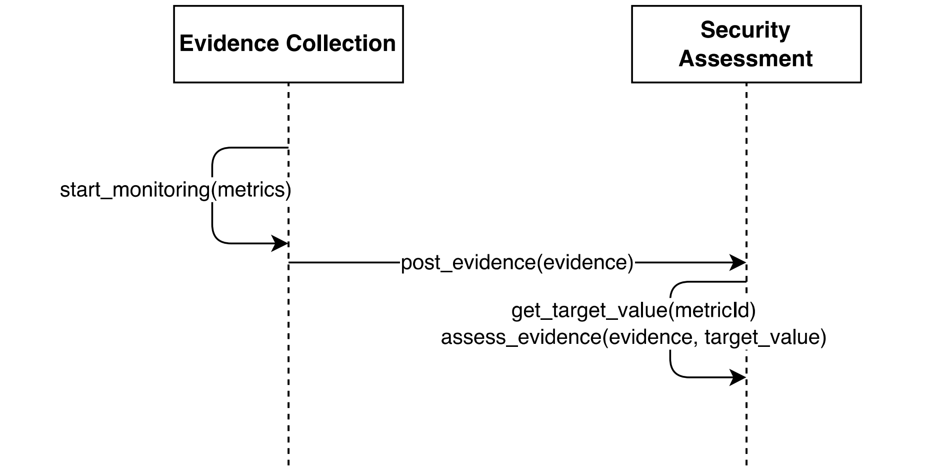


Figure 2. Overview of the Clouditor architecture

3.1.1.1.2 Component cards

Table 2. Component card for the Cloud Evidence Collector

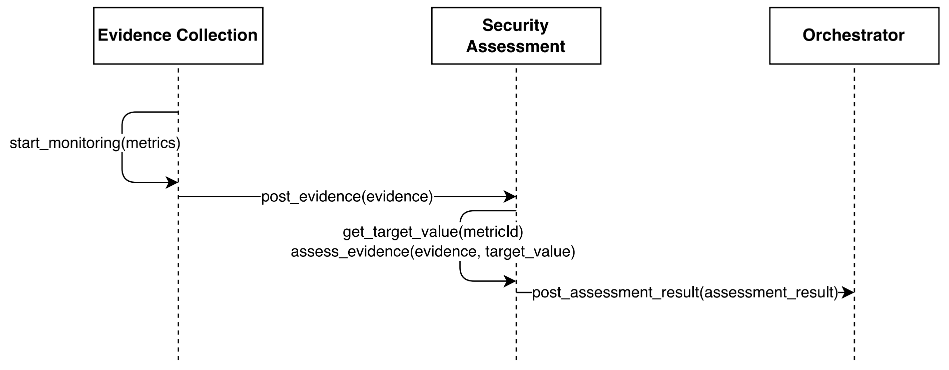
<table>
<tr><td><b>Component Name</b></td><td><b><i>Cloud Evidence Collector</i></b></td></tr>
<tr><td><b>Main functionalities</b></td><td>The component provides the following functionalities:
• Evidence gathering from Cloud APIs</td></tr>
<tr><td><b>Sub-components Description</b></td><td>The evidence gathering discovers resources in cloud systems, like Azure and AWS, via their standard APIs, and forwards this information to the Security Assessment. It is thus composed of the subcomponents <i>Azure-Discovery, AWS-Discovery,</i> and <i>Kubernetes-Discovery.</i></td></tr>
<tr><td><b>Main logical Interfaces</b></td><td>The <i>Cloud Evidence Collector</i> does not offer APIs to be called by other components. It only retrieves data about cloud resources and translates them to the MEDINA evidence data model.
Note that a graphical presentation of the resources that are discovered by this component can be obtained via the <i>Orchestrator</i> UI.</td></tr>
<tr><td><b>Requirements Mapping</b></td><td>List of requirements covered by this component (see in D5.2 [3]):
TEGT.C.01-02, TEGT.S.01</td></tr>
<tr><td><b>Interaction with other components</b></td><td><table><tr><th>Interfacing Component</th><th>Interface Description</th></tr><tr><td><i>Security Assessment</i></td><td>Send evidence</td></tr></table></td></tr>
<tr><td><b>Relevant sequence diagram/s</b></td><td>
Evidence Collection
Security Assessment
start_monitoring(metrics)
post_evidence(evidence)
get_target_value(metricId)
assess_evidence(evidence, target_value)

</td></tr>
<tr><td><b>Current TRL[4]</b></td><td>TRL4</td></tr>
<tr><td><b>Target TRL[5]</b></td><td>TRL5</td></tr>
<tr><td><b>Programming language</b></td><td>Go</td></tr>
<tr><td><b>License</b></td><td>Apache 2.0</td></tr>
<tr><td><b>WP and task</b></td><td>WP3, Task 3.1</td></tr>
<tr><td><b>MEDINA Workflows</b></td><td>WF2 “Preparation of MEDINA Components”, and
WF5 “EUCS Compliance Assessment” (see D5.4 [5])</td></tr>
</table>

[4] TRL value before validation

[5] TRL value after validation

Table 3. Component card for the Security Assessment

<table>
<tr><td><b>Component Name</b></td><td><b><i>Security Assessment</i></b></td></tr>
<tr><td><b>Main functionalities</b></td><td>The component provides the following functionalities:
• Assessment of evidence according to pre-defined metrics</td></tr>
<tr><td><b>Sub-components Description</b></td><td>The <i>Security Assessment</i> compares the received evidence against pre-defined metrics and their target values and forwards the resulting <i>assessment results</i> to the <i>Orchestrator</i>.</td></tr>
<tr><td><b>Main logical Interfaces</b></td><td>Note that a graphical presentation of assessment results can be obtained via the <i>Orchestrator</i> UI.
<table>
<tr><th>Interface name</th><th>Description</th><th>Interface technology</th></tr>
<tr><td>Assessment interface</td><td>An interface for providing evidence to be assessed against suitable metrics</td><td>gRPC</td></tr>
</table></td></tr>
<tr><td><b>Requirements Mapping</b></td><td>List of requirements covered by this component (see D5.2 [3]):
EAT.01-03</td></tr>
<tr><td><b>Interaction with other components</b></td><td>
<table>
<tr><th>Interfacing Component</th><th>Interface Description</th></tr>
<tr><td><i>Orchestrator</i></td><td>Send assessment results</td></tr>
</table></td></tr>
<tr><td><b>Relevant sequence diagram/s</b></td><td>
Evidence Collection
Security Assessment
Orchestrator
start_monitoring(metrics)
post_evidence(evidence)
get_target_value(metricId)
assess_evidence(evidence, target_value)
post_assessment_result(assessment_result)

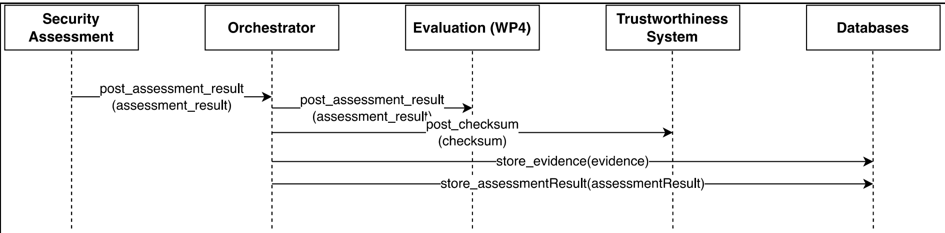
</td></tr>
<tr><td><b>Current TRL</b>[6]</td><td>TRL4</td></tr>
<tr><td><b>Target TRL</b>[7]</td><td>TRL5</td></tr>
<tr><td><b>Programming language</b></td><td>Go</td></tr>
<tr><td><b>License</b></td><td>Apache 2.0</td></tr>
<tr><td><b>WP and task</b></td><td>WP3, Task T3.2</td></tr>
<tr><td><b>MEDINA Workflows</b></td><td>WF2 “Preparation of MEDINA Components”, and
WF5 “EUCS Compliance Assessment” (see D5.4 [5])</td></tr>
</table>

[6] TRL value before validation

[7] TRL value after validation

Table 4. Component card for the Orchestrator

<table>
<tr><th>Component Name</th><th><i>Orchestrator</i></th></tr>
<tr><td>Main functionalities</td><td>The component provides the following functionalities:<ul><li>Provide configuration interfaces</li><li>Provide interfaces to the databases</li><li>Forward assessment results to appropriate components</li></ul></td></tr>
<tr><td>Sub-components Description</td><td>The Orchestrator mainly provides APIs to various components (see below).</td></tr>
<tr><td>Main logical Interfaces</td><td><table>
<tr><th>Interface name</th><th>Description</th><th>Interface technology</th></tr>
<tr><td>Assessment results storage</td><td>An interface to provide assessment results which are then stored in the relevant database, and forwarded to the relevant components</td><td>REST / gRPC</td></tr>
<tr><td>Database access</td><td>An interface that provides access to stored evidence and assessment results, to the configuration of cloud services and targets of evaluation, etc.</td><td>REST / gRPC</td></tr>
<tr><td>DLT storage</td><td>An interface to the DLT through which evidence and assessment result checksums are stored to the trustworthiness system.</td><td>REST</td></tr>
<tr><td>Configure metrics and target values</td><td>An interface that provides access to metrics and target values</td><td>REST / gRPC</td></tr>
<tr><td>Graphical UI</td><td>A graphical UI that allows to view stored data, configure cloud services and targets of evaluation, etc.</td><td>JavaScript</td></tr>
</table></td></tr>
<tr><td>Requirements Mapping</td><td>List of requirements covered by this component (see D5.2 [3]):
ECO.01-04</td></tr>
<tr><td>Interaction with other components</td><td><table>
<tr><th>Interfacing Component</th><th>Interface Description</th></tr>
<tr><td>Assessment tools</td><td>Receives assessment results from assessment tools</td></tr>
<tr><td>Databases</td><td>Stores and retrieves evidence/assessment results from the relevant databases</td></tr>
<tr><td>Trustworthiness system</td><td>Sends assessment result hashes to the trustworthiness system</td></tr>
<tr><td>Metrics and target values repository</td><td>Retrieves metrics and target values for the assessment components and offers an API to modify them</td></tr>
</table></td></tr>
</table>

Relevant sequence diagram/s	 Security Assessment Orchestrator Evaluation (WP4) Trustworthiness System Databases post_assessment_result (assessment_result) post_assessment_result (assessment_result) post_checksum (checksum) store_evidence(evidence) store_assessmentResult(assessmentResult)
Current TRL[8]	TRL4
Target TRL[9]	TRL5
Programming language	Go
License	Apache 2.0
WP and task	WP3, Task3.1
MEDINA Workflows	WF2 “Preparation of MEDINA Components”, WF3 “EUCS deployment on ToC”, WF5 “EUCS Compliance Assessment”, and WF6 “EUCS – Maintenance of ToC certificate” (see D5.4 [5])

3.1.1.1.3 Related requirements, common for all *Clouditor*’s components

The relevant requirements from Deliverable D5.2 [3] are listed below with a brief description of how they are implemented. The requirements are grouped by each *Clouditor* component.

Common requirements

Requirement id	TEGT.C.01
Short title	Continuous collection
Description	The developed tools must be able to collect evidence continuously, i.e. in (high)-frequency intervals.
Implementation state	Fully implemented

Currently, the discovery interval in the *Cloud Evidence Collector* component is set to 5 minutes and can only be changed in the source code.

Requirement id	TEGT.C.02
Short title	Provision to defined interfaces
Description	The developed tools must provide collected evidence to a security assessment tool via its offered APIs.
Implementation state	Fully implemented

The *Cloud Evidence Collector* sends evidence to the *Security Assessment* component via its offered APIs.

[8] TRL value before validation

[9] TRL value after validation

Related requirements for the *Cloud Evidence Collector* component

Requirement id	TEGT.S.01
Short title	Collect evidence from cloud interfaces
Description	The developed tool must be able to collect evidence of cloud workloads, e.g., virtual machines, containers, and serverless functions.
Implementation state	Fully implemented

The *Cloud Evidence Collector* gathers evidence of cloud workloads from different CSPs (Azure, AWS, etc.). Resources are currently discovered in compute, storage, and network services in Azure, compute and storage services in AWS, and compute and network services in Kubernetes.

Requirement id	TEGT.S.09
Short title	Collect evidence from CSP-native services
Description	The developed tool should be able to query findings from CSP-native services, like Azure Policy, to integrate them in MEDINA by querying the respective cloud API.
Implementation state	Fully implemented

Currently, a prototypical implementation of the CSP-native evidence collection is implemented. CSPs benefit from this component by integrating security assessment results from existing security posture management systems, such as Microsoft Azure Security Center.

Related requirements for the *Security Assessment* component

Requirement id	EAT.01
Short title	Evidence assessment target
Description	The target values for the evidence assessment must be retrieved from a central repository of target values (WP2).
Implementation state	Fully implemented

The *Security Assessment* component retrieves target values from the *Orchestrator*, which in turn retrieves them from a central repository.

Requirement id	EAT.02
Short title	Continuous evidence assessment
Description	All evidence collection tools must forward evidence and measurement results (according to the data format defined in MEDINA) to the respective assessment components.
Implementation state	Fully implemented

The *Cloud Evidence Collector* sends evidence to the *Security Assessment* by using the provided APIs. They are then used to generate assessment results which indicate if the evidence is compliant or not.

Requirement id	EAT.03
Short title	Evidence assessment results
Description	The assessment results of evidence assessments must be submitted to the evidence Orchestrator via the API it provides.

Implementation state	Fully implemented

The *Security Assessment* component submits the assessment results by using the provided *Orchestrator* APIs.

Requirement id	**EAT.04**
Short title	Assess CSP-Native evidence
Description	The developed tool should be able to assess the CSP-native evidence or translate CSP-native assessment results to the MEDINA data model.
Implementation state	Fully implemented

Currently, a prototypical implementation of the CSP-native evidence collection and assessment is implemented. See also TEGT.S.09.

Related requirements for the *Orchestrator* component

Requirement id	**ECO.01**
Short title	Provision of Interfaces
Description	The evidence Orchestrator must provide standard interfaces for the evidence collection and assessment tools (T3.2-T3.4) to securely store their results.
Implementation state	Fully implemented

Interfaces are provided by RPC (Remote Procedure Call) APIs with gRPC[10], as well as via REST. Currently, the assessment tools send assessment results accompanied by the evidence they are based on. The transmission of evidence to the database can be encrypted.

Requirement id	**ECO.02**
Short title	Conformity to selected assurance level
Description	The evidence Orchestrator must ensure that the evidence collection (T3.2-T3.4) is performed according to the selected assurance level, i.e., it must trigger the evidence collection of the respective tools.
Implementation state	Not implemented (out of scope)

Currently, the *Cloud Evidence Collector* is triggered via a CLI (Command Line Interface) command. Then the collected evidence is sent to the *Security Assessment* and the generated assessment results are then sent to the *Orchestrator* which stores them in a database. Since MEDINA focuses only on the *high* assurance level, this requirement is out of scope and is not implemented.

Requirement id	**ECO.03**
Short title	Secure Transmission to evidence storage
Description	The evidence Orchestrator must securely transmit evidence to the evidence storage.
Implementation state	Fully implemented

[10] https://grpc.io/

The *Orchestrator* can store evidence either in memory or in a persistent database. In the latter case, the evidence can also be transmitted in an encrypted form, simply by specifying an SSL URL.

Requirement id	ECO.04
Short title	Transmission of evidence checksums
Description	The evidence Orchestrator should integrate a Ledger client that stores checksums of evidence in a DLT.
Implementation state	Fully implemented

The *Orchestrator* transforms assessment results into the desired format and forwards them to the ledger client.

3.1.1.2 Technical description

In the following sections, we provide the technical description of *Clouditor*'s components. First, the architectural design is presented consisting of the architectural view and the connection between the respective components. Then, information about the single components is presented and, finally, an overview of the technical description for the implementation of the prototype is given.

3.1.1.2.1 Prototype architecture

Clouditor employs a microservice architecture allowing individual components to scale and to be replaced, or allowing to add new components, e.g., adding evidence collection tools for new cloud services/providers. The *Cloud Evidence Collector*, *Security Assessment* and *Orchestrator* are such modular components that represent microservices. Like all parts in *Clouditor*, they are written in Go and communicate among each other via the gRPC protocol.

Since the architecture is defined by the three components and the communication between them, interface snippets of the individual components are provided below. For the detailed specification see the *./proto* folder within the *Clouditor* repository[11]. The specification is defined in the *Protocol Buffer Version 3 Language Specification*. In addition, see the *./openapi* folder containing each component`s auto generated *.yaml* files which follow the OpenAPI description for REST APIs.

Cloud Evidence Collector interface

Table 5. Overview of the Cloud Evidence Collector's API functions

Function Name	Parameters	Return Type	Description
Start	-	successful (bool)	Triggers the start of the discovering process. Returns true if the component started without errors
Query	filtered_type (string)	results (list of evidence)	Returns the latest set of evidence discovered

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

Security Assessment interface

Table 6. Overview of the Security Assessment module's API functions

Function Name	Parameters	Return Type	Description
TriggerAssessment	options (string)	-	Triggers the security assessment
ListAssessmentResults	-	results (list of assessment results)	Lists the latest set of assessment results
AssessEvidence/ AssessEvidences	evidence (Evidence) / evidences (stream of Evidence)	successful (bool)/ -	Assesses the evidence/ stream of evidence provided by the evidence collection tool

Orchestrator interface

Table 7. Overview of the Orchestrator's API functions

Function Name	Parameters	Return Type	Description
RegisterAssessmentTool	tools (AssessmentTool)	tool (AssessmentTool)	Registers the assessment tool
GetAssessmentTool	tool_id (string)	tool (AssessmentTool)	Returns the assessment tool with the given tool id
UpdateAssessmentTool	tool_id (string), tool (AssessmentTool)	tool (AssessmentTool)	Updates the assessment tool given by the tool id
DeregisterAssessmentTool	tool_id (string)	-	Deregisters the assessment tool with the given tool id
StoreAssessment/ StoreAssessment	result (AssessmentResult) / results (stream of AssessmentResult)	-	Stores the assessment result/ stream of assessment results provided by the assessment tool
StoreEvidenceResult/ StoreEvidenceResults	result (EvidenceResult)/ results (stream of EvidenceResult)	-	Stores the evidence provided by an assessment tool
GetMetric	metric_id (string)	Metric	Returns the metric with the given metric id
ListMetrics	-	List of Metrics	Returns a list of all metrics provided by the *Catalogue of Controls and Metrics*
GetCertificate	Certificate ID	Certificate	Gets a certificate, e.g., an EUCS certificate

Function Name	Parameters	Return Type	Description
ListCertificate	-	List of certificates	Returns a list of certificates
CreateCertificate	Certificate	Certificate	Validates, stores, and returns the created certificate
UpdateCertificate	Certificate, Certifiate ID	Certificate	Validates, updates, and returns the updated certificate
RemoveCertificate	Certificate ID	-	Deletes the certificate
GetCloudService	Cloud Service ID	Cloud Service	Returns the Cloud Service
ListCloudServices	-	List of Cloud Services	Returns a list of Cloud Services
RegisterCloudService	Cloud Service	Cloud Service	Validates, stores, and returns the created certificate
UpdateCloudService	Cloud Service, Cloud Service ID	Cloud Service	Validates, updates, and returns the updated certificate
RemoveCloudService	Cloud Service ID	-	Deletes the Cloud Service
GetMetricImplementation	Metric Implementation	Metric Implementation	Gets a Metric Implementation
ListMetricImplementations	-	List of certificates	Returns a list of Metric Implementations
CreateMetricImplementation	Metric Implementation	Metric Implementation	Validates, stores, and returns the created Metric Implementation
UpdateMetricImplementation	Metric Implementation, Metric Implementation ID	Metric Implementation	Validates, updates, and returns the updated Metric Implementation
RemoveMetricImplementation	Metric Implementation ID	-	Deletes the Metric Implementation
GetCatalog	Catalog ID	Catalog	Gets a Catalog
ListCatalogs	-	List of Catalogs	Returns a list of Catalogs
CreateCatalog	Catalog	Catalog	Validates, stores, and returns the created Catalog
UpdateCatalog	Catalog, CatalogID	Catalog	Validates, updates, and returns the updated Catalog
RemoveCatalog	CatalogID	-	Deletes the Catalog
GetCategory	Category ID	Category	Gets a Category
ListCategory	-	List of Categories	Returns a list of Categories
GetControl	Control ID	Control	Gets the Control

Function Name	Parameters	Return Type	Description
ListControl	-	Controls	Returns a list of Controls
GetTargetOfEvaluation	Target Of Evaluation ID	Target Of Evaluation	Gets a Target Of Evaluation
ListTargetsOfEvaluation	-	List of Targets Of Evaluation	Returns a list of Targets Of Evaluation
CreateTargetOfEvaluation	Target Of Evaluation	Target Of Evaluation	Validates, stores, and returns the created Target Of Evaluation
UpdateTargetOfEvaluation	Target Of Evaluation, Target Of Evaluation ID	Target Of Evaluation	Validates, updates, and returns the updated Target Of Evaluation
RemoveTargetOfEvaluation	Target Of Evaluation ID	-	Deletes the Target Of Evaluation

3.1.1.2.2 Description of components

This section presents the tools provided by *Clouditor* (see Figure 2), describing how they have been developed to meet the MEDINA requirements.

Cloud Evidence Collector

The functionality of the *Cloud Evidence Collector* can be divided into 3 parts:

- Fetching relevant properties of cloud resources (discovery),
- Creation of evidence objects, including their ontological concepts (see also D2.5 [6]), and
- Forwarding this evidence to the *Security Assessment* component.

Within the *Cloud Evidence Collector*, the discovery package is located at the top-level. Its purpose is to communicate with other services/components (in this case the *Security Assessment* component). In a first step, this service establishes a connection to the *Security Assessment* component, then it starts the various discoverers (e.g., for AWS S3), and forwards the collected evidence in a continuous manner. The transmission is done via a gRPC channel.

For each cloud vendor there is a separate sub-package, e.g., for AWS and Azure. In such a package there is one file (e.g. *aws.go*) containing the cloud vendor-specific discoverer which loads and initializes configurations and credentials that all underlying services share. For each discovered cloud service, there is a corresponding Go file that fetches the desired properties of that service via API calls (programmatic access). According to the ontology defined in WP2, these properties are then converted into a format that is independent from the used cloud vendor. The properties that can be fetched are dependent on the range of API calls the respective cloud vendor provides.

For Microsoft Azure, the currently discoverable services are *compute, blob storage* and *network.* In the case of Amazon Web Services, *compute* as well as *blob storage.* Through the Kubernetes API *compute* and *network* resources are currently discoverable.

Security Assessment

Clouditor's *Security Assessment* is responsible for evaluating incoming evidence and sending the generated assessment results to the *Orchestrator*.

Evidence is received from components – the evidence collectors – such as the *Cloud Evidence Collector* or the *Wazuh and VAT Evidence Collector*. As mentioned above, *Clouditor* follows a microservice architecture which allows any evidence collector to connect to it in a modular way. Such evidence collection tools only need to implement the MEDINA API in gRPC to send evidence as Protocol Buffer messages. Additionally, REST over HTTP is available for evidence collecting tools. The gRPC approach, however, allows to send evidence in a stream which can significantly increase the throughput.

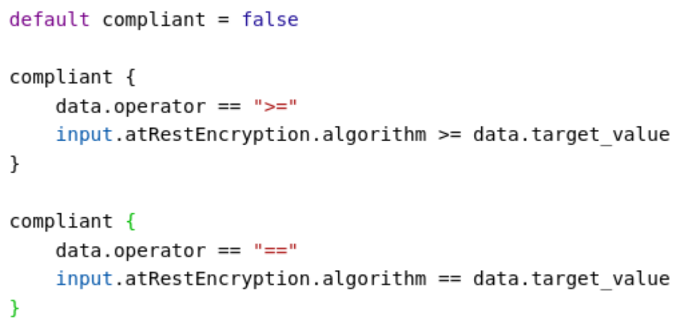
In a previous version of *Clouditor*, a dedicated policy rule language was used to assess evidence. Since no other tools outside the *Clouditor* tool suite needed to be connected to it, this approach was sufficient. In MEDINA, however, various evidence collection tools may connect to *Clouditor* – either to the *Security Assessment* or to the *Orchestrator*. To simplify the definition of policies, a more commonly used policy language, Rego from Open Policy Agent (OPA)[12], was introduced instead. OPA uses Rego as a uniform declarative policy language. A policy written in Rego asserts that an input (e.g., an evidence) conforms to user-specified constraints (target values and operators).

In the definition of Rego policies, the cloud resource ontology (see D2.45 [6]) is used. Since evidence provided by the evidence collection tools provide their ontological assignments, the Rego policies only need to specify rules based on properties following the format of the ontology. Consider the following example: A policy checks if an encryption algorithm's key length is larger than the given target value, e.g., 256 bits (see Figure 3). The user-specific constraint may then state that the algorithm's key length must be at least 256 bit long (see Figure 4). The input, i.e., the evidence, is illustrated in Figure 5. The algorithm version in the input is 256, therefore the policy engine will output the compliance state of true. Both, input and the policy written in Rego, are aligned with the cloud resource ontology. These policies can be written more easily by non-experts without having to know how evidence collection, assessment, orchestration, etc. work. The person defining the policy only needs to know the ontology to write policies based on it.

In the MEDINA framework, the Rego policies are generated from metrics which are stored in the *Catalogue of controls and metrics* component (see D2.2 [7]). When the Orchestrator triggers the assessment to start, it also sends the respective metrics along. The assessment then stores these metrics in cache for fast processing of the evidence.

The outcome of these assessments, the assessment results, are then sent to the *Orchestrator* and will eventually reach the Continuous Certification Evaluation component (see D4.3 [8]).

[12] https://www.openpolicyagent.org/docs/latest/policy-language/

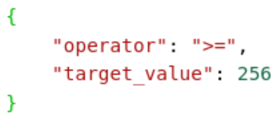


```
default compliant = false

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

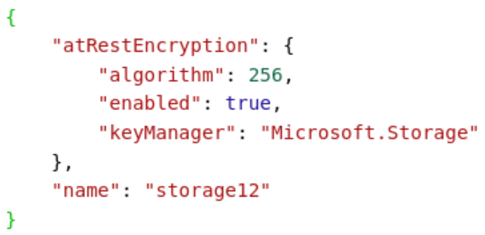
compliant {
    data.operator == "=="
    input.atRestEncryption.algorithm == data.target_value
}
```

Figure 3. Sample policies written in Rego: They compare a given encryption algorithm to a given target value (see next figures), depending on a given operator



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

Figure 4. Sample data that is provided by the central Catalogue of Controls and Metrics values



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

Figure 5. A sample excerpt of an evidence

The approach of semantically enriching evidence to enable a generic assessment as shown above has also been published at the SAC 2023 conference[13].

Orchestrator

The *Orchestrator* is a central component in the MEDINA framework and manages dataflows between components. As such, it also manages the interaction between components of different work packages. The *Orchestrator* also offers APIs to store and retrieve data and manage assessment tools. The APIs are defined in gRPC allowing other components to only implement the given API to send the data as Protocol Buffer messages. For some APIs it is also possible to send the data in a stream which can increase the throughput. Its interactions with other components and its functionalities are summarized in the following:

- The *Orchestrator* exposes two APIs for the security assessment tools, e.g., *Clouditor* Security Assessment, CSP-native or *Codyze* security assessment tool. One API is for the assessment results and another one to store evidence directly.
- The *Orchestrator* also acts as the central interface to the *Catalogue of controls and metrics* which is developed within WP2 (see deliverable D2.2 [7]). As such, it is responsible for providing relevant metrics to the assessment component.

[13] At the time of writing, the proceedings have not yet been published.

- Furthermore, the *Orchestrator* stores checksums of evidence and assessment results in the DLT via a Blockchain client. The implementation of the *MEDINA Evidence Trustworthiness Management System* component is also described in D3.3 [2].
- Additionally, **t**he *Orchestrator* forwards assessment results to the *Continuous Certification Evaluation* component which is developed within WP4 (see deliverable D4.3 [8]).
- The *Orchestrator* stores the evidence as well as the assessment results into the associated storages. It offers an in-memory storage as well as a PostgreSQL connection.

3.1.1.2.3 Technical specifications

The prototype is written in Go (version 1.19). A selection of key libraries is shown in the following and a full list of used libraries can be found in the Github repository[11].

- *github.com/Azure/azure-sdk-for-go*
- *github.com/aws/aws-sdk-go-v2*
- *k8s.io/client-go*
- *google.golang.org/grpc*
- *google.golang.org/protobuf*
- *gorm.io/driver/postgres*
- *gorm.io/driver/sqlite*

Either an in-memory or a PostgreSQL database can be used.

3.1.2 Delivery and usage

The following sections give a short overview of the delivery and usage of the prototype. Further technical details can be found in the *Clouditor* Github Repository[11].

Please note that the README, installation instructions and user manual of *Clouditor* can be found in *Appendix B: Clouditor - Readme, Installation instructions and User* manual.

3.1.2.1 Package information

Table 8 shows the structure of the important folders and a brief description of them.

Table 8. Overview of the Clouditor package structure

Folder	Description
api/	Code needed for the communication between the microservices. It mainly consists of auto-generated *Protobuf* and gRPC files.
cli/	This folder contains the *Clouditor* CLI based source code files.
cmd/	This folder contains the main files.
openapi/	This folder contains the auto-generated *OpenAPI* files.
persistence/	This folder contains the DB specific files.
policies/	This folder contains the *Rego* policy files per metric.
proto/	This folder contains the *Protobuf* files.
rest/	This folder contains the REST gateway implementation.
service/	This folder contains the source code for the microservices separated in individual folders for each service.
voc/	This folder contains the vocabulary files based on the ontology defined in WP2.

3.1.2.2 Licensing information

The *Clouditor* components are licensed under the open-source Apache License 2.0.

3.1.2.3 Download

The *Clouditor* source code can be found in the *Clouditor* Github repository[14]. The adapted MEDINA components can be found in the public MEDINA repository[15].

3.1.3 Advancements within MEDINA

Several modifications and features have been implemented in *Clouditor* throughout the MEDINA project:

- The component has been completely reimplemented in the Go programming language.
- The previous *Clouditor* architecture has been redesigned to create several microservices, e.g., separate microservices for evidence gathering, assessment, and orchestration. This modularization allows for better scalability, as well as allows to integrate alternative services, for instance other evidence gathering tools.
- The evidence gathering service has been extended with an ontology mapping, i.e., the resource properties that are discovered are enhanced with a mapping to a cloud resource ontology. For example, a virtual machine’s properties are extended with a mapping to the ontology concepts *computing* and *virtual machine*. This approach allows to define metrics independently from the cloud provider and certification catalogue. For information, please refer to the respective description in deliverable D2.5 [6] (cloud resource ontology).
- As described above, the assessment service has been reimplemented as a separate microservice as well to conform to the MEDINA guidelines and data model. Also, its usage of the OPA[16] policy engine has been added, which is used to evaluate incoming evidence against metrics and their target values. These are defined using the OPA policy language Rego.
- The *Orchestrator* service is a completely new component in *Clouditor*, i.e., its APIs, data model, and integration with other components has been designed and implemented from scratch within MEDINA.
- The three components have been integrated and tested with each other, as well as with other MEDINA components, such as the *Continuous Certification Evaluation*, the *MEDINA Evidence Trustworthiness Management System*, the *Catalogue of Controls and Metrics*, and the *DSL Mapper*.
- Additional APIs have been implemented, for instance for the update of metric target values.
- Integration of the various components with the central OAuth server.
- Improvement of the data model and persistence.
- Introduction of data entities and APIs for adding cloud services and certification frameworks.
- Introduction of the Target of Evaluation which binds a cloud service to a certification framework and supports an n:m relation between cloud services and certification frameworks to be evaluated.

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

[15] https://git.code.tecnalia.com/medina/public

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

- Also, multiple integrations with other components have been developed to automatically propagate newly created, or somehow updated, cloud services and targets of evaluation to all components which need this data (e.g. the *Risk Assessment and Optimisation Framework*).
- A user interface has been developed and adapted to the needs of the MEDINA use cases.
- Based on the feedback received from the first validation phase, numerous improvements have been made with regard to the usability of the UI. Furthermore, authorization functionalities have been implemented to limit the retrieval of information about cloud services, Targets of Evaluation, certificates, etc., to the authorized roles as specified in D5.4 [5].

3.1.4 Limitations and future work

Cloud Evidence Collector

The Cloud Evidence Collector, which currently collects evidence from Microsoft Azure, AWS, and Kubernetes systems, is limited by the access rights it is given in the respective user management system, such as Azure Active Directory. Therefore, it will only measure the resources that are visible to its given user. Furthermore, cloud provider APIs may change, so the component needs to be updated accordingly. If, for instance, relevant security properties like access control properties change, their inclusion in the MEDINA evidence needs to be aligned in this component. Also, the evidence collection is limited by the information that the cloud provider APIs implement: if a certain encryption property, for example, would not be implemented by an API, the evidence collection for that property would not be possible. Since *Cloud Evidence Collector* adds ontological terms to the evidence, also limitations of the ontology need to be considered. First, the ontology terms need to be added correctly to the evidence or the *Security Assessment* will apply the wrong metrics to it. Second, the ontology needs to be maintained and its changes need to be implemented accordingly in the evidence collection.

Security Assessment

The *Security Assessment* component uses the Open Policy Agent (OPA) and Rego to perform the assessment of evidence against expected values (defined in the MEDINA metrics). OPA is in version 0.50 as of March 2023; future breaking changes therefore may occur which have to be incorporated in this component. It is furthermore dependent on the availability of the *Orchestrator* since it must forward assessment results to the *Orchestrator* and receive metric implementations (Rego code) from it.

Orchestrator

The *Orchestrator* is the central management component in MEDINA. While it presents an efficient component for forwarding data, managing database accesses, etc., it is also a single point of failure for the framework since without it, no evidence or assessment results can be processed or stored.

3.2 Wazuh

Wazuh [9] is an open-source security monitoring tool for threat detection, integrity monitoring, incident response and basic compliance monitoring. It can be deployed on-premises or in hybrid and cloud environments.

3.2.1 Implementation

The following subsections provide functional and technical descriptions of *Wazuh*.

3.2.1.1 Functional description

Wazuh's role in MEDINA is to provide capabilities for threat detection to MEDINA users (CSPs) while producing evidence related to its usage and potentially detected threats. *Wazuh*'s connection to MEDINA is enabled by the *Wazuh and VAT Evidence Collector* component. It connects to *Wazuh* to query its configuration and detected events and produces evidence based on this data.

Unlike some other evidence gathering tools (e.g., *Clouditor*), *Wazuh* is not primarily connected to the cloud interfaces, but its agents are installed directly on the (virtual) machines of the monitored infrastructure. The agents can run on many different platforms, such as Windows, Linux, Mac OS X, AIX, Solaris, and HP-UX. *Wazuh* includes several modules that each support their respective detection capability. For each of the modules, specific rules are defined that include internal metrics and thresholds to trigger events or alerts. When an alert is produced based on some detected event(s), additional actions can be triggered to notify a user or another component about it. With this capability, certain events (e.g., malware detected, *Wazuh* agent shutdown…), can trigger changes of values for specific MEDINA metrics and event-driven generation of evidence.

Wazuh's detection modules include the following:

- Occurrence of changes within system files (file integrity checks): *Wazuh* agent monitors the file system to detect changes in system files' content or attributes. Changes of system settings or other critical files can signify that the monitored machine is compromised.
- Detection of malware and rootkits installed on the infrastructure: *Wazuh* can scan the monitored system for various types of malware. It combines a signature-based approach for detecting suspicious programs with anomaly detection capabilities, detecting intrusions by monitoring system call responses. Signature-based malware detection is supported through integration with the open-source antivirus engine ClamAV [10] or VirusTotal [11], an online API for analysis of suspicious files.
- Number and severity of infrastructure vulnerabilities detected (e.g., CVE level of dependencies installed on the OS being monitored): *Wazuh* identifies the software installed on the monitored system and compares the versions with its online inventory in order to find software known to contain vulnerabilities.
- Monitoring cloud logs via IaaS or PaaS API: *Wazuh* includes modules for integration with some cloud providers' APIs (Amazon AWS, Azure, Google Cloud) to analyse security configuration of the cloud and notify about detected weaknesses.
- Compliance level with standards such as PCI DSS, HIPAA, GDPR: *Wazuh* integrates verification for some of the basic requirements of the mentioned standards. The *Wazuh* UI provides a dashboard with an overview of these requirements' fulfilment.

The main innovation of the usage of *Wazuh* and the extensions provided by MEDINA mainly lies in the flexibility of the proposed architecture. MEDINA can offer *Wazuh* and its extensions to the CSPs as a tool for incident detection and continuous monitoring of security indicators. Using *Wazuh*, compliance with several security controls can be automatically verified and the produced evidence integrated with the MEDINA framework. The controls that can be satisfied with *Wazuh* relate to malware protection, logging, threat analytics, and automatic monitoring (alerting). The final analysis of EUCS requirements covered by *Wazuh* is further described in D3.3 [2]. Beside the provided functionalities, *Wazuh* also offers a platform for implementing custom detectors on the monitored machines and easily integrating them with MEDINA.

An example of collecting evidence with *Wazuh* is provided here for verifying the fulfilment of (draft) EUCS requirement **OPS-05.3H**. This requirement reads: "*The CSP shall automatically monitor the systems covered by the malware protection and the configuration of the corresponding mechanisms to guarantee fulfilment of above requirements, and the antimalware scans to track detected malware or irregularities.*" (The above requirement refers to the other requirements from the OPS-05 control: Protection against malware – implementation). According to the descriptions of these requirements, the conditions for regarding a machine compliant with OPS-05.3H as verified by *Wazuh*, are:

- Enabled file integrity monitoring module
- Enabled malware and rootkit detection module
- Enabled integration with ClamAV or VirusTotal for additional malware protection
- At least one alerting service enabled in *Wazuh* to automatically notify the responsible persons in case of detected alerts

The first three conditions ensure that malware protection is enabled, while the last condition verifies that automatic monitoring is configured as well. To verify all conditions, the *Wazuh* and *VAT Evidence Collector* component makes several API queries to *Wazuh* for each of the (virtual) machines in scope. An evidence object is produced for each of the monitored machines with a measurement value according to the obtained result – positive if (and only if) all the mentioned conditions are satisfied.

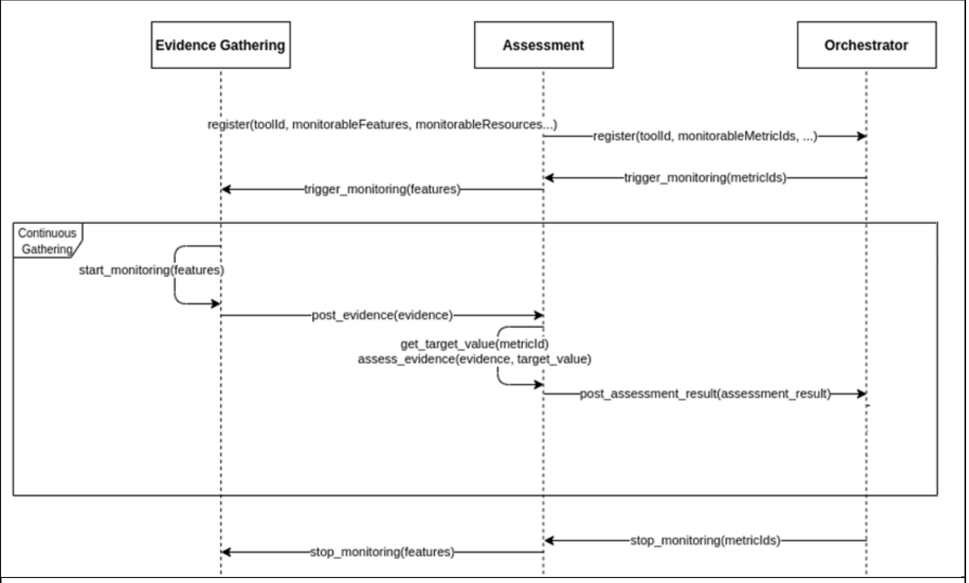
3.2.1.1.1 Fitting into overall MEDINA Architecture

Wazuh is integrated with the rest of the MEDINA framework through the *Wazuh and VAT Evidence Collector* component that gathers evidence from both *Wazuh* and *VAT*. *Wazuh* is installed inside the CSP's infrastructure and gathers information about possible security threats of the system. The state of *Wazuh*'s operation and the detected security events, gathered by *Wazuh*, are queried by the *Wazuh and VAT Evidence Collector*, which forwards such information to *Clouditor*'s *Security Assessment* in the form of evidence. Figure 1 shows the positioning of *Wazuh* in the MEDINA architecture among the WP3-related components.

3.2.1.1.2 Component cards

Table 9. Component card for Wazuh

<table>
<tr><th>Component Name</th><th><i>Wazuh</i></th></tr>
<tr><td>Main functionalities</td><td>In general, <i>Wazuh</i> is a HIDS solution that provides the following functionalities:<ul><li>Malware and intrusion detection</li><li>Log data analysis</li><li>File integrity monitoring</li><li>Vulnerability detection</li><li>Configuration assessment</li><li>(Limited) monitoring of data about AWS & Azure infrastructure with simple compliance assessment</li></ul>In MEDINA, <i>Wazuh</i> will be offered to the users as a tool to help CSPs satisfy compliance with certain EUCS controls as well as an evidence gathering tool.</td></tr>
<tr><td>Sub-components Description</td><td><i>Wazuh</i> is composed of a <i>Wazuh</i> server and <i>Wazuh</i> agents. The agents are deployed on the individual monitored machines and communicate information about the detected anomalies to the server.<ul><li>The <i>Wazuh</i> server includes the <i>Wazuh</i> manager component along with the ELK (ElasticSearch, Logstash, Kibana) stack for gathering, storing, and display of data. Custom integrations are possible to send alerts from <i>Wazuh</i> to any external component.</li><li>The <i>Wazuh</i> Agents communicate with the server using Rsyslog.</li></ul><i>Wazuh</i> is plugged into MEDINA with the <i>Wazuh and VAT Evidence Collector</i>, which is responsible for extracting the data relevant for MEDINA metrics, and transforming it into evidence, compatible with the <i>Security Assessment</i> component. It also includes two-way communication with the <i>Security Assessment</i> component (<i>Clouditor</i>).</td></tr>
<tr><td>Main logical Interfaces</td><td><table><tr><th>Interface name</th><th>Description</th><th>Interface technology</th></tr><tr><td>Wazuh WUI</td><td>Main web UI</td><td>Web, based on Kibana</td></tr><tr><td>ElasticSearch</td><td></td><td>ElasticSearch HTTP API (REST)</td></tr></table></td></tr>
<tr><td>Requirements Mapping</td><td>List of requirements covered by this component (see D5.2 [3]):
TEGT.C.01, TEGT.C.02, TEGT.S.08</td></tr>
<tr><td>Interaction with other components</td><td><table><tr><th>Interfacing Component</th><th>Interface Description</th></tr><tr><td><i>Wazuh and VAT Evidence Collector</i></td><td><i>Wazuh and VAT Evidence Collector</i> pulls information from the Wazuh server (custom integration sub-component). Interface technology is HTTP REST API.</td></tr></table></td></tr>
<tr><td>Relevant sequence diagram/s</td><td></td></tr>
</table>

	 Evidence Gathering
	Assessment
	Orchestrator
	register(toolId, monitorableFeatures, monitorableResources...)
	register(toolId, monitorableMetricIds, ...)
	trigger_monitoring(metricIds)
	trigger_monitoring(features)
	Continuous Gathering
	start_monitoring(features)
	post_evidence(evidence)
	get_target_value(metricId)
	assess_evidence(evidence, target_value)
	post_assessment_result(assessment_result)
	stop_monitoring(metricIds)
	stop_monitoring(features)

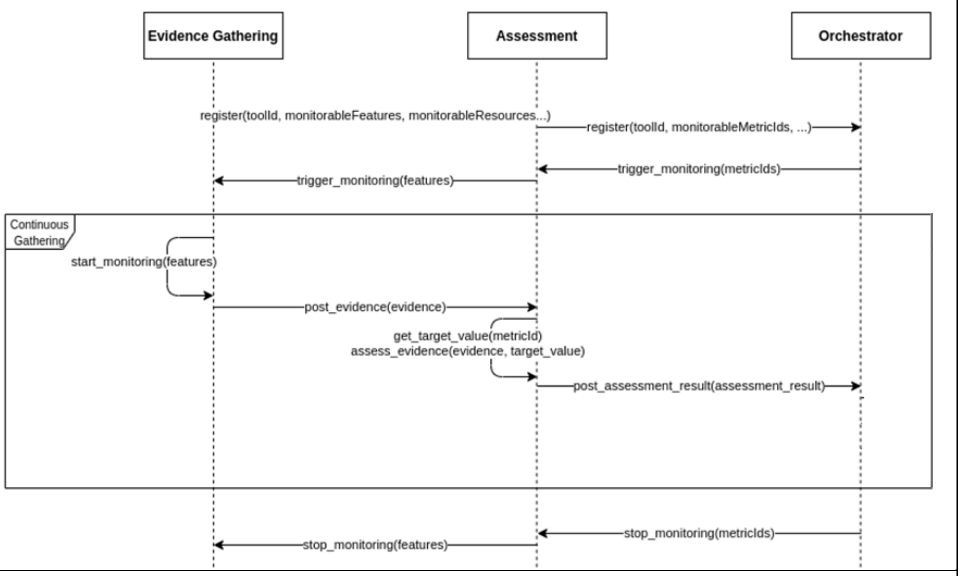
Current TRL [17]	Based on existing open-source *Wazuh* platform: TRL9.
Target TRL [18]	Based on existing open-source *Wazuh* platform: TRL9.
Programming language	C, Python, C++, Javascript
License	Open source: GNU GPL v2
WP and task	WP3, Task 3.2
MEDINA wokflows	WF2 "Preparation of MEDINA Components", and WF5 "EUCS Compliance Assessment" (see D5.4 [5])

Table 10. Component card for Wazuh and VAT Evidence Collector

Component Name	***Wazuh and VAT Evidence Collector***
Main functionalities	*Wazuh* and *VAT Evidence Collector* provides the following functionalities: • Collecting data from *Wazuh* • Creating scans and fetching scan results from *VAT* • Creating evidence based on data gathered from *Wazuh* and *VAT* • Forwarding evidence to the *Security Assessment* interface (*Clouditor*)
Sub-components Description	It is composed of following subcomponents: • Wazuh Evidence Collector: responsible for collecting data from *Wazuh* • VAT Evidence Collector: creating *VAT* scans and gathering data from scans • Clouditor Interface: forwards evidence to the *Security Assessment* (*Clouditor*) and is in charge of *Clouditor* Authentication and other communication with *Orchestrator*.
Main logical Interfaces	No endpoints.
Requirements Mapping	List of requirements covered by this component (see D5.2 [3]): TEGT.C.01, TEGT.C.02, TEGT.S.08

[17] TRL value before validation

[18] TRL value after validation

<table>
<tr><td><b>Interaction with other components</b></td><td>
<table>
<tr><th>Interfacing Component</th><th>Interface Description</th></tr>
<tr><td><i>Wazuh</i></td><td>Pulls data from <i>Wazuh</i> API</td></tr>
<tr><td><i>Wazuh</i> Elasticsearch</td><td>Pulls data from <i>Wazuh</i>`s Elasticsearch API</td></tr>
<tr><td><i>VAT</i></td><td>Post scan requests and pulls data from <i>VAT</i></td></tr>
<tr><td>Assessment Interface</td><td>Forwards evidence from <i>Wazuh</i> and <i>VAT</i></td></tr>
<tr><td>Authentication Interface</td><td><i>Wazuh</i> and <i>VAT</i> <i>Evidence Collector</i> authenticates with the <i>Clouditor</i></td></tr>
<tr><td><i>Orchestrator</i> Interface</td><td>Get a cloud service ID for <i>Wazuh</i> and <i>VAT</i></td></tr>
</table>
</td></tr>
<tr><td><b>Relevant sequence diagram/s</b></td><td>

Evidence Gathering
Assessment
Orchestrator
register(toolId, monitorableFeatures, monitorableResources...)
register(toolId, monitorableMetricIds, ...)
trigger_monitoring(metricIds)
trigger_monitoring(features)
Continuous Gathering
start_monitoring(features)
post_evidence(evidence)
get_target_value(metricId)
assess_evidence(evidence, target_value)
post_assessment_result(assessment_result)
stop_monitoring(metricIds)
stop_monitoring(features)
</td></tr>
<tr><td><b>Current TRL [19]</b></td><td>TRL5</td></tr>
<tr><td><b>Target TRL [20]</b></td><td>TRL6</td></tr>
<tr><td><b>Programming language</b></td><td>Python</td></tr>
<tr><td><b>License</b></td><td>Apache 2.0</td></tr>
<tr><td><b>WP and task</b></td><td>WP3, Task 3.2</td></tr>
<tr><td><b>MEDINA Workflows</b></td><td>WF2 “Preparation of MEDINA Components”, and
WF5 “EUCS Compliance Assessment” (see D5.4 [5])</td></tr>
</table>

3.2.1.1.3 Related requirements

Below is the collection of requirements (described in D5.2 [3]) related to the component and a description of how and to what extent these requirements are implemented at this point of development.

Requirement id	**TEGT.C.01**
Short title	Continuous collection
Description	The developed tools must be able to collect evidence continuously, i.e., in high-frequency intervals.

[19] TRL value before validation

[20] TRL value after validation

Implementation state	Fully implemented

Continuous collection is implemented for all selected metrics.

Requirement id	**TEGT.C.02**
Short title	Provision to defined interfaces
Description	The developed tools must provide collected evidence to a security assessment tool via its offered APIs.
Implementation state	Fully implemented

Interface between the *Wazuh* and *VAT Evidence Collector* components and *Clouditor* (providing the security assessment capabilities) is implemented.

Requirement id	**TEGT.S.08**
Short title	Provision of malware and vulnerability detection tools
Description	Tools for malware detection, intrusion detection, and vulnerability scanning must be provided to assist CSPs with satisfying related requirements of security standards or to verify the compliance with such requirements.
Implementation state	Fully implemented

Wazuh offers capability of malware scanning and vulnerability detection of the infrastructure and applications (in some cases). *Wazuh* agents pull software inventory data and send this information to the *Wazuh* Manager, where it is correlated with continuously updated CVE databases, in order to identify well-known vulnerable software. Automated vulnerability assessment helps the user identify the weak spots of their critical assets. Integration through the *Wazuh and VAT Evidence Collector* allows MEDINA to verify the malware detection state and gather evidence about it.

3.2.1.2 Technical description

The following subsections describe the technical details of *Wazuh*.

3.2.1.2.1 Prototype architecture

Wazuh is composed of a *Wazuh* server and multiple *Wazuh* agents. The agents are deployed on the individual monitored machines and communicate information about the detected anomalies to the server. In a cloud environment, the agents are deployed on the virtual machines inside the monitored cloud infrastructure, independent of the cloud provider. *Wazuh* server should be installed on a dedicated (virtual) machine, ideally in the same network segment as the agents or otherwise made available by the network routing rules.

The server includes the *Wazuh* manager component along with the ELK (ElasticSearch, Logstash, Kibana) stack for gathering, storing, and displaying data. Custom integrations are possible to send alerts from *Wazuh* to any external component.

The basic architecture of *Wazuh* is depicted in Figure 6. Looking at it from high-level, it consists of *Wazuh* Agents and *Wazuh* Server. The *Wazuh* agent (installed on endpoints) with different interfaces (modules) is able to detect different metrics on the host. The *Wazuh* Server consists of worker nodes (*Wazuh* cluster), a Kibana Server that provides a web user interface for overview of all logs and relevant events, and an ElasticSearch database server that stores the logs and detected events, coming from the agents.

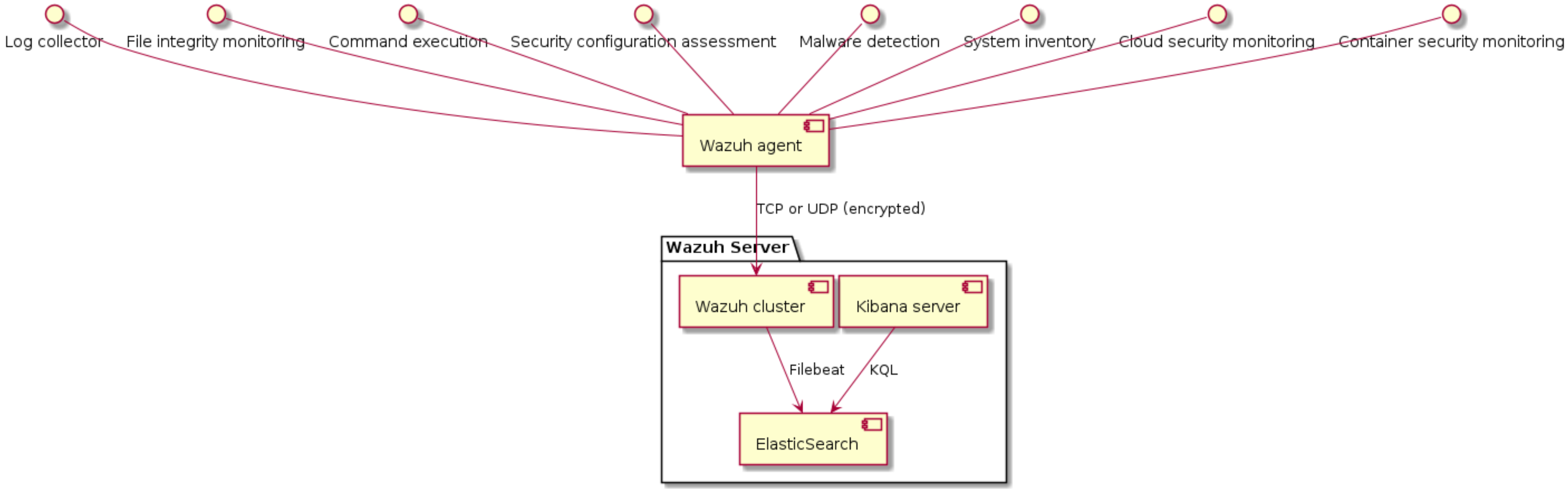


Figure 6. High-level Wazuh's architecture

3.2.1.2.2 Description of components

The components comprising *Wazuh* are described below.

Wazuh Agents communicate with the Wazuh server using Rsyslog. *Wazuh* is plugged into MEDINA with the *Wazuh and VAT Evidence Collector* component, which is responsible for extracting the data, relevant for MEDINA metrics, and transforming it into evidence compatible with the *Security Assessment* component. The Evidence Collector communicates with the *Wazuh* server using HTTP (API exposed by *Wazuh*). It also includes a two-way communication with the Security Assessment component (*Clouditor*). This is depicted below in Figure 7.

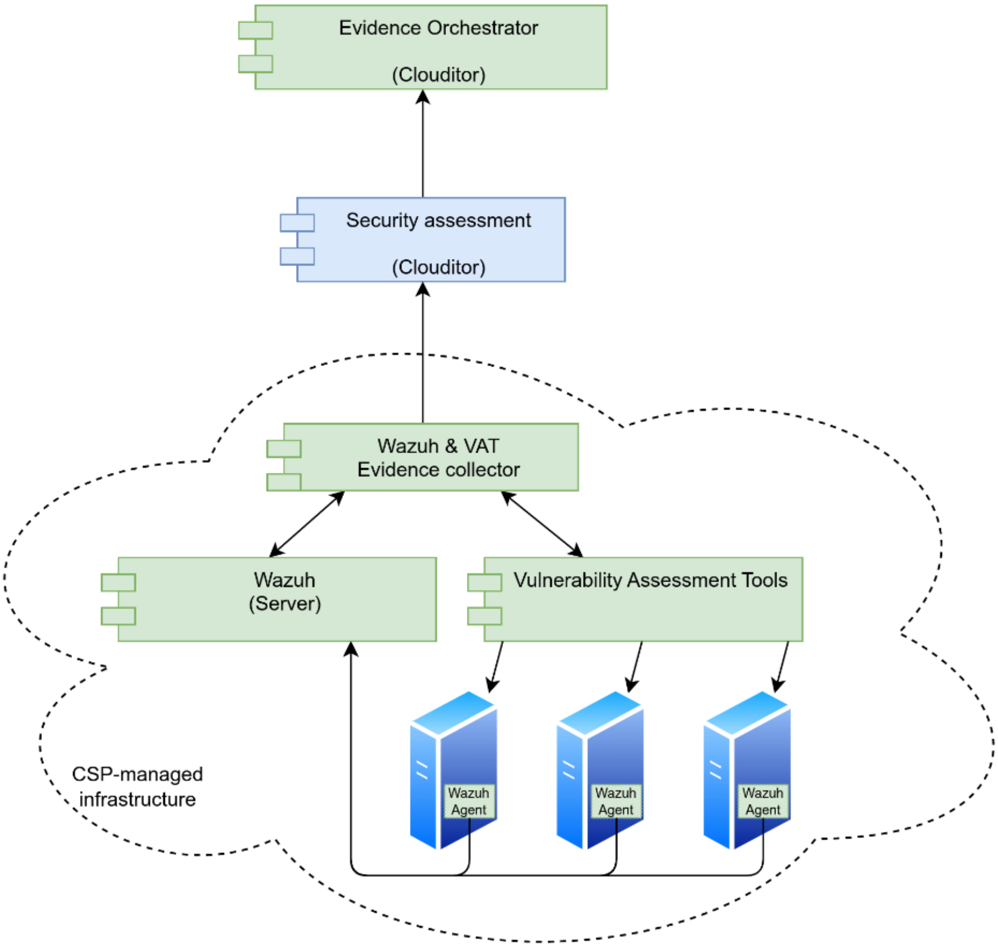


Figure 7. High-level schema of Wazuh, VAT, and related components

3.2.1.2.3 Technical specifications

The prototype's implementation consists of the following:

- MEDINA-specific deployment and configuration scripts for *Wazuh* (Ansible deployment scripts, YAML definitions, configuration). This also contains specific MEDINA configurations of *Wazuh* rules (XML, JSON).
- *Wazuh and VAT Evidence Collector*, a component that integrates *Wazuh* and *VAT* with the MEDINA *Security Assessment*. This component is developed in Python and packaged as a Docker container.

3.2.2 Delivery and usage

The following sections give a short overview of the delivery and usage of the *Wazuh* tool.

Please note that the README and installation instructions can be found in *Appendix F: Wazuh and VAT Evidence Collector - Readme and installation* instructions.

3.2.2.1 Package information

Tables 11 and 12 show the structure of the important folder with a brief description for *Wazuh* and the *Wazuh and VAT Evidence Collector*.

Wazuh deployment package

The *Wazuh* deployment package contains all the needed deployment and configuration scripts for installing *Wazuh*, *Wazuh and VAT Evidence Collector*, and also *Clouditor* (needed by the Evidence Collector to connect with). For demonstrative purposes and replicating a deployment on CSP's infrastructure, the process locally creates five virtual machines (using Vagrant): a *Wazuh* server, two *Wazuh* agents, *Wazuh and VAT Evidence Collector*, and *Clouditor* Security Assessment. Table 11 describes the important folders and files of this package.

Table 11. Overview of the Wazuh-deploy package structure

File / folder	Description
ansible/	This folder contains Ansible playbooks – scripts for installation of individual sub-components.
environments/	This folder contains several sets of configurations for different installation environments (based on the purpose, some components might not be installed or have various configuration applied – README file contains details).
custom-provision/	This folder contains the configuration set for installation on existing machines in the CSP's infrastructure.
Makefile	Contains simplified make scripts that trigger the installation procedures.
README.md	Contains details about installation requirements and instructions.

Wazuh and VAT Evidence Collector

The *Wazuh and VAT Evidence Collector* repository contains the source code of the connector between *Wazuh* (or *VAT*) and the MEDINA *Security Assessment* component (*Clouditor*). The source code is written in Python and contains a Docker file so that it can be simply built and used as a Docker container. Table 12 describes the important files and folders of this package.

Table 12. Overview of the Wazuh and VAT Evidence Collector package structure

File / folder	Description
clouditor_interface/	This folder contains code responsible for packaging and sending evidence objects to *Clouditor* (Security Assessment).
grpc_gen/	This folder contains code, automatically generated based on the protocol buffer definitions of the *Clouditor* Security Assessment gRPC interface.
id_maps/	This folder contains JSON files with ID maps for various services and resources.

File / folder	Description
kubernetes/	This folder contains Kubernetes definition files for automated deployment on the MEDINA Kubernetes dev & test clusters.
log_config/	This folder contains logging configuration code.
proto/	This folder contains the protocol buffer definitions of various *Clouditor* gRPC interfaces.
scheduler/	This folder contains code responsible for scheduling of the evidence gathering process.
test/	This folder contains code for (self-) testing of the component as a part of a CI pipeline.
vat_evidence_collector/	This folder contains the contains all the code related to VAT evidence gathering including VAT API client.
wazuh_evidence_collector/	This folder contains the core code responsible for the communication with *Wazuh* API and building (internal) evidence objects.
Dockerfile	This file contains the definition for building the Docker image of *Wazuh and VAT Evidence Collector*.
Licence	Contains Apache License 2.0
Manifest	Contains version and service info.
Makefile	Wrapper file containing all the commands and required variables to simplify CLI interactions with *Wazuh and VAT Evidence Collector*.
README.md	Contains installation and configuration instructions.
entrypoint.sh	Contains Docker image entrypoint script.
kubernetes_clouditor_demo.env	Contains Kubernetes environment variables - used for local testing purposes.
requirements.txt	Contains required libraries, modules, and packages information.

3.2.2.2 Installation instructions

Requirements:

- Vagrant 2.2.14
- Ansible 2.9.16

Clone the *Wazuh* deployment repository (see 3.2.2.5 below). To setup the demo, simply provision the *Wazuh* server, *Wazuh* agents, *Clouditor*, and Evidence Collector virtual machines by running:

```
make create provision
```

For other installation options, please consult the README file in the repository.

3.2.2.3 User Manual

After installation on a local Vagrant environment, the *Wazuh* UI can be accessed by navigating a web browser to https://192.168.33.10:5601 (if using the default deployment configuration). Default credentials (admin:changeme) can be used for logging in the web interface.

After accessing the “Wazuh” section in the web UI, the user can notice two agents registered and running with *Wazuh*. Evidence Collector is configured to collect evidence about the malware detection running on the agent machines every minute. This can be inspected by examining the logs of the Evidence Collector virtual machine.

3.2.2.4 Licensing information

The core *Wazuh* [9] component is open source, licensed with a modified GPLv2 license[21].

The deployment scripts for the MEDINA proof-of-concept and the *Wazuh and VAT Evidence Collector*, developed by XLAB, are licenced with the open-source Apache Licence 2.0.

3.2.2.5 Download

The code of open-source components built by MEDINA is available on the project's git repository, on GitLab hosted by TECNALIA, for *Wazuh and VAT Evidence Collector*[22] and for *Wazuh* deployment repository[23].

3.2.3 Advancements within MEDINA

Wazuh is a software solution developed independently of MEDINA by its respective owner, Wazuh Inc. In the scope of MEDINA, several advancements have been made in terms of integration with MEDINA, associated configurations and implementation of the *Wazuh and VAT Evidence Collector* component. The (non-exclusive) list of work done in the project is as follows:

- Analysis of the EUCS requirements to determine which of the requirements can be verified or satisfied by using *Wazuh*.
- Definition of architecture for collecting evidence with *Wazuh* and integrating it with MEDINA (*Wazuh and VAT Evidence Collector*).
- Implementation of *Wazuh and VAT Evidence Collector* with connection to *Clouditor Security Assessment* service.
- Implementation of CI/CD pipelines for automatic installation on the MEDINA "dev" and "test" deployments.
- Production of deployment scripts to enable easier installation of software on the pilots' infrastructure.
- Improvements of *Wazuh and VAT Evidence Collector* to support multiple metrics.
- Implementation of changes related to advancements in the MEDINA data model.

3.2.4 Limitations and future work

Evidence gathering with *Wazuh* is currently possible for a limited number of metrics related to the (draft) EUCS requirement OPS-05.3H.

Wazuh uses various techniques for evidence gathering. By using the integrated anti-malware and intrusion detection systems, a CSP is satisfying the standardisation requirements. In this case, evidence is produced bearing the information about the functioning of *Wazuh* and its modules. Such evidence has a high level of confidence. If the CSP uses other (unrelated) tools for malware detection, the limitation is that an integration layer needs to be developed between those tools and *Wazuh*. While *Wazuh*'s log collection capabilities make such integration relatively easy with most tools, support by the other tool might be limited.

Custom *Wazuh* rules can also be written to evaluate logs, coming from other services and produce events or alerts based on their contents. Evidence can in turn be produced based on

[21] https://github.com/wazuh/wazuh/blob/master/LICENSE
[22] https://git.code.tecnalia.com/medina/public/wazuh-vat-evidence-collector
[23] https://git.code.tecnalia.com/medina/public/wazuh-deploy

such events or alerts. The level of confidence obtained in this way is fully dependent on the implementation of the *Wazuh* particular rule.

3.3 Vulnerability Assessment Tools

Vulnerability Assessment Tools (VAT) act as a vulnerability scanning and detection framework. *VAT* is intended to be deployed in the CSP's infrastructure and configured to periodically scan the machines and servers on the monitored network, using several tools to detect vulnerabilities.

3.3.1 Implementation

The following subsections provide functional and technical descriptions of *VAT*.

3.3.1.1 Functional description

VAT's collection of vulnerability scanning tools comprise two web vulnerability scanners (W3af [12] and OWASP ZAP [13]), a network discovery and auditing tool Nmap [14], and a framework for including user-defined custom scripts for detecting specific issues or simply notifying about unavailability of particular services.

Beside the vulnerability scanners, *VAT* is composed of several components supporting the scheduling of scanning tasks, definition of custom scripts for scanning or monitoring, as well as communication and integration with other MEDINA tools.

The innovation that *VAT* brings to MEDINA is the usage of vulnerability scanners for automated verification of compliance. There are several requirements of security standards that can be either satisfied with *VAT* or evidence that can be gathered about their fulfilment. EUCS requirements covered by *VAT* include vulnerability detection and management categories and the usage of encrypted communication protocols.

A more detailed coverage of EUCS requirements by *VAT* is also described in deliverable D3.3 [2].

3.3.1.1.1 Fitting into overall MEDINA Architecture

The position of *Vulnerability Assessment Tools* inside the MEDINA architecture is depicted in Figure 1 (section 2) and in slightly more detail in Figure 7. *VAT* scans the monitored machines inside the CSP's infrastructure, which is communicated to the *Wazuh and VAT Evidence Collector* component that constructs the evidence about fulfilment of the monitored metrics and sends them to the *Security assessment* component (*Clouditor*) for further processing.

3.3.1.1.2 Component card

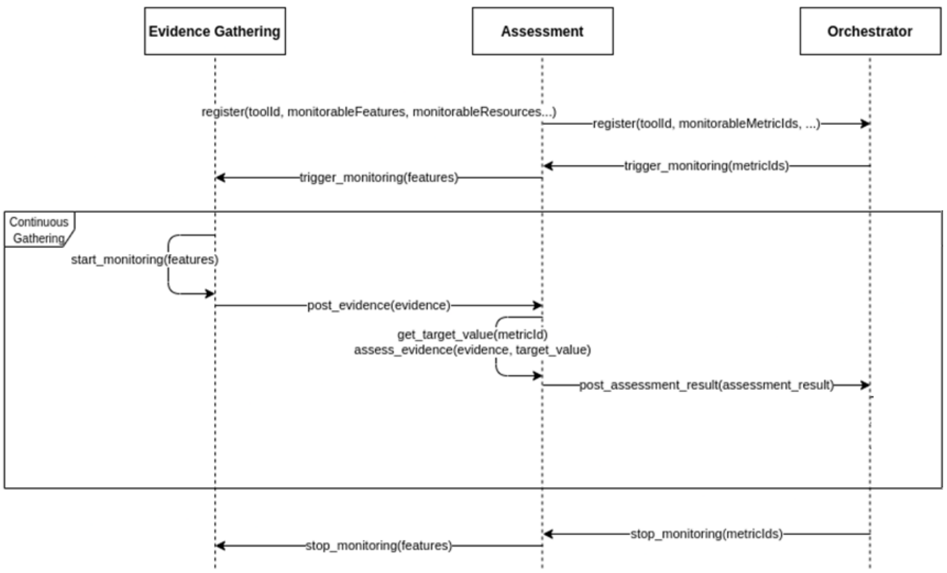
Table 13. The component card for the Vulnerability Assessment Tools

Component Name	***Vulnerability Assessment Tools (VAT)***
Main functionalities	The component provides the following functionalities: • Detection of web vulnerabilities by running integrated vulnerability scanners to scan web applications (OWASP ZAP[24], w3af[25]) • Network reconnaissance (running hosts, open ports – exposed services) using integrated Nmap[26] • Detection of vulnerable software (known vulnerable service versions) • Running custom scripts for detection of specific vulnerabilities

[24] https://owasp.org/www-project-zap/
[25] http://w3af.org/
[26] https://nmap.org/

<table>
<tr><td></td><td><ul><li>Scheduling repeating vulnerability scans</li></ul>In MEDINA, <i>VAT</i> will be offered to the users as a tool to help CSPs satisfy compliance with certain EUCS controls as well as an evidence gathering tool.</td></tr>
<tr><td><b>Sub-components Description</b></td><td><ul><li><b>Scheduler</b>: responsible for triggering scanning tasks according to the configured schedules</li><li><b>Docker interface</b>: a component managing the connection with the Docker runtime, executing the tasks by running appropriate docker images and obtaining their results</li><li><b>Frontend</b>: web UI management interface</li><li><b>RabbitMQ</b>: connection between the subcomponents</li><li><b>VAT-genscan</b>: integrating and orchestrating some vulnerability scanning tools and combining their results into a common report (based on Faraday CSCAN[27])</li><li><b>Wazuh and VAT Evidence Collector</b>: a component responsible for extracting the data relevant for MEDINA metrics, and transforming it into evidence, compatible with the <i>Security assessment</i> component. It also includes a communication with the <i>Security assessment</i> component (<i>Clouditor</i>).</li></ul></td></tr>
<tr><td><b>Main logical Interfaces</b></td><td><table>
<tr><th>Interface name</th><th>Description</th><th>Interface technology</th></tr>
<tr><td>Scan reports output</td><td>Pushing the results of scan tasks (vulnerability reports)</td><td>RabbitMQ (AMQP), JSON</td></tr>
<tr><td>Management UI</td><td>Web UI to manage the scanning tasks and review their results</td><td>Web</td></tr>
</table></td></tr>
<tr><td><b>Requirements Mapping</b></td><td>List of requirements covered by this component (see D5.2 [3]):
TEGT.C.01, TEGT.C.02, TEGT.S.08</td></tr>
<tr><td><b>Interaction with other components</b></td><td><table>
<tr><th>Interfacing Component</th><th>Interface Description</th></tr>
<tr><td><i>Wazuh and VAT Evidence Collector</i></td><td><i>Wazuh and VAT Evidence Collector</i> pulls reports from <i>VAT</i>. Interface technology is HTTP REST API.</td></tr>
</table></td></tr>
</table>

[27] https://github.com/infobyte/faraday/tree/master/scripts/cscan

Relevant sequence diagram/s	 Evidence Gathering
	Assessment
	Orchestrator
	register(toolId, monitorableFeatures, monitorableResources...)
	register(toolId, monitorableMetricIds, ...)
	trigger_monitoring(metricIds)
	trigger_monitoring(features)
	Continuous Gathering
	start_monitoring(features)
	post_evidence(evidence)
	get_target_value(metricId)
	assess_evidence(evidence, target_value)
	post_assessment_result(assessment_result)
	stop_monitoring(metricIds)
	stop_monitoring(features)

Current TRL [28]	Based on existing *Vulnerability Assessment Tools* component developed in the scope of H2020 CYBERWISER[29], which been adapted and integrated in the MEDINA framework: the TRL is 4. Integrated vulnerability scanners used are separately developed components by their respective owners: their TRLs are higher (8-9).
Target TRL [30]	*VAT* integrated in the MEDINA framework: TRL 6
Programming language	Go, node.js, Javascript, Python, Bash.
License	The *VAT* platform is proprietary, closed-source (developed by XLAB). Some sub-components and integrated vulnerability scanning tools are open-source: • OWASP ZAP: Apache License • W3af: GNU GPL v2 • Nmap: Nmap Public Source License based on GNU GPL v2
WP and task	WP3, Task 3.2
MEDINA Workflows	WF2 "Preparation of MEDINA Components", and WF5 "EUCS Compliance Assessment" (see D5.4 [5])

3.3.1.1.3 Related requirements

Below is the collection of requirements (from D5.2 [3]) related to the component and a description of how and to what extent these requirements are implemented at this point of development.

Requirement id	**TEGT.C.01**
Short title	Continuous collection
Description	The developed tools must be able to collect evidence continuously, i.e., in (high)-frequency intervals.

[28] TRL value before validation

[29] CYBERWISER.eu | Cyber Range & Capacity Building in Cybersecurity

[30] TRL value after validation

Implementation state	Fully implemented

VAT framework enables configuration of the scanning tasks and continuous scanning with adjustable intervals and manually configured metrics. Evidence is currently collected from the results of the generic (integrated) vulnerability scanners.

Requirement id	**TEGT.C.02**
Short title	Provision to defined interfaces
Description	The developed tools must provide collected evidence to a security assessment tool via its offered APIs.
Implementation state	Fully implemented

The interface for communication between the component core API, the *VAT & Wazuh Evidence Collector*, and *Clouditor* (*Security Assessment*) is implemented.

Requirement id	**TEGT.S.08**
Short title	Provision of malware, intrusion, and vulnerability detection tools
Description	Tools for malware detection, intrusion detection, and vulnerability scanning must be provided to assist CSPs with satisfying related requirements of security standards or to verify the compliance with such requirements.
Implementation state	Fully implemented

VAT includes several vulnerability scanners and a framework for their orchestration and automated running of the scans. The possibility to add custom vulnerability scanning scripts is also implemented.

3.3.1.2 Technical description

The following subsections describe the technical details of the *Vulnerability Assessment Tools.*

3.3.1.2.1 Prototype architecture

The internal architecture of the *Vulnerability Assessment Tools* consists of several microservices (see Figure 8). The main components are: Scan Configurator (web user interface), Vulnerability Scanning Registry, Catalogue of custom scripts, and VAT Service Orchestrator with several subcomponents. The figure also shows an example of a user's request to issue a scan originating in web interface and the data flow through the other *VAT* subcomponents. The connection to other MEDINA components for evidence gathering is issued through the *Wazuh and VAT Evidence Collection* component (see also Figure 7), which communicates with the *VAT* Service Orchestrator API.

3.3.1.2.2 Description of components

The components comprising Vulnerability Assessment Tools are described below.

Scan Configurator

A web interface for *Vulnerability Assessment Tools*. It enables users to configure and trigger vulnerability scans, set schedules for scanning tasks, review tasks' results, as well as create custom vulnerability detection scripts.

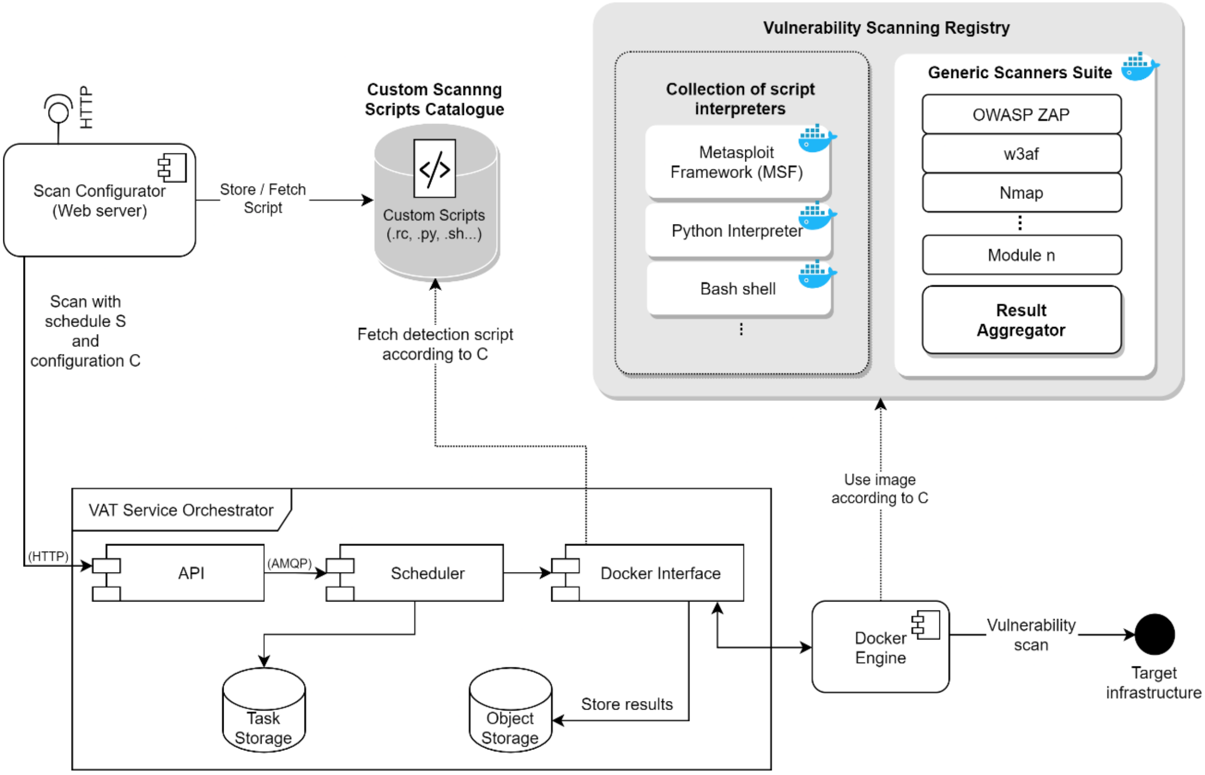


Figure 8. Internal architecture schema of the Vulnerability Assessment Tools

Custom Scanning Scripts Catalogue

The custom vulnerability detection scripts are stored in the Custom Scanning Scripts Catalogue. They can be written in any of the scripting languages, supported by the script interpreters included in the Registry. The Catalogue can also store script templates that need to have some missing parameters or code added before execution.

Vulnerability Scanning Registry (with Generic Scanners Suite and Result Aggregator)

A collection of Docker images for running vulnerability scans. It contains a Generic Scanners Suite image with several integrated scanning modules and a Result Aggregator component that combines results of the scanning modules into a single JSON result that can be shown in the Scan Configurator UI in a user-friendly way. The integrated scanning modules are OWASP ZAP [13], w3af [12], and Nmap [14]. ZAP and w3af are web application vulnerability scanners. When a scan is launched against a targeted website, they use crawlers to scan the website and identify potentially vulnerable pages and endpoints. For the detection of injection vulnerabilities, they use crafted payloads in automatic queries and observe the server's output, searching for patterns that would indicate potential vulnerabilities. Several server misconfiguration weaknesses can also be detected. Nmap is a network reconnaissance tool with vulnerability scanning capabilities. It can detect devices on the network and servers (listening ports) running on them, identify versions of the running servers and use various scripts to remotely detect specific vulnerabilities.

Collection of script interpreters

Beside the Generic Suite, the Vulnerability Scanning Registry also holds several Docker images [15] of script interpreters that can run user-provided custom scanning scripts. This Collection of script interpreters can be used to detect specific vulnerabilities, to monitor uptime and availability of services, or for other repetitive tasks. Three interpreters are currently included: Metasploit Framework [16] (running Metasploit resource scripts and Metasploit modules written in Ruby), Python, and Bash. The Scanning Registry is designed in a modular way, so that additional scanners or script interpreters can be added easily.

The functionality of custom scanning scripts can be used to gather evidence about any metric obtainable by executing some code. For example, such scripts can access a CSP-internal API, check whether some server is available, search for errors in logs obtained from another server, etc. The script should return the result either in its exit code or as contents of a file. The result is then used by *VAT* to construct an evidence object with the pre-configured metric identifier.

VAT Service Orchestrator

This component contains several subcomponents responsible for scheduling and orchestration of scans, as well as communication with other components. Internal communication between components is realized through the AMQP protocol by a RabbitMQ [17] server (not shown in Figure 8 for clarity). The Scan Configurator server communicates with the core components through the API, which also provides authentication and authorization capabilities. The API component is also accessed by the *Wazuh and VAT Evidence Collector* component (see Figure 7), which generates evidence objects according to the configuration and results of *VAT* and forwards them to the *Security Assessment* component (part of *Clouditor*) to be in turn processed by other MEDINA components.

Scheduler

A component responsible for triggering scanning tasks according to their configured schedules. A Task Storage database is used to store the schedules and configurations. When a specific task is triggered, it communicates its configuration to the Docker Interface component that prepares the required files and parameters and executes the container spawned with the respective Docker image from the Registry in the Docker Engine. The Docker Interface also retrieves results of the finished scanning tasks and stores their output files in the Object Storage database, from where it can be retrieved by users.

3.3.1.2.3 Technical specifications

The various subcomponents of *VAT* use different programming languages, frameworks, and libraries. The backend components are mostly written in Node.js, except Scheduler which is written in Go. MongoDB [18] is used for the Task Storage, and OpenStack Swift [19] for the Object Storage and storage of custom scanning scripts. Scan Configurator frontend is built with the Angular [20] web framework.

The Generic Scanning Suite is built as a single Docker image with Ubuntu as base image with required scanning modules installed (OWASP ZAP [13], w3af [12], Nmap [14]). The Result Aggregator is written in Python and outputs a JSON file containing outputs of all the scanning modules used.

3.3.2 Delivery and usage

3.3.2.1 Package information

The code of Vulnerability Assessment Tools is structured in several Git repositories according to the components described above. All components are packaged as Docker images.

The component acting as a bridge of *Vulnerability Assessment Tools to MEDINA*, *Wazuh* and *VAT Evidence Collector*, is described in section 3.2.2.1.

3.3.2.2 Installation instructions

Deployment scripts are provided using Vagrant [21] and Ansible [22] in the “vat-deployment” repository. A single virtual machine is provisioned running all the necessary services for *VAT*.

To run the demo deployment process, simply clone the repository and run:

```
make create provision
```

3.3.2.3 User Manual

By navigating an internet browser to the IP address of the management machine, the user can access the *VAT* configuration portal, review the demonstrative vulnerability scans, or create new scanning tasks.

3.3.2.4 Licensing information

Vulnerability Assessment Tools framework is licensed as proprietary, Copyright by XLAB.

The Generic Scanning Suite, a containerized component integrating several vulnerability scanners, is developed by XLAB and open-sourced with the Apache Licence 2.0.

Several sub-components used as part of *VAT* are open source:

- OWASP ZAP (Apache Licence) [13]
- w3af (GPLv2) [12]
- Nmap (modified GPLv2)[31] [14]
- Faraday (GPLv3) [23]
- Metasploit (BSD) [16]

Wazuh and VAT Evidence Collector, developed by XLAB, is available open source (Apache Licence 2.0).

3.3.2.5 Download

VAT deployment demo repository is available at MEDINA’s GitLab[32], along with the Generic Scanning Suite source code repository[33].

Due to proprietary licensing, other parts of the *VAT* framework are hosted on XLAB’s internal GitLab. The code can be made available upon request.

Source code of the individual included scanners can be found in their respective project repositories.

Source code of the *Wazuh and VAT Evidence Collector* is available in a separate repository[34].

[31] https://nmap.org/npsl/
[32] https://git.code.tecnalia.com/medina/public/vat-deploy
[33] https://git.code.tecnalia.com/medina/public/vat-genscan
[34] https://git.code.tecnalia.com/medina/public/wazuh-vat-evidence-collector

3.3.3 Advancements within MEDINA

Vulnerability Assessment Tools were developed in a previous H2020 project, CYBERWISER.eu [24]. In that project, the *VAT* framework was used for the detection of vulnerabilities as well as for the scheduling of various actions connected to defence and attacks of infrastructure in a controlled and enclosed cyber range environment.

In the initial phase of MEDINA, an analysis was made to determine the EUCS requirements that were feasible to be verified or satisfied by *VAT*. Later, the internal architecture of *VAT* was restructured, and the deployment scripts were rewritten to support the deployment in a general (cloud) environment instead of the specific cyber range setting. The APIs were adapted to be prepared for the integration with the MEDINA components. The *Wazuh and VAT Evidence Collector* component was developed with a specific module to interact with *VAT* and produce relevant evidence.

In the scope of MEDINA, the following advancements have been made so far:

- Analysis of the EUCS requirements, feasible to be verified or satisfied by *VAT*.
- Restructuring of the internal *VAT* architecture.
- Deployment scripts adapted and rewritten to support deployment in a general (cloud) environment.
- Adaptation of APIs.
- Development of the *Wazuh and VAT Evidence Collector* with the connection to *Clouditor* (*Security Assessment* and *Orchestrator*) and a *VAT*-interface module.
- Adaptation of the web interface and authorization modules.

Based on the feedback from the first-round validation in WP6 in 2023, a more detailed Readme for *Wazuh and VAT Evidence Collector* was produced that better describes the installation and configuration of the component (see *Appendix F: Wazuh and VAT Evidence Collector - Readme and installation* instructions).

3.3.4 Limitations and future work

As described above, *VAT* is composed of multiple modules: several vulnerability scanners and also a framework for running custom, user-defined evidence collection scripts. Confidence of the evidence gathered with *VAT* can vary greatly depending on the *VAT* module used and the definition of a specific metric. The generic vulnerability scanners (e.g., w3af, OWASP ZAP) are primarily designed to be used in manually guided penetration tests. Thus, vulnerability detection results can often contain false positives that should be analysed by an expert. Evidence collected solely based on the results of such results therefore cannot be regarded with full confidence.

On the other hand, there are considerably less errors when a vulnerability detection tool is configured to check for the presence of a specific vulnerability (e.g., Nmap or Metasploit script). The accuracy of custom (user-provided) scripts entirely depends on their implementation, in this case *VAT* is only used as a framework for running such scripts and packaging and forwarding the results as evidence.

Some requirements of the EUCS standardisation framework require the CSP to have vulnerability tools deployed on certain systems and to monitor their results. By using the vulnerability scanning capabilities of *VAT* (combined with monitoring of the results), the CSP effectively satisfies such requirements for their cloud service. In this case, the automatically obtained evidence refers to the functioning of *VAT*, which can be managed effectively and monitored with high confidence.

Future work would be to exploit the modularity and flexibility of the *VAT* framework. Beside the included vulnerability detection tools, users could define their own scripts written in one of the several supported programming languages, or even integrate their own vulnerability scanning tools, depending on their specific needs. *VAT*'s feature of including custom scripts for monitoring metrics would enable the user to easily provide their own code for checking specific metrics. Ideally, the output of such code (script) would be automatically transformed into evidence and integrated into the MEDINA workflows.

This scenario was already mentioned in the deliverable D3.5 [3] but the process of integrating *VAT* into MEDINA framework using the *Wazuh and VAT Evidence Collector* showed that the user-provided custom scripts could not work properly in the current MEDINA framework setup. It was therefore decided to stay with the (pre)configured *VAT* scans.

4 Security Assessment of Cloud Applications

This section presents the MEDINA components related to estimating the security of cloud applications and collecting evidence based on the analysis of their source code. The functionalities and implementation of the two components under development in Task 3.3 (*Cloud Property Graph*, *Codyze*, and extensions) are described in the following subsections.

4.1 Cloud Property Graph

The *Cloud Property Graph* (*CloudPG*) is a further tool, developed within the first year of the MEDINA project and improved and extended until M30. It combines static source code analysis with cloud infrastructure analysis. To that end, a library for static code analysis, the cpg[35], has been extended with analysis logic for cloud workloads. The implementation of this tool is developed in an open-source repository on GitHub[36].

One problem the *CloudPG* addresses is that isolated security analysis on workload- or source code-level can result in many false positives: for example, authorization or encryption requirements may be implemented either on the infrastructure- or source code-level, thus both levels have to be analysed in combination to allow for a comprehensive assessment of, e.g., authorization or encryption requirements.

4.1.1 Implementation

The following subsections provide functional and technical descriptions of the *Cloud Property Graph* tool.

4.1.1.1 Functional description

The cpg library, which forms the basis of the *CloudPG*, creates a property graph of source code that is enhanced by the *CloudPG* with information about the current resource configurations. Also, data flows between resources are added to the graph, e.g., HTTP requests between microservices for an excerpt of a graph generated by the *Cloud Property Graph*. Figure 9 shows several nodes and edges introduced by the *CloudPG*, for example security properties (based on the cloud resource ontology), like authenticity and transport encryption (see top left), and HTTP calls: the POST node describes a HTTP POST request to (TO edge) a HTTP endpoint that in turn has a certain path as its PROXY_TARGET.

This way, it is possible to identify security problems in the intersection between infrastructure and source code. For example, it can be detected if logging functionalities are implemented and if yes, if the logs are stored in an allowed region. This combined reasoning would be more difficult to do when assessing isolated evidence about source code and cloud workloads.

To also enable the detection of privacy threats, the *CloudPG* implementation has been extended with the following features: improved detection of data flows in HTTP connections, taint tracking via dedicated labels, detection of cryptographic libraries (e.g., for cryptographic signatures), detection of database operations, and more. This extension for semi-automatic detection of privacy threats has been published in [25].

Since this combined analysis requires a common model of how, e.g., logging functionalities are implemented, and what they are called in different cloud systems, the *CloudPG* again makes use of the cloud resource ontology presented in deliverable D2.5 [6], and the security properties it

[35] https://github.com/Fraunhofer-AISEC/cpg
[36] https://github.com/clouditor/cloud-property-graph/

defines. Consequently, security-relevant concepts can be analysed across source code and infrastructure configurations.

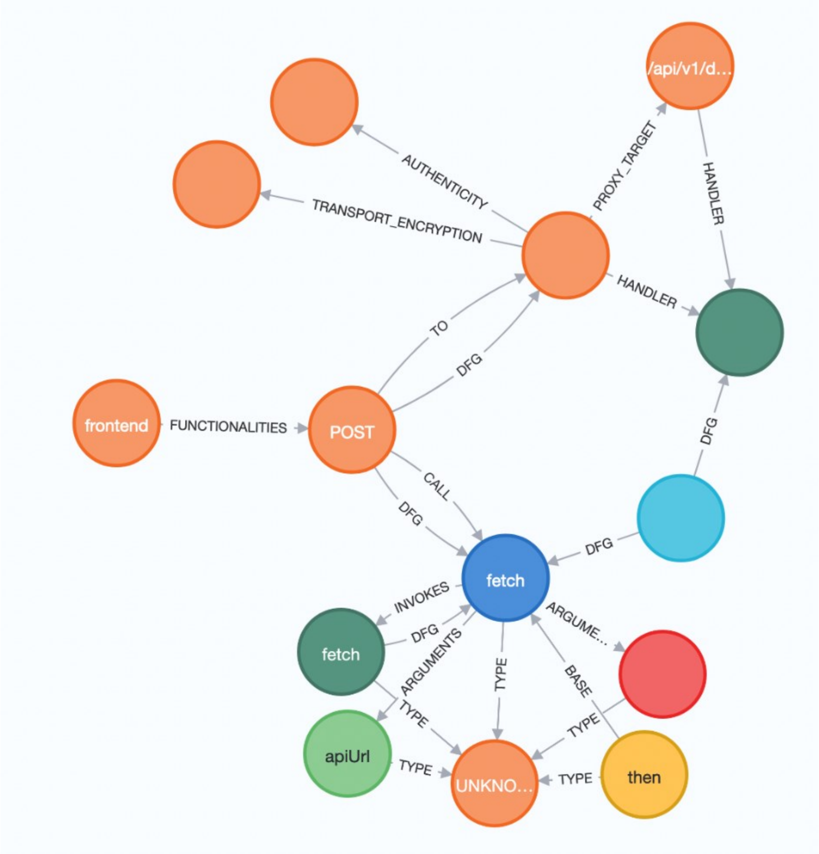


Figure 9. An excerpt from the graph generated by the Cloud Property Graph

4.1.1.1.1 Fitting into overall MEDINA Architecture

The *CloudPG* is a separate component implementing evidence gathering and assessment. As it is a new research approach, it is not integrated with other MEDINA components. Two possible approaches exist for its integration: First, its output may be adjusted to provide evidence in the required MEDINA format to the *Clouditor Security Assessment*. In this case the *CloudPG*'s graph-based analysis results would have to be transformed to the respective evidence format. Second, it can be extended with a custom assessment service which is then integrated with the *Orchestrator*. This latter approach has the advantage of leaving more room for custom assessment logic but may require more effort.

4.1.1.1.2 Related requirements

The relevant requirements from the deliverable D5.2 [3] are listed below with a brief description of how they are implemented.

Requirement id	TEGT.S.02
Short title	Collect evidence from source code via CPG
Description	The developed tool must be able to parse the source code of cloud applications written in different programming languages and transform into the agnostic representation of the CPG.
Implementation state	Fully implemented

Requirement id	TEGT.S.03
Short title	Implement information and data flow analysis
Description	The developed tool must be able to perform information and data flow analysis on a cloud application.
Implementation state	Fully implemented

Similar to *Codyze*, the *CloudPG* is able to analyse source code regarding data flows, control dependence, and other properties.

Requirement id	TEGT.S.10
Short title	Connect infrastructure- and application-level security analyses
Description	The developed tool should be able to bridge the gap between infrastructure- and application-level security analysis by extending graph-based code analysis to the cloud resources, allowing to identify data flows across cloud resources.
Implementation state	Fully implemented

This requirement describes the core idea of the *CloudPG*: by combining both source code analysis and deployment information, an accurate assessment of security properties can be made.

Requirement id	TEGT.S.07
Short title	Support for common programming languages, libraries, cloud services
Description	The developed tool should support common programming languages, libraries and cloud services. Support for all programming languages, libraries and cloud services is infeasible.
Implementation state	Fully implemented

Similar to *Codyze*, the *CloudPG* supports multiple programming languages, including Java, C, Python, Go, and Typescript.

Note that the common requirements (TEGT.C.X) are currently not relevant for this tool. They specify, for instance, that the tools need to comply to the MEDINA data model. The *Cloud Property Graph*, however, is still a novel research concept, whose integration into the framework needs to be designed and implemented in future work.

4.1.1.2 Technical description

The following subsections describe the technical details of the *CloudPG*.

4.1.1.2.1 Prototype Architecture

The *CloudPG*'s architecture is based on the architecture of the underlying cpg library. First, it uses the cpg library to build a code property graph of the given source code. It then applies custom *passes*, i.e., extendible modular analysis logic, to analyse properties of the code and its deployment that are relevant in the context of (cloud) security. This added information is then introduced in the graph to make it accessible for manual analysis and possibly automatic analysis applications.

Some examples of such custom passes are presented in the following:

- HTTP calls: The *CloudPG* analyses code to detect HTTP calls between microservices and adds edges to the graph between the respective nodes, e.g., from a code entity that uses an HTTP framework to realize the HTTP call to the respective HTTP endpoint. HTTP endpoints are identified in another pass which is able to detect these in the Spring framework for Java, the Flask framework for Python, as well as the Gin framework for Go.
- Logging: The *CloudPG* detects logging functionality, such as the zerolog[37] library for Go.
- Deployment information: The *CloudPG* detects GitHub workflow files in a project, which specifies where the code is deployed, e.g., as Docker containers in a Kubernetes cluster, and adds configuration information about the deployment environment.
- Databases: The *CloudPG* identifies connections and other operations related to databases, such as MongoDB and PostgreSQL.
- Label extraction: In the context of improvements of the *CloudPG* for privacy analysis, labels have been added to mark personal data (or otherwise sensitive data), which can be extracted by a dedicated pass to track their flow across the application.

4.1.1.2.2 Description of components

The *CloudPG* is not divided into separate components. A separate assessment component may be developed in the future. It does, however, employ an easily extendible structure for additional passes.

4.1.1.2.3 Technical specifications

The *CloudPG* is written in Kotlin. As described above, it makes use of the cpg library to build a basic code property graph.

Please note that the user manual and installation instructions of *CloudPG* can be found in *Appendix D: Cloud Property Graph - Installation instructions and User* manual.

4.1.2 Delivery and usage

The following sections give a short overview of the delivery and usage of the *CloudPG*. Please note that additional instructions can be found in *Appendix D: Cloud Property Graph - Installation instructions and User* manual.

4.1.2.1 Package

The tool is not yet available as a Docker image. It currently needs to be installed as described in *Appendix D: Cloud Property Graph - Installation instructions and User* manual.

4.1.2.2 Licensing

The tool is licensed under the open-source Apache License 2.0.

4.1.2.3 Download

The project is available open source on GitHub[38].

4.1.3 Advancements within MEDINA

The *Cloud Property Graph* is based on the cpg, which is a project developed independently of MEDINA. The *CloudPG*'s additions described above, however, have completely been developed

[37] https://pkg.go.dev/github.com/rs/zerolog

[38] https://github.com/clouditor/cloud-property-graph/

within the MEDINA project. In its approach of combining source code analysis with infrastructure analysis, it complements *Codyze* (see section 4.3).

During the second and third iteration of the MEDINA components' development, the *CloudPG* was extended with dedicated privacy analysis functionalities. To that end, the various privacy goals defined in the LINDDUN framework [26] were analysed, operationalized, and translated to code properties. Finally, respective passes have been integrated, e.g., for database connections, specific HTTP connections, privacy labels, and more. Also, a testing library has been developed for the evaluation of the component.

A scientific paper about the tool and the described approach has been published at the IEEE International Conference on Cloud Computing 2021 (CLOUD) [27]. A further publication about extensions of the *CloudPG* for privacy analysis – the Privacy Property Graph – has been published at the 23rd Privacy Enhancing Technologies Symposium [25].

4.1.4 Limitations and future work

The approach implemented in the *Cloud Property Graph* has some general limitations. First, it is constrained by the available source code and accessible APIs, i.e., when libraries are used whose source code is not available, or source code is not available for other reasons, it will not be part of the resulting graph and cannot be analysed for security problems. Regarding the APIs, the limitation is the same as for the evidence collection with *Clouditor*: only the information the cloud APIs offer can be analysed. Second, the approach currently generates additional manual effort since the tool has to be set up, it has to be manually applied, and its results need to be manually evaluated. However, its application and result analysis have potential for automation which should be addressed in future work. Also, regarding integration with the MEDINA framework, i.e., with the *Security Assessment* or *Orchestrator* should be addressed in future work, as the novel approach that the *Cloud Property Graph*'s implements, generates results that are not yet compatible with the MEDINA data model.

4.2 LLVM Extensions of the Code Property Graph

While the *CloudPG* component requires source code to be available to analyse it for compliance with security requirements, in a real-world cloud system, the source code is not always available. Consider the following use case: A CSP operates a cloud system including different types of computing resources which include a number of Function-as-a-Service resources ("serverless functions"). These functions make use of external libraries which are not open-source, but are only available as binaries. In this case, we require a new method that allows to translate a binary to the code property graph representation which we can analyse, subsequently similarly to the *CloudPG*.

A number of extensions of the code property graph library enable such an analysis of low-level code representations of applications. In particular, the extensions are able to parse LLVM-IR, an intermediate representation which is used by various compilers and binary lifters, and transfer this representation to the code property graph. The translation step is designed in a way to be fully compatible with the existing cpg representation which is already supported by the tools *CloudPG* (see section 4.1) and *Codyze* (see section 4.3). This allows full reusability of the existing concepts to analyse such code.

The current implementation supports nearly all combinations which exist in LLVM-IR retrieved during compilation and from lifters and improves the scalability of the approach by an order of magnitude. In addition, existing analyses can be reused with minimal changes.

4.2.1 Implementation

The following subsections provides functional and technical descriptions of the *LLVM extensions* to the *Code Property Graph*.

4.2.1.1 Functional description

This approach extends the cpg library which is used by *Clouditor* and *Codyze* as a unified code representation with the ability to parse LLVM-IR. The extension consists of a cpg language frontend which uses the LLVM API through JNI to parse LLVM-IR files and translates each statement and expression to its equivalent in the cpg representation.

This translation is implemented in a way to reuse only the existing types of nodes and edges in the *Code Property Graph*. Furthermore, the translation tries to map LLVM-IR's specific data types and operations to the ones used by high-level programming languages to reuse the existing analyses without the need to adapt them. However, since the cpg's data model does not account for all statements which exist in LLVM-IR, several specifics of the programming language are modeled to minimize the loss of information while still keeping the semantics of the program.

Furthermore, a cpg pass optimizes the result of the translation by removing auxiliary nodes which are required during the translation to collect relevant information and yet parse the files sequentially. In addition, the pass refines the information which is contained in the nodes.

Contrary to prior approaches, the translation scales well even in the presence of large LLVM-IR files.

4.2.1.1.1 Fitting into overall MEDINA Architecture

The LLVM extensions of the cpg are integrated into the *Code Property Graph* library which is used by the *CloudPG* as described in section 4.1 and by *Codyze* as described in section 4.3. Therefore, the LLVM extensions are also available to both tools and can be used in the exact same way as the other components of the code property graph library. The translations are designed in a way to be fully compatible with the other existing programming languages.

4.2.1.1.2 Related requirements

The relevant requirements from D5.2 [3] are listed below and a brief description of how they are implemented is given.

Note that these requirements are not mandatory for this tool since it is a novel research concept, whose integration into the framework needs to be implemented in future work.

Requirement id	TEGT.S.05
Short title	Verify security requirements
Description	The developed tool must be able to verify security requirements and raise warnings/errors with respect to secure coding practices and secure information and data flows.
Implementation state	Partially implemented

The cpg library features a built-in analysis and query API which allows the user to assess security risks and verify security requirements based on custom queries. This API is also usable for the respective extensions. However, developing analyses and queries is not part of MEDINA.

Requirement id	TEGT.S.07
Short title	Support for common programming languages, libraries, cloud services
Description	The developed tool should support common programming languages, libraries and cloud services.
Implementation state	Partially implemented

Numerous programming languages can be translated to LLVM-IR by their compile toolchains as part of their compilation process. Similarly, libraries and closed-source binary files can be transferred to LLVM-IR. This enables support for analysis of almost all major programming languages and closed-source libraries, among others.

Requirement id	TEGT.S.08
Short title	Provision of malware, intrusion, and vulnerability detection tools
Description	Tools for malware detection, intrusion detection, and vulnerability scanning must be provided to assist CSPs with satisfying related requirements of security standards or to verify the compliance with such requirements.
Implementation state	Partially implemented

The LLVM Extensions of the cpg can be used to represent all kinds of programs even if the source code is unavailable for the analysis (by applying binary lifters). It is therefore capable of providing a representation for vulnerability and malware search of various programs. However, the analysis techniques are not developed within the scope of MEDINA.

Similar to the *Cloud Property Graph*, note that the common requirements (TEGT.C.X) are currently not relevant for the LLVM extensions, since it is still a novel research concept, whose integration into the framework needs to be designed and implemented in future work.

4.2.1.2 Technical description

The following subsections describe the technical details of the LLVM extensions to the cpg.

4.2.1.2.1 Prototype Architecture

The LLVM extensions of the cpg are available as source code, and a jar file and can be used as a software dependency. It is executed on source code and does not implement any server components nor agents.

4.2.1.2.2 Description of components

The LLVM extensions of the cpg consist of a cpg language frontend parsing the LLVM-IR instructions as well as a specific pass to improve the translation. It is developed as a single subproject of the cpg library.

4.2.1.2.3 Technical specifications

The LLVM extensions of the cpg is written in Kotlin. As described above, it is integrated into the cpg library to build a basic code property graph and thus uses some of its core functionalities. It makes use of LLVM's public C API which is offered through JNI by the JavaCPP Presets for LLVM project[39].

The user manual of the LLVM extension of the cpg can be found online as part of the open-source software.

4.2.2 Delivery and usage

The following sections give a short overview of the delivery and usage of the LLVM Extensions fo the *CloudPG*.

4.2.2.1 Package information

The artifact is available as a jar file.

4.2.2.2 Licensing information

The code is part of the code property graph library that is licensed under an Apache 2 license.

4.2.2.3 Download

The project is available as open-source software on GitHub[40] as the subproject 'cpg-language-llvm'.

4.2.3 Advancements within MEDINA

A scientific paper about this approach has been published at the Information Security Conference [28].

4.2.4 Limitations and future work

The translation of LLVM-IR to the cpg representation features all statements but require minor development efforts to support all possible combinations of internal expressions. To identify such combinations, further testing is required.

[39] https://github.com/bytedeco/javacpp-presets/tree/master/llvm

[40] https://github.com/Fraunhofer-AISEC/cpg

Since the LLVM representation of some concepts differs between the programming languages, other passes may be necessary to improve the coverage of such concepts for the programming languages which are relevant for a certain target.

In addition, future work should identify suitable lifters to support binary analysis and whether they can be applied to typical artifacts which exist in cloud deployments of software.

Last, future work should develop further queries and analysis capabilities on the cpg to assess vulnerabilities or potentially malicious code in the artifacts.

4.3 Codyze

Codyze is an open-source static application security testing tool. Its main goal is to verify if application source code complies to security requirements. Security requirements are derived from security requirement catalogues such as ENISA EUCS [1]. Security requirements are broken down into checkable source code properties. Afterwards, *Codyze* verifies specified source code properties and thereby can provide evidence and assessment results if a requirement is sufficiently realized in software.

Codyze supports security by design. It can recognize potential security flaws violating compliance to security standards like ENISA EUCS. It provides early feedback during the development process and can ensure that less security flaws remain in a production-ready deployed cloud service. It can also act as a quality gate within an CI/CD pipeline and prevent that cloud services are deployed in production which don't meet defined compliance requirements. It supplements the MEDINA framework by ensuring that consumable cloud applications and services are implemented securely.

4.3.1 Implementation

The following subsections provides functional and technical descriptions of *Codyze*.

4.3.1.1 Functional description

Codyze uses the MARK DSL [29] to specify checkable software properties. MARK can model entities and define rules that must hold for the usage of an entity. *Codyze* evaluates MARK rules against provided source code and attest if a rule is adhered to or not. Based on the evaluation result from MARK rules, software properties required to fulfil security requirements are validated.

Currently, *Codyze* analyses source code written in C/C++ and Java. MARK rules cover cryptographic libraries Bouncy Castle for Java and Botan for C++ as well as transport encryption based on Java's secure socket extension (JSSE). Thus, source code can be checked if cryptographic operations and transport encryption are properly implemented and thereby attest state-of-the-art cryptography of sufficient strength.

In addition, *Codyze* inspects the source code repository of a software project. It assesses whether good development practices are followed during development. In particular, it checks which developers committed source code to the repository and whether these additions are signed. This information is checked against a list of authorized developers and signatures. The corresponding assessment result attests that only authorized accounts and personnel made changes in accordance with requirements like EUCS CCM-05.

As *Codyze* analyses source code and repositories, it is not integrated into the cloud platform itself. It is a tool used by CSPs to validate the source code of applications and services prior to deployment and general availability in the cloud. Therefore, *Codyze* must be integrated into the CSP's development, continuous integration, and continuous deployment pipeline. Once integrated, *Codyze* can check submitted code while it is being developed. Configured as a breaking check point in a CI/CD pipeline, it can prevent the roll out of software not meeting security requirements.

Codyze submits results from its analysis to the *Orchestrator*. It uses the MEDINA data model to send evidence. Afterwards, *Codyze* posts its assessment results referencing the previously submitted evidence.

In addition, *Codyze* creates a report in the SARIF format [30]. SARIF is an OASIS standard that specifies a format to encode information and findings from static code analysis into an exchangeable format. This report is saved during a CI/CD pipeline and can be reviewed afterwards to identify problems and fix them. Thereby, code repository platforms with integrated CI/CD functionality such as GitHub[41] can automatically process SARIF reports and represent the respective information integrated on their platform. Developers can use this information to fix problems in their source code.

4.3.1.1.1 Fitting into overall MEDINA Architecture

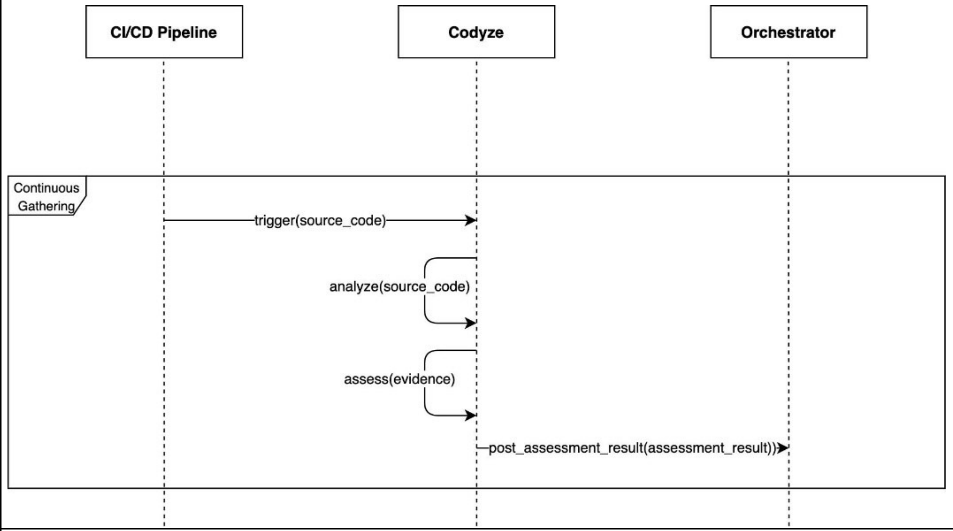
Codyze integrates itself into the overall MEDINA architecture as an application-level evidence collection and security assessment tool (see Figure 1 in section 2). *Codyze* assesses source code of cloud application and ensures compliance to security requirements catalogues like ENISA EUCS within applications. It submits assessment results to the *Orchestrator* for further processing. In addition, it stores the evidence used to derive an assessment result with the *Orchestrator* to verify provenance.

4.3.1.1.2 Component card

Table 14. Component card for Codyze

<table>
<tr><th>Component Name</th><th><em>Codyze</em></th></tr>
<tr><td><strong>Main functionalities</strong></td><td>The component provides the following functionalities:<ul><li>Static code analysis</li><li>Validation of program specifications</li><li>Validation of good development processes</li></ul></td></tr>
<tr><td><strong>Sub-components Description</strong></td><td>MARK is a domain specific language to specify verifiable properties that source code must adhere to. It can for example restrict possible data values and their flow, or specify interactions between objects.
The CPG library is used for the internal source code representation. It represents source code as a multigraph based on the concept of code property graphs.
The <em>Codyze</em> library provides the implementation for the analysis engine. It initiates the parsing of source code with the CPG library and performs the interpretation of MARK rules. It performs the necessary analysis steps to extract properties from source code that are checked against the requirements specified in MARK rules. The analysis results are collected for further processing by <em>Codyze</em> for MEDINA.</td></tr>
<tr><td><strong>Main logical Interfaces</strong></td><td><table>
<tr><th>Interface name</th><th>Description</th><th>Interface technology</th></tr>
<tr><td>CLI</td><td><em>Codyze</em> provides a command line interface. It can be used to call <em>Codyze</em> to analyse a set of files and produce results. It is suitable for example for a CI/CD pipeline.</td><td>stdin/stdout</td></tr>
<tr><td>MARK</td><td>MARK depends on Eclipse Xtext and reuses the UI elements of</td><td>UI of Eclipse</td></tr>
</table></td></tr>
</table>

[41] https://docs.github.com/en/code-security/code-scanning/integrating-with-code-scanning

<table>
<tr><td></td><td><table><tr><td></td><td>Eclipse and Xtext. Writing MARK requires an Eclipse IDE.</td><td></td></tr></table></td></tr>
<tr><td><b>Requirements Mapping</b></td><td>List of requirements covered by this component (see D5.2 [3]): TEGT.C.01-02, TEGT.S.02-04, 06-07, TEGT.08.</td></tr>
<tr><td><b>Interaction with other components</b></td><td><table><tr><th>Interfacing Component</th><th>Interface Description</th></tr><tr><td><i>Orchestrator</i></td><td>Send assessment results</td></tr></table></td></tr>
<tr><td><b>Relevant sequence diagram/s</b></td><td> </td></tr>
<tr><td><b>Current TRL [42]</b></td><td>Based on existing open-source <i>Codyze</i> library: TRL 4.</td></tr>
<tr><td><b>Target TRL [43]</b></td><td>Based on existing open-source <i>Codyze</i> library: TRL 5.</td></tr>
<tr><td><b>License</b></td><td>Apache 2.0</td></tr>
<tr><td><b>WP and task</b></td><td>WP3, Task 3.3</td></tr>
<tr><td><b>MEDINA Workflows</b></td><td>WF2 “Preparation of MEDINA Components”, and
WF5 “EUCS Compliance Assessment” (see D5.4 [5])</td></tr>
</table>

4.3.1.1.3 Related requirements

The relevant requirements from D5.2 [3] are listed below and a brief description of how they are implemented is given.

Requirement id	**TEGT.C.01**
Short title	Continuous collection
Description	The developed tools must be able to collect evidence continuously, i.e., in (high)-frequency intervals.
Implementation state	Fully implemented

[42] TRL value before validation

[43] TRL value after validation

Codyze is integrated into the CI/CD pipeline at CSPs. It is executed based on the frequency of committed code changes.

Requirement id	TEGT.C.02
Short title	Provision to defined interfaces
Description	The developed tools must provide collected evidence to a security assessment tool via its offered APIs.
Implementation state	Fully implemented

Codyze submits all evidence to the *Orchestrator*. As *Codyze* provides its own assessment results, evidence is submitted for reference in the resulting assessment result.

Requirement id	TEGT.S.03
Short title	Implement information and data flow analysis
Description	The developed tool must be able to perform information and data flow analysis on a cloud application.
Implementation state	Fully implemented

Codyze can perform information and source code analysis; the extended analysis for contextual information of cloud workloads has been addressed in the *Cloud Property Graph* tool which is closely related to *Codyze*. For example, it can analyse infrastructure configurations and CI/CD information of respective configuration files to check where a certain piece of code is deployed in a cloud service.

Requirement id	TEGT.S.04
Short title	Support expression of security requirements
Description	The developed tool must be able to support the expression of security requirements to be checked on application code. Requirements come for example from WP2.
Implementation state	Fully implemented

While *Codyze* can verify security requirements, defined in the MARK DSL, it is not yet able to verify MEDINA-related requirements, e.g., written in Rego. Instead, requirements are mapped to MARK rules such that validation of MARK rules indicates compliance to requirements.

Requirement id	TEGT.S.05
Short title	Verify security requirements
Description	The developed tool must be able to verify security requirements and raise warnings/errors with respect to secure coding practices and secure information and data flows.
Implementation state	Fully implemented

Codyze is currently able to generate warnings for identified non-compliances. It remains to integrate these warnings in MEDINA, e.g., in a user interface. The current rule set needs to be extended.

Requirement id	TEGT.S.06
Short title	Retrieve source code of cloud applications

Description	The developed tool should be able to retrieve (semi-)automatically the source code of cloud applications requiring analysis.
Implementation state	Fully implemented

Source code is provided as part of the CI/CD pipeline. The fulfilment of this requirement will be validated during field tries with partners.

Requirement id	**TEGT.S.07**
Short title	Support for common programming languages, libraries, cloud services
Description	The developed tool should support common programming languages, libraries and cloud services.
Implementation state	Fully implemented

4.3.1.2 Technical description

The following subsections describe the technical details of *Codyze*.

4.3.1.2.1 Prototype architecture

Codyze consists of an executable binary distribution and runs stand-alone. It is also available as a container image. *Codyze* is executed on the source code of cloud application and services. Therefore, there are no server components or agents. Figure 10 depicts *Codyze* architecture.

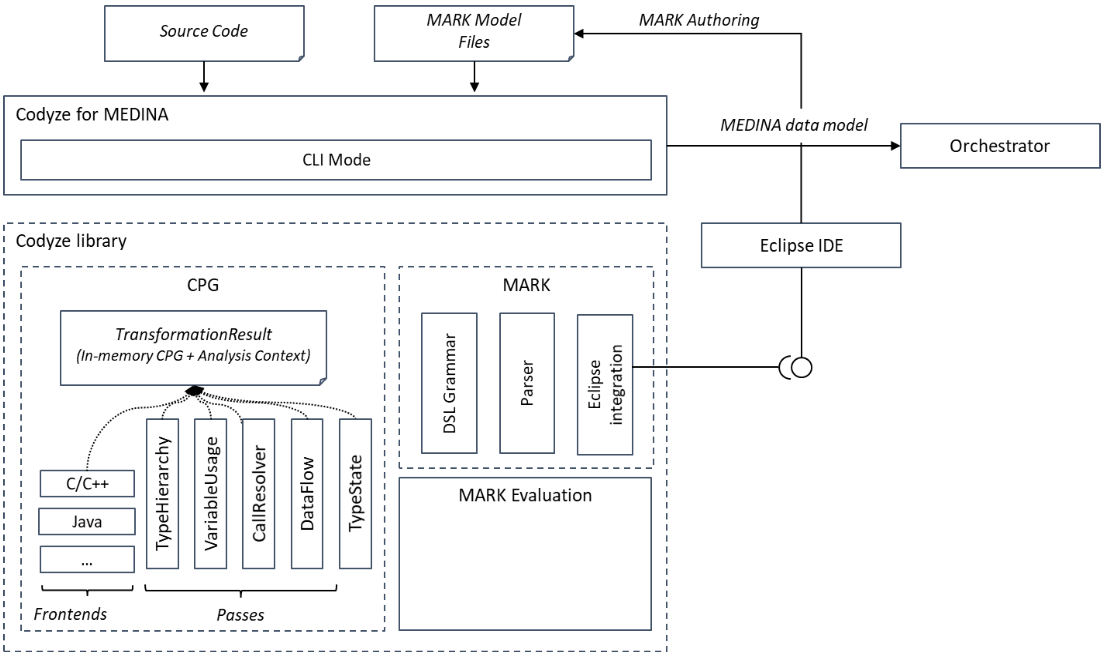


Figure 10. Codyze architecture

Codyze for MEDINA provides a command line interface. This is the main interface to run *Codyze* automatically in a CI/CD pipeline. In this mode, *Codyze* generates a report that contains problematic source code locations where security requirements are not met. In addition, this mode will return an error code when security requirements are not met and can thus terminate CI/CD pipelines. This behaviour ensures that *Codyze* prevents the roll out of cloud applications and services that do not comply to security requirements as required by catalogues like ENISA EUCS.

Internally, *Codyze* uses the cpg library. This library implements a code property graph. This graph is a multigraph representing source code structures in a graph representation. On this graph model of the source code, *Codyze* can perform the source code evaluation.

The evaluation is specified in MARK. MARK files are provided to *Codyze* either as part of *Codyze* or as a path to MARK files on the command line. These MARK files are parsed by *Codyze* and define the necessary evaluation steps to validate the compliance to security requirements.

The results of the evaluation are provided to developers as SARIF reports. In addition, findings are submitted to the *Orchestrator* of the MEDINA framework in the specified data format.

4.3.1.2.2 Description of components

Codyze for MEDINA makes use of the following components:

MARK

MARK is a domain specific language to specify verifiable properties that source code must adhere to. It can, for example, restrict possible data values and their flow, or specify interactions between objects. A corresponding software library built on top of Xtext[44] provides the language grammar and parser functionality. In addition, a generated Eclipse plugin provides MARK specific editing support in Eclipse IDE.

CPG

CPG is a library implementing a code representation based on the concept of a code property graph [31]. It is responsible for parsing source code and providing a graph-based code representation suitable for querying code properties.

Codyze library

Codyze library provides the analysis engine for *Codyze*. It uses the CPG to parse source code. In addition, it uses the MARK library to parse MARK files. The *Codyze* library implements the analysis steps to interpret MARK rules and identify rule violations in source code. Assessed rules generate a finding that either certifies compliance or documents a rule violation.

4.3.1.2.3 Technical specifications

Codyze is developed in Java and Kotlin. It uses the libraries CPG and MARK as dependencies. In addition, *Codyze* ships with MARK rules that check compliance to strong, state-of-the-art cryptography.

The integration of *Codyze* at the CSPs requires a platform for CI/CD. *Codyze* can be integrated into CI/CD pipelines either by using the binary distribution or the container image as a step during validation step of the pipeline.

Codyze should be configured as a static application security test. It should prevent successful CI/CD pipeline completion if violations are discovered. It thus prevents the deployment of artefacts into production that do not meet minimum compliance requirements.

In addition, *Codyze* generates a report in SARIF format. SARIF is a standardized description of findings from static analysis tools. Its adoption is rising and many SAST tools now include support

[44] https://www.eclipse.org/Xtext/

for SARIF. This is also reflected in platforms for source code management and CI/CD integration. Platforms like GitHub support the evaluation of SARIF reports and include results within the UI of their platform. Developers can identify problematic results on these platforms for example as code annotation in merge requests.

Please note that the user manual and installation instructions of *Codyze* can be found in *Appendix C: Codyze - Installation instructions and User* manual.

4.3.2 Delivery and usage

The following sections give a short overview of the delivery and usage of *Codyze*. Please note that additional instructions can be found in *Appendix C: Codyze - Installation instructions and User* manual.

4.3.2.1 Package information

Codyze is packaged as a binary distribution in a ZIP archive. In addition, the MEDINA public GitLab repository[45] contains a Dockerfile to build a container image from source.

4.3.2.2 Licensing information

Codyze for MEDINA and its components are licensed under the open-source Apache License 2.0.

4.3.2.3 Download

Codyze for MEDINA is available from the public MEDINA GitLab repository[45].

The source code for the *Codyze* library is available from its GitHub repository[46] and the MARK source code is available from its GitHub repository[47].

4.3.3 Advancements within MEDINA

Codyze has been successfully adopted for MEDINA. The original *Codyze* library has been extended to support the analysis of source code commonly seen in cloud service implementations. The analysis engine within *Codyze* has been improved.

Secondly, *Codyze* has been integrated into the MEDINA architecture. It supports the MEDINA data model. Findings from *Codyze* are submitted as evidence and assessment results with the *Orchestrator*. Thus, results from the analysis of *Codyze* can be reviewed within the MEDINA framework.

Thirdly, new MARK specifications have been written that model common cloud service functionality. These models have been used to define MARK rules that support compliance to EUCS requirements. The mapping between rules and requirements has been implemented. Currently, the recently developed MARK specifications cover transport encryption using TLS in support of EUCS control CKM-02 “Encryption of Data in motion” and cryptographic operations for secure data storage in support of EUCS control CKM-03 “Encryption of Data at Rest”.

Finally, *Codyze* has been extended to inspect the corresponding source code repositories for the analysed source code. Thereby, *Codyze* can validate whether good development practices are

[45] https://git.code.tecnalia.com/medina/public/codyze
[46] https://github.com/Fraunhofer-AISEC/codyze
[47] https://github.com/Fraunhofer-AISEC/codyze-mark-eclipse-plugin

followed. In particular, this extension to *Codyze* confirms that only authorized accounts and personnel made changes to the source code of cloud services in support of the EUCS control CCM-05 “Performing and Logging Changes”.

4.3.4 Limitations and future work

Codyze analyses source code and its usefulness is therefore limited by the inputs it gets: Since there is no reliable source for knowing which code exists and should be deployed, it is also not possible to verify within *Codyze* if all relevant code has been analysed. Therefore, we assume that *Codyze* is applied to all relevant code.

Another limitation of *Codyze* is the use of MARK as a specification language. Source code properties need to be modelled and rules need to be specified. The resulting MARK specification is to some extend specific for a programming language and modelled software library. Hence, *Codyze* can analyse and assess only source code for which specifications exist. Moreover, specifications need to be updated to keep up to date with changing requirements and updated software libraries.

5 Assessment of Organisational Measures

Assessment of organisational measures is handled by the MEDINA component named *AMOE* (*Assessment and Management of Organisational Evidence*), which is designed to extract evidence from policy documents. Furthermore, it allows to set and submit assessment results to the MEDINA framework.

5.1 Implementation

The following subsections provides functional and technical descriptions of *AMOE*.

5.1.1 Functional description

AMOE is a proof-of-concept prototype for assessment and management of organisational evidence. It is designed to extract evidence based on organisational metrics, targeted to specific parts of policy documents. After extraction, the evidence can be inspected in the GUI of the tool. Users can then decide on the compliance status of a metric. Once the user is satisfied with the assessment, the result and evidence can be forwarded to the *Orchestrator*. While extracting the evidence is fully automated, the final decision is made by the user.

To improve *AMOE*, different evidence extraction methods have been researched, the main one is depicted in Figure 11. After uploading a policy document, via GUI or API, it goes through the various stages of the extraction pipeline. There are two stages, the first is the pre-processing, and the second the actual evidence extraction.

Pre-processing

The PDF document is transformed into a HTML with poppler utils'[48] pdftohtml. While it is processed, common errors for section headings are fixed. The result is a structured document that eases the filtering process. In the last stage of the pre-processing, information such as table of contents, or parts of the header or footer are removed. The whole process depends heavily on the quality of the PDF, which is reflected in the results.

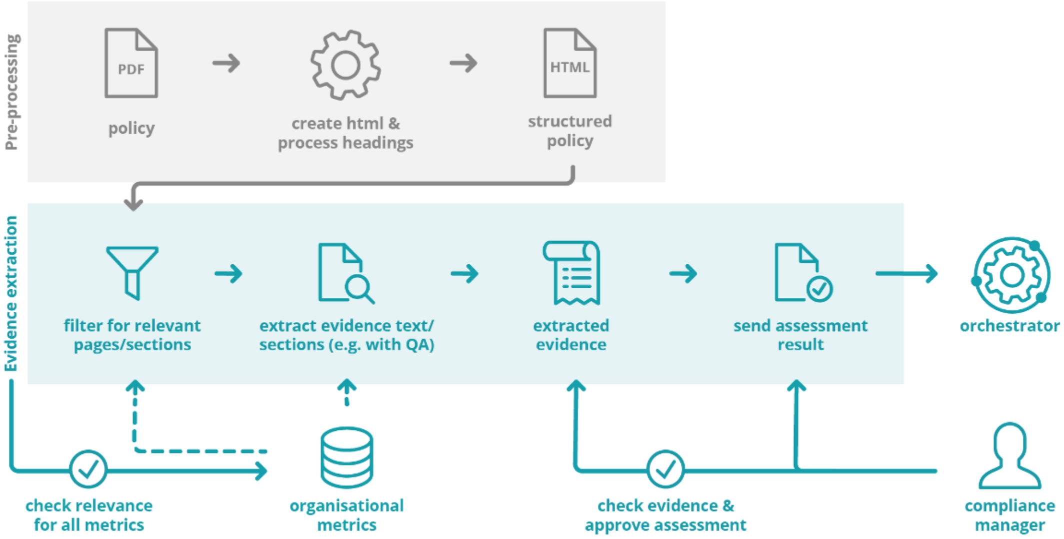


Figure 11. Main architecture for AMOE (keyword-based extraction method)

[48] Library for pdf transformation/rendering: https://poppler.freedesktop.org

Evidence extraction

After the preparation of the document, a subprocess is started for every organisational metric. First, the search space is reduced by filtering the document for relevant sections/paragraphs. This is done using the keywords defined in the metric and the section headings marked in the pre-processing. Second, the metric question as well as the filtered text is fed into the pre-trained question answering model. Third, extracted answer/evidence is marked in the HTML document and the evidence is stored.

In the case a target value is defined for an organisational metric, *AMOE* derives an assessment hint. This hint is using the extracted evidence and compares it with the defined target value. As some metrics are phrased as an open question, not all have a target value defined. The main goal of *AMOE* is to aid the user, not perform blind assessment. Therefore, the results need to be verified by a human.

At this stage, the data is available for a user (i.e., compliance manager) to inspect the result via the GUI or API and set the final assessment result. Once set, the assessment result can be sent to the Orchestrator.

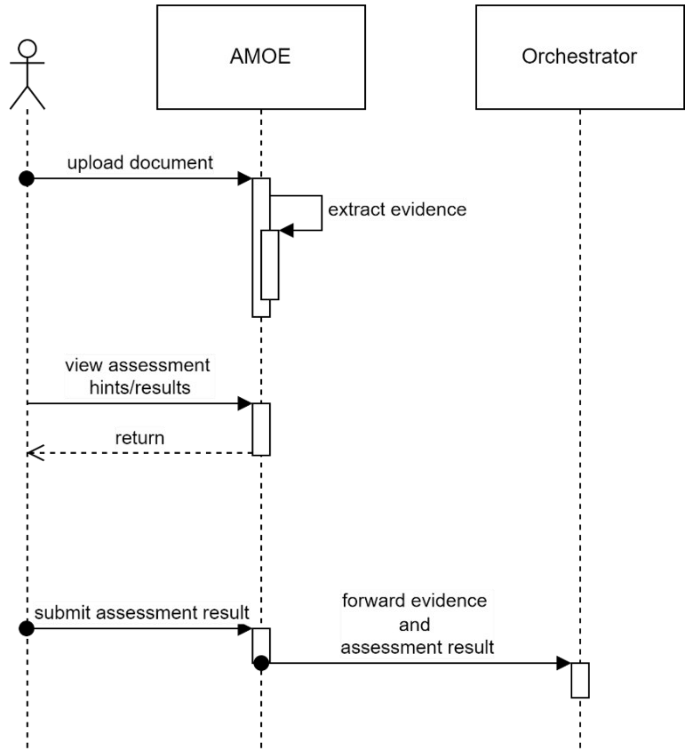
5.1.1.1 Fitting into overall MEDINA Architecture

AMOE provides the functionality to add assessment results of organisational requirements/metrics to the MEDINA framework. It works with organisational metrics from the *Catalogue of Controls and Metrics* and accesses the predefined target values from the *Orchestrator* API (metric configuration). Alternatively, the metrics can be read from a local file. Once an uploaded file is processed and the evidence is processed and confirmed by a user, it can be forwarded to the *Orchestrator* and further according to the evidence pipeline defined.

5.1.1.2 Component card

Table 15. Component card for the Assessment and Management of Organisational Evidence

<table>
<tr><td><b>Component Name</b></td><td><b><i>Assessment and Management of Organisational Evidence (AMOE)</i></b></td></tr>
<tr><td><b>Main functionalities</b></td><td>The component provides the following functionalities:
• Gathering and processing organizational evidences
• Providing evidences to the <i>Clouditor</i> for assessment</td></tr>
<tr><td><b>Sub-components Description</b></td><td>Organizational evidence is collected by applying NLP and organisational metrics to an uploaded document. The processing part transforms this evidence in the form of technical evidence. This transformed evidence then is provided to the <i>Security Assessment</i> of <i>Clouditor</i> which can handle such technical evidence.</td></tr>
<tr><td><b>Main logical Interfaces</b></td><td><table>
<tr><th>Interface name</th><th>Description</th><th>Interface technology</th></tr>
<tr><td>UI</td><td>GUI to upload documents,
Retrieve evidence,
Set assessment results, and
Submit/forward assessment results</td><td>webservice</td></tr>
</table></td></tr>
</table>

<table>
<tr><td></td><td><table><tr><td>API</td><td>Upload documents,
Retrieve evidence,
Set assessment results, and
Submit/forward assessment results</td><td>REST</td></tr></table></td></tr>
<tr><td><b>Requirements Mapping</b></td><td>List of requirements covered by this component (see D5.2 [3]): OEGM.01, OEGM.02, OEGM.03, OEGM.04, OEGM.05.</td></tr>
<tr><td><b>Interaction with other components</b></td><td><table><tr><th>Interfacing Component</th><th>Interface Description</th></tr><tr><td><i>Orchestrator</i></td><td>Send collected evidences + assessment results
Retrieve metric configurations</td></tr><tr><td><i>Catalogue of Controls and Metrics</i></td><td>Retrieve metrics and requirements as needed</td></tr></table></td></tr>
<tr><td><b>Relevant sequence diagram/s</b></td><td>
</td></tr>
<tr><td><b>Current TRL [49]</b></td><td>TRL 3</td></tr>
<tr><td><b>Target TRL [50]</b></td><td>TRL 4</td></tr>
<tr><td><b>Programming language</b></td><td>Python</td></tr>
<tr><td><b>License</b></td><td>Apache 2.0</td></tr>
<tr><td><b>WP and task</b></td><td>WP3: T3.4</td></tr>
<tr><td><b>MEDINA Workflows</b></td><td>WF2 “Preparation of MEDINA Components”,
WF3 “EUCS deployment on ToC”, and</td></tr>
</table>

[49] TRL value before validation

[50] TRL value after validation

	WF5 "EUCS Compliance Assessment" (see D5.4 [5])

5.1.1.3 Related requirements

The relevant requirements from Deliverable D5.2 [3] are listed below with a brief description of how they are implemented.

Requirement id	OEGM.01
Short title	Continuous collection of organizational evidence
Description	The developed tool using NLP must be able to collect organizational evidence.
Status	Fully implemented

Evidence is automatically extracted after a file has been uploaded.

Requirement id	OEGM.02
Short title	Provision to defined interfaces
Description	The developed tool using NLP must provide collected evidence to the central evidence collection component (T3.1) via its offered APIs.
Status	Fully implemented

The users of the component can forward an assessment result (and evidence as needed by the API) to the *Orchestrator*. This can be triggered by the UI or API once a compliance status has been set for a metric or extracted evidence.

Requirement id	OEGM.03
Short title	Usability for auditors
Description	The evidence management component should provide easy-to-use functionalities for auditors to search through relevant evidence. The assessment is handled manually though the UI. The assessment can be adjusted via API (should be checked/verified by a human beforehand).
Status	Fully implemented

AMOE provides the extracted results in an interactive UI. Evidence is highlighted in the extracted answer as well as in the processed document. Furthermore, it extracts the page number where the evidence was found so one can double check the information in the original uploaded document. The computed assessment hints are designed to help users in reaching their decisions. As they might be false, the final decision on an assessment is done by the user.

Requirement id	OEGM.04
Short title	Minimum evidence storage
Description	The evidence management component must be able to store and provide evidence at least back to the last assessment (if needed).
Status	Fully implemented

The uploaded and extracted data is stored until it is manually deleted. It can be deleted via the GUI. Log information can be deleted by an administrator with access to the database.

Requirement id	OEGM.05
Short title	Evidence Assessment results
Description	The assessment results of evidence assessments must be submitted to the evidence Orchestrator via the API it provides.
Status	Fully implemented

The assessment results can be forwarded to the Orchestrator using the API or GUI. See also Req. OEGM.02

5.1.2 Technical description

The following subsections describe the technical details of *AMOE*.

5.1.2.1 Prototype Architecture

Figure 12 depicts the *AMOE* architecture. The prototype core is the webservice based on the Quart Python library[51]. The API and GUI are served by this central component. For the interaction with the rest of the MEDINA framework, there is a dedicated subcomponent (MEDINA component API, see Figure 12) incorporating the auto-generated Python clients to the APIs based on their OpenAPI specifications.

The session management of the webservice uses a separately deployed Redis instance. The evidence and log information are stored in the separately deployed MongoDB instance using the dedicated functions.

Not directly part of the prototype, but core part of the evidence extraction research is the quality check functions and separately deployed Inception[52] instance. Inception can be used to annotate the data needed for the quality measurements.

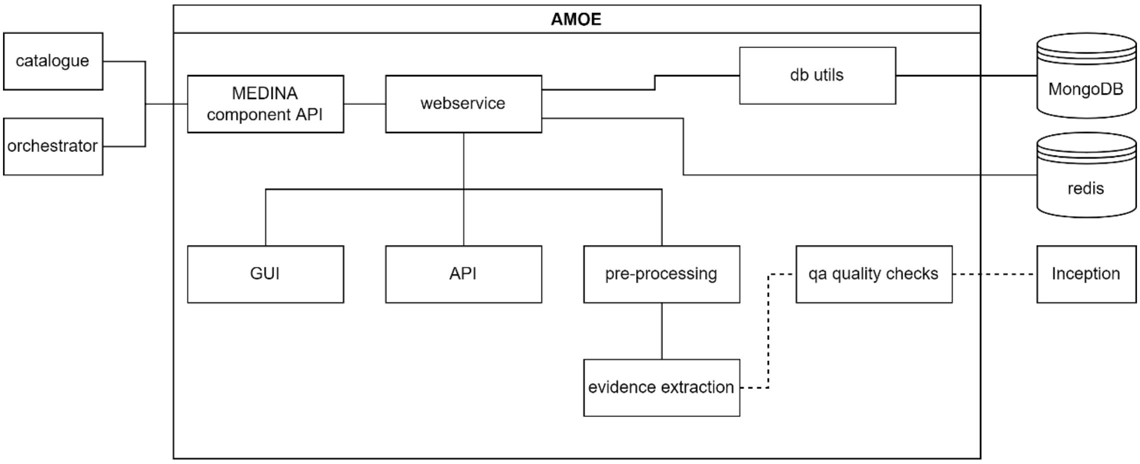


Figure 12. AMOE prototype architecture

5.1.2.2 Description of components

AMOE consists of a main webservice serving the GUI and the API. There are some parts that could be used independently of the main webservice. Figure 12 shows the main components of *AMOE*:

Webservice

This is the core component redirecting the data flow to the relevant subcomponents. It serves the GUI and API, and verifies authentication via the Keycloak[53] instance from MEDINA.

[51] https://pypi.org/project/quart/
[52] Annotation tool for NLP: https://inception-project.github.io/
[53] https://www.keycloak.org/

MEDINA component API

This subcomponent is used to access the different components from the MEDINA framework. It is used to retrieve requirements and metrics from the *Catalogue of Controls and Metrics* and the metric configuration from the *Orchestrator*. Furthermore, it is used for submitting the assessment results and extracted evidence.

DB utils

This subcomponent is used to store and access evidence results as well as local assessment results. It is also used to log relevant information such as by whom and when a document has been uploaded or an assessment result has been changed.

GUI

The graphical user interface serves to upload documents as well as to access the processed evidence. It enables the user to search, filter and manage the organisational evidence.

API

The API enables data access for other applications such as the *Company Compliance Dashboard*. It can be used to perform the most essential functions of the GUI. These include uploading a document, retrieving the processed evidence, setting assessment results, and submitting the assessment results to the *Orchestrator*.

Pre-processing

This subcomponent is triggered in a background process once a document has been uploaded. It performs the necessary transformations to enable the evidence extraction.

Evidence extraction

This subcomponent is triggered after the pre-processing pipeline is done. It works as described in section 5.1.1.

5.1.2.3 Technical specifications

The *AMOE* tool is written in Python 3.x. It uses various Python libraries as well as the pdftohtml functionality from poppler utils[54]. The webservice is built on Quart[55], the evidence extraction is based on transformers[56], PyTorch[57] and the roberta-base-squad2[58] model from huggingface.

The component is using MongoDB[59] and Redis[60] to store the data. Evidence and logs are stored in the MongoDB. Redis is used in par with the quart-session library.

[54] https://poppler.freedesktop.org/
[55] https://pypi.org/project/quart/
[56] https://github.com/huggingface/transformers
[57] https://pytorch.org/
[58] https://huggingface.co/deepset/roberta-base-squad2
[59] https://www.mongodb.com/
[60] https://redis.io/

5.2 Delivery and Usage

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

Please note that the User Manual can be found in *Appendix E: AMOE User Manual.*

5.2.1 Package

AMOE can be deployed as a Docker container. Table 16 shows an overview of the repository folders and files.

Table 16. Overview of AMOE's source code package contents

Folder	Description
clouditor_evidence_client_legacy/	Contains the generated Python client for the evidence API of *Clouditor* based on their openapi file.
clouditor_orchestrator_client_legacy/	Contains the generated Python client for the Orchestrator API of *Clouditor* based on their openapi file.
inception_kubernetes/	Contains the kubernetes files to deploy an instance of the annotation tool inception.
jenkins/	Contains the Jenkins pipeline code.
kubernetes/	Contains the kubernetes files for the deployment of *AMOE*.
metric_data/	Contains the local version of the metrics.
paragraph_extraction/	Contains the code for the pre-processing pipeline.
qa/	Contains the code for evidence extraction using the question answering model as well as code to compute quality scores.
static/	Contains the stylesheets and images for the webservice.
templates/	Contains the HTML templates for the webservice.
utils/	Contains code for utility functions of the webservice such as use of other MEDINA component's API, evidence extraction and database management.
/	The root folder contains the main webservice and configuration as well as the Dockerfile.

5.2.2 Installation

Clone the *AMOE* repository. Set up a MongoDB and a Redis instance (see kubernetes files in the repository).

Set the following fields in the config.py:

```
MONGODB_URL

KEYCLOAK_URL + authentication settings

CATALOGUE_API_URL

ORCHESTRATOR_API_URL
```

Run `hypercorn app:app -b 0.0.0.0:8000` to deploy the service locally, or deploy with kubernetes.

5.2.3 User Manual

AMOE user manual can be found in *Appendix E: AMOE User Manual.*

5.2.4 Licensing

The component is planned to be licenced under Apache 2.0.

5.2.5 Download

The component code can be downloaded from the MEDINA public git repository[61].

5.3 Advancements within MEDINA

The *AMOE* component has been developed from scratch for MEDINA by Fabasoft to cover evidence extraction for organisational requirements in the light of Task 3.4 [62]. The advancements can be grouped into two parts – data management and the application itself.

Organisational metrics and annotation

Similar to the technical metrics, organisational metrics have been developed in cooperation with the domain experts of Bosch. For a subset of the organisational metrics, the relevant evidence has been annotated using the tool Inception.

Pre-processing pipeline

To enable evidence extraction on PDF documents, we developed a pre-processing stage, where the document is transformed into a HTML file. The information is reduced to what is deemed useful (e.g., removal of header and footer of the page; skip table of contents). Furthermore, only text that is deemed relevant is selected.

Evidence extraction

AMOE enables evidence extraction based on the organisational metrics. This works with standard NLP techniques and the use of a pre-trained question answering model.

UI development

To provide the information and functionality to a user, the webservice has been developed.

API development

To provide the information and functionality to a different client (e.g., the *Company Compliance Dashboard* (CCD), see D6.3 [32], Appendix D – Workflows of Use Case 2), the API has been developed.

Quality checks

To get an impression of how well the prototype is working and to be able to develop / research for better extraction methods, quality checks have been implemented.

[61] https://git.code.tecnalia.com/medina/public/amoe

[62] Development started late in the project as Fabasoft adopted this task from another partner. *AMOE* is foreseen as a Proof of Concept (PoC) in MEDINA.

Currently, there are four basic approaches implemented for evidence extraction. The first one (keyword based) is described in the functional description in section 5.1.1 (Evidence extraction) above. The other approaches are quite similar; however, the order of the results differ. As they are still under development, the description is kept to a higher level in this paragraph. The main difference with the keyword-based approach is that the same metric question is evaluated in every paragraph instead of once per filtered document text. The results are ordered in various ways, e.g., based on a similarity score (cosine similarity) between each paragraph and the keywords. The top result is selected to be extracted as evidence. The keyword-based approach performs best, therefore it is used in the deployment of the prototype.

Test cases have been constructed to determine the quality of the approach. Each test case is using a policy document and a set of organisational metrics for which the evidence has been annotated on the document. The annotations were edited using the Inception tool (see also section 5.1.2.1). The test case policy document is used for empirical and numerical analysis of the evidence extraction. In the rest of the section, details and results of the quality checks (score) and the two test cases are described.

Fabasoft test case

Fabasoft created a specific document containing dummy policies for test purposes, as the internal could not be shared with the consortium. These policies are contained in a single document further referenced as “Fabasoft dummy policy document”.

Bosch test case

Bosch has shared two policy documents, one of which contains more details and is therefore used for the experiments. The document used is further referenced as “Bosch IoT policy document”.

Results

Current results for the Fabasoft dummy policy document (Fabasoft test case):

- Keyword based approach: `19 / 28 = 0.68`
- Score based approach: `13 / 28 = 0.46`
- Similarity based approach: `7 / 28 = 0.25`
- Similarity + score based approach: `13 / 28 = 0.46`

Although, the main focus in research was on the metrics and policies by Fabasoft, here are the current results for the Bosch IoT policy document (Bosch test case):

- Keyword based approach: `10 / 50 = 0.20`
- Score based approach: `6 / 50 = 0.12`
- Similarity based approach: `8 / 50 = 0.16`
- Similarity + score based approach: `9 / 50 = 0.18`

28 organisational metrics have been annotated for the Fabasoft test case and 50 for the Bosch test case. The score is the ratio of the number of correctly retrieved text samples to the total number of the respectively annotated text passages ($score = \frac{\#\text{correctly retrieved evidence}}{\#\text{total annotated evidence}}$).

Some of the results can be explained by the difference of expertise between the annotators and the metric developer. However, we think the results can be improved by improving on the extraction pipeline, and performing curation on the annotated data and metrics. The metric questions and especially the linked keywords need to be revised.

5.4 Limitations and future work

The following sections describe current and future limitations of the *AMOE* proof-of-concept prototype.

Current limitations that can be overcome

In Task 3.4, the *AMOE* prototype is subject to limitations of the dataset (quantity and quality). Given that no suitable publicly available datasets were found, data needs to be constructed and adapted to the needs of the project. This is addressed by the creation of specifically designed organizational metrics designed on the basis of policies provided by Bosch and Fabasoft. For good results this requires time and expertise to map the relevant information in the policy text to concrete metrics that are specialized for extraction of this organisational information. These provided metrics and documents are the main input for the prototype and are used for building the evidence extraction pipeline as well as its evaluation measures. The organisational metrics can be revised and extended to increase the available dataset. Thus, we are able to get a broader picture on how well the prototype works in practice as well as whether some improvements to the evidence extraction pipeline are fruitful the way they are intended.

The quality of the current extraction results varies from sample to sample. However, with further research and improvement of the evidence extraction methods, this can be overcome up to a certain point. Confidence in the results of the extraction pipeline is measured by calculating how much of the annotated data (ground truth) can be correctly extracted. This score can be used to validate added improvements to the extraction pipeline.

Generic limitations that will not be overcome (due to technology, specific requirements, etc.)

As the data for this task is rather limited, it was decided to use a pre-trained model. This model has been trained for question answering in a different domain, so it is not specific to the terminology of cloud service provider's policies. Empiric analysis has shown that the model provides reasonable answers for the given task, however, we suspect that the results could be improved with the roberta-base-squad2 model, given enough data. As we do not want to overfit on the little data acquired, we refrain from performing this step, to retain the generic output.

As the development and test environment by the MEDINA project do only include limited resources (no GPU, limited RAM and CPU), the processing of the policy documents takes quite some time. This could be partly overcome (in a future setup, not MEDINA) by using e.g., GPU for the question answering (it has been verified in a local test environment). However, even with the use of GPUs this is not an instant process.

The *AMOE* component deals at the moment only with policy data in PDF. This is because it is hard to obtain enough generic evidence deemed organisational, without developing a solution that is too specific for a single cloud service provider. To gain a tool with high quality results for multiple providers, we focus on this task.

The evidence extraction process is limited to the organisational metrics. The used questions should be defined to retrieve a concrete answer / value to measure. No binary answers are possible with the chosen model settings and the model only selects a single answer. This could be extended in future work, for the moment it was deemed to be not in the scope as complexity increases and focus is on concrete evidence extraction.

There will be no fully automatic assessment; due to no guarantee that the assessment would be correct, thus the prototype is designed to aid the CAB. Results might be biased up to a certain point due to construction of the examples from Bosch and Fabasoft. Shortcomings in this regard could be mitigated by using same metric on different policy documents, as then either the metric or the extraction method needs to be adapted.

For this proof-of-concept component, no log files or screenshots of evidence will be covered, the focus is on policy documents. Given the structure of the dataset and the focus on building a prototype that works; the organisational metrics are specific to policy documents and no other “raw” evidence sources. Future work could extend the prototype to work with other document types as well as forms of evidence. However as most of other document types used to write policies can be converted to PDF anyway this is not considered for now.

6 Conclusions

This document presents a final technical report about the design, architecture, and implementation states of MEDINA evidence gathering components. It details the individual components' functions, internal structure, technical description and the description of their subcomponents. Information about their limitations and future planned work is also presented.

The components presented in this document include three tools supporting the security assessment of cloud infrastructure (*Clouditor, Wazuh*, and *Vulnerability Assessment Tools (VAT)*, along with corresponding *Wazuh* and *VAT Evidence Collector*), a pair of tools for assessing the security and compliance of cloud application's source code (*Codyze* and *CloudPG*, along with its LLVM Extensions), and a component for the assessment of organisational measures based on analysis of CSP's documentation (*Assessment and Management of Organisational Evidence, AMOE*).

At the end of their development, the described components satisfy all of their functional requirements (for additional info see *Appendix A: MEDINA Requirements Implementation Overview*) elicited in scope of WP5 and presented in D5.2 [3]. The components are also fully integrated with other parts of the MEDINA framework.

7 References

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

[2] MEDINA Consortium, “D3.3 Tools and techniques for the management of trustworthy evidence - v3,” 2023.

[3] MEDINA Consortium, “D5.2 MEDINA requirements, Detailed architecture, DevOps infrastructure and CI/CD and verification strategy - v2,” 2022.

[4] MEDINA Consortium, “D3.5 Tools and techniques for collecting evidence of tehcnical and organisational measures - v2,” 2022.

[5] MEDINA Consortium, “D5.4: MEDINA integrated solution-v2,” 2023.

[6] MEDINA Consortium, “D2.5 Specification of the Cloud Security Certification Language - v3,” 2023.

[7] MEDINA Consortium, “D2.2 Continuously certifiable technical and organizational measures and catalogue of cloud security metrics-v2,” 2023.

[8] MEDINA Consortium, “D4.3 Tools and techniques for the management and evaluation of cloud security certifications-v3,” 2023.

[9] Wazuh Inc., “Wazuh,” [Online]. Available: https://wazuh.com/. [Accessed April 2023].

[10] Cisco, “ClamAV,” [Online]. Available: https://www.clamav.net/. [Accessed April 2023].

[11] Chronicle Security, “VirusTotal,” [Online]. Available: https://www.virustotal.com/. [Accessed April 2023].

[12] “w3af,” [Online]. Available: http://w3af.org/. [Accessed April 2023].

[13] OWASP Foundation, “OWASP Zed Attack Proxy (ZAP),” [Online]. Available: https://owasp.org/www-project-zap/. [Accessed April 2023].

[14] “Nmap,” [Online]. Available: https://nmap.org/. [Accessed April 2023].

[15] Docker, Inc., “Docker,” [Online]. Available: https://www.docker.com/. [Accessed April 2023].

[16] Rapid7, “Metasploit,” [Online]. Available: https://www.metasploit.com/. [Accessed April 2023].

[17] VMware, Inc., “RabbitMQ,” [Online]. Available: https://www.rabbitmq.com/. [Accessed April 2023].

[18] MongoDB, Inc., “MongoDB,” [Online]. Available: https://www.mongodb.com/. [Accessed April 2023].

[19] OpenStack, “OpenStack Swift (Github repository),” [Online]. Available: https://github.com/openstack/swift. [Accessed April 2023].

[20] Google LLC, “Angular,” [Online]. Available: https://angular.io/. [Accessed April 2023].

[21] HashiCorp, Inc., “Vagrant,” [Online]. Available: https://www.vagrantup.com/. [Accessed April 2023].

[22] Red Hat, Inc., “Ansible,” [Online]. Available: https://www.ansible.com/. [Accessed April 2023].

[23] Faraday Security, “Faraday (Github repository),” [Online]. Available: https://github.com/infobyte/faraday. [Accessed April 2023].

[24] CYBERWISER.eu consortium, “CYBERWISER.eu,” [Online]. Available: https://www.cyberwiser.eu/. [Accessed April 2023].

[25] I. Kunz, K. Weiss, A. Schneider and C. Banse, “Privacy Property Graph: Towards Automated Privacy Threat Modeling via Static Graph-based Analysis,” in *Proceedings on Privacy Enhancing Technologies*, Lausanne, 2023.

[26] M. Deng, K. Wuyts, R. Scandariato, B. Preneel and W. Joosen, “A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements,” *Requirements Engineering,* vol. 16, pp. 3-32, 2011.

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

[28] A. Küchler and C. Banse, “Representing LLVM-IR in a Code Property Graph,” in *Information Security (ISC)*, 2022.

[29] Fraunhofer AISEC, “MARK (Modeling Language for Cryptography Requirements and Guidelines) GitHub page,” [Online]. Available: https://github.com/Fraunhofer-AISEC/codyze-mark-eclipse-plugin. [Accessed April 2023].

[30] OASIS SARIF TC, “Static Analysis Results Interchange Format (SARIF) Version 2.1.0,” 27 March 2020. [Online]. Available: https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html. [Accessed April 2023].

[31] F. Yamaguchi, N. Golde, D. Arp and K. Rieck, “Modeling and Discovering Vulnerabilities with Code Property Graphs,” in *2014 IEEE Symposium on Security and Privacy*.

[32] MEDINA Consortium, “D6.3 Use cases development and validation-prototypes-v1,” 2022.

8 Appendix A: MEDINA Requirements Implementation Overview

Table 17 below presents an overview of requirements and their fulfilment with the currently implemented tools presented in this document. The requirements were elicited in WP5 and are detailed in D5.2 [3]. For the common requirements, implementation status is given for all the related components, while tool-specific requirements are presented in groups according to their respective components, such as they are also structured in D5.2. The implementation status has three possible values represented in the table by colours:

- Green: fully implemented
- Orange: partially implemented
- Red: not implemented

This table is updated and presented in all versions of this report for easier comparison and progress tracking. Note that some requirements were added, moved between categories, and/or their titles changed since the initial version of this deliverable.

Table 17. Overview of requirements satisfaction according to current implementation of the tools presented in this deliverable

Requirement ID	Short title	Implementation status			
Common requirements for technical evidence gathering		*Clouditor*	*Wazuh*	*VAT*	*Codyze*
TEGT.C.01	Continuous collection	🟩	🟩	🟩	🟩
TEGT.C.02	Provision to defined interfaces	🟩	🟩	🟩	🟩
Clouditor (Gathering evidence from cloud interfaces)					
TEGT.S.01	Collect evidence from cloud interfaces				🟩
EAT.02	Continuous evidence assessment				🟩
Clouditor (Security assessment)					
EAT.01	Evidence assessment target				🟩
EAT.03	Evidence assessment results				🟩
Clouditor (Evidence orchestration)					
ECO.01	Provision of Interfaces				🟩
ECO.02	Conformity to selected assurance level				🟥
ECO.03	Secure Transmission to evidence storage				🟩
ECO.04	Transmission of evidence checksums				🟩
Clouditor (Gathering evidence from CSP-Native Services)					
TEGT.S.09	Collect evidence from CSP-native services				🟩
EAT.04	Assess CSP-Native Evidence				🟩
Wazuh (Gathering evidence from computing resources)					
TEGT.S.08	Provision of malware, intrusion, and vulnerability detection tools				🟩
VAT (Gathering evidence from computing resources)					
TEGT.S.08	Provision of malware, intrusion, and vulnerability detection tools				🟩
CloudPG (Gathering evidence from application source code)					
TEGT.S.02	Collect evidence from source code via CPG				🟩

Requirement ID	Short title	Implementation status
TEGT.S.03	Implement information and data flow analysis	🟩
TEGT.S.10	Connect infrastructure- and application-level security analyses	🟩
TEGT.S.07	Support for common programming languages, libraries, cloud services	🟩
Codyze (Gathering evidence from application source code)		
TEGT.S.03	Implement information and data flow analysis	🟩
TEGT.S.04	Support expression of security requirements	🟩
TEGT.S.05	Verify security requirements	🟩
TEGT.S.06	Retrieve source code of cloud applications	🟩
TEGT.S.07	Support for common programming languages, libraries, cloud services	🟩
TEGT.S.08	Provision of malware, intrusion, and vulnerability detection tools	🟥
Assessment and Management of Organisational Evidence (AMOE)		
OEGM.01	Continuous collection of organizational evidence	🟩
OEGM.02	Provision to defined interfaces	🟩
OEGM.03	Usability for auditors	🟩
OEGM.04	Minimum evidence storage	🟩
OEGM.05	Evidence Assessment results	🟩

Table 18 presents the basic statistic of requirement coverage by each component. In total, there are 35 requirements (if the common requirements are counted separately – once for each component) related to the presented components. 33 (94%) of them are currently marked as fully implemented, 0 (0%) as partly implemented, and 2 (6%) as not implemented.

Table 18. Requirements satisfied by each tool

Tool	Number of requirements	Fully implemented	Partially implemented	Not implemented
Clouditor	**12**	11	0	1
Wazuh	**3**	3	0	0
VAT	**3**	3	0	0
CloudPG	**4**	4	0	0
Codyze	**8**	7	0	1
AMOE	**5**	5	0	0

However, if we take into consideration that one of the requirements (TEGT.S.08), which is not covered by *Codyze*, is actually covered by *VAT* (see section 3.3.1.1.3) and that the requirement ECO.02 is out of scope (since MEDINA focuses on the assurance level *high* only) (see section 3.1.1.1.3), we can conclude that **all the relevant 33 (100%) requirements are covered**.

In addition, the LLVM Extensions of the *Code Property Graph*, as a new tool, partially cover three requirements: TEGT.S.05, TEGT.S.07 and TEGT.S.08. Since this tool is a novel research concept and its integration into the MEDINA framework is future work beyond the scope of the project, the number of requirements they cover are not counted towards the number of all requirements covered, as presented in Table 18. Furthermore, these three partially covered requirements are fully covered by other tools (*Wazuh*, *VAT* and *Codyze*).

9 Appendix B: Clouditor - Readme, Installation instructions and User manual

9.1 README

Clouditor Community Edition

Introduction

Clouditor is a tool which supports continuous cloud assurance. Its main goal is to continuously evaluate if a cloud-based application (built using, e.g., Amazon Web Services (AWS) or Microsoft Azure) is configured in a secure way and thus complies with security requirements defined by, e.g., Cloud Computing Compliance Controls Catalogue (C5) issued by the German Office for Information Security (BSI) or the Cloud Control Matrix (CCM) published by the Cloud Security Alliance (CSA).

Features

Clouditor currently supports over 60 checks for Amazon Web Services (AWS), Microsoft Azure and OpenStack. Results of these checks are evaluated against security requirements of the BSI C5 and CSA CCM.

Key features are:

- automated compliance rules for AWS and MS Azure,
- granular report of detected non-compliant configurations,
- quick and adaptive integration with existing service through automated service discovery,
- descriptive development of custom rules using Cloud Compliance Language (CCL) to support individual evaluation scenarios,
- integration of custom security requirements and mapping to rules.

Build

Install necessary protobuf tools.

```
go install google.golang.org/protobuf/cmd/protoc-gen-go \
google.golang.org/grpc/cmd/protoc-gen-go-grpc \
github.com/grpc-ecosystem/grpc-gateway/v2/protoc-gen-grpc-gateway \
github.com/google/gnostic/cmd/protoc-gen-openapi
```

Also make sure that `$HOME/go/bin` is on your `$PATH` and build:

```
go generate ./...
go build -o ./engine cmd/engine/engine.go
```

Usage

To test, start the engine with an in-memory DB

```
./engine --db-in-memory
```

Alternatively, be sure to start a Postgres DB:

```
docker run -e POSTGRES_HOST_AUTH_METHOD=trust -d -p 5432:5432 postgres
```

Clouditor CLI

The Go components contain a basic CLI command called `cl`. It can be installed using `go install cmd/cli/cl.go`. Make sure that your `~/go/bin` is within your $PATH. Afterwards the binary can be used to connect to a *Clouditor* instance.

```
cl login <host:grpcPort>
```

Command Completion

The CLI offers command completion for most shells using the `cl completion` command. Specific instructions to install the shell completions can be accessed using `cl completion --help`.

9.2 Installation instructions

The full up-to-date installation instructions can be found in the README at the *Clouditor* Github repository[11].

To build *Clouditor*, the Gradle build tool[63] is used. To enable an auto-discovery for AWS and/or Azure the credentials must be stored in the home folder.

Since *Protobuf* is used, the corresponding packages must also be installed (the installation command can be found in the README):

- *google.golang.org/protobuf/cmd/protoc-gen-go*
- *google.golang.org/grpc/cmd/protoc-gen-go-grpc*
- *github.com/grpc-ecosystem/grpc-gateway/v2/protoc-gen-grpc-gateway*
- *github.com/googleapis/gnostic/apps/protoc-gen-openapi*

The *Clouditor* features its own CLI for which *~/go/bin* must be within the *$PATH environment variable.*

To build the prototype make sure that *$HOME/go/bin* is within your *$PATH* and run the following commands:

- *go generate ./...*
- *go build ./…*

The engine can be started by using an in-memory DB as well as a Postgres DB. To start the engine with an in-memory DB, use *./engine –db-in-memory* if starting with a separate Postgres DB use *./engine*. Start the Postgres DB.

For development, an overview for the installation instructions is given in the following:

- Build *Clouditor* with Gradle or alternatively via a docker image
- Build Go components (*Protobuf* tools needed for compiling the *Protobuf* files)
- Start the *Clouditor* with in-memory DB or a Postgres DB
- Install and use the CLI for running the *Clouditor* at runtime

[63] https://github.com/clouditor/clouditor/blob/main/README.md

9.3 User Manual

The *Clouditor* components can be used with CLI commands. The help is shown by running *cl –help:*

```
user@user:~$ cl --help
The Clouditor CLI

Usage:
  cl [command]

Available Commands:
assessment-result        Assessment result commands
cloud                    Target cloud services commands
completion               Generate completion script
evidence                 Evidence commands
help                     Help about any command
login                    Log in to Clouditor
metric                   Metric commands
requirement              Requirement commands
resource                 Resource commands
service                  Service commands
tool                     Tool commands

Flags:
  -h, --help                         help for cl
  -s, --session-directory string     the directory where the session will be saved and
loaded from (default "/home/user/.clouditor/")

Use "cl [command] --help" for more information about a command.
```

Each command can have additional subcommands which are explained by the corresponding help, e.g., *cl assessment-result –help*.

Please note that before using the *Clouditor* CLI it is necessary to login to *Clouditor*: *cl login <host:grpcPort>.*

10 Appendix C: Codyze - Installation instructions and User manual

10.1 Installation instructions

The latest installation instructions for *Codyze for MEDINA* are described in the README available in the public MEDINA repository[64]. In addition, the latest installation instructions for each component are available in their respective GitHub repositories and on the main *Codyze* website[65].

Codyze for MEDINA is built with Gradle. The project repository contains the Gradle wrapper. To build *Codyze* for MEDINA two build steps are required. First, the REST API for the Orchestrator needs to be built. It is generated from the OpenAPI specifications distributed by the Orchestrator. The build commands are:

- `./gradlew[.bat] generateAll`
- `./gradlew[.bat] installDist`

After the build, the *Codyze for MEDINA* executable is located at `{project-dir}/build/install/codyze/`. In this directory one can find three directories:

- bin/ contains a shell Windows batch script to run *Codyze*
- lib/ contains all library files
- mark/ contains the MARK files included in *Codyze*

The start scripts in bin/ will print a command help when executed. The command help contains short descriptions of each command argument and parameter.

The binary distribution of *Codyze for MEDINA* as ZIP archive has the same structure. In fact, Gradle is used to build the binary distribution as archive by calling

- `./gradlew[.bat] assembleDist`

The resulting archive can be found in `{project-dir}/build/distributions/`.

The components used by *Codyze for MEDINA* are consumed as library dependency and automatically retrieved when *Codyze for MEDINA* is built from source. In case the components need to be built from source, their respective code repositories contain up to date information on the build instruction, prerequisites, and procedure.

Finally, the MARK plugin for Eclipse IDE is provided by an Eclipse update site. The installation of this plugin is described on the *Codyze* website.

10.2 User Manual

The user manual for *Codyze for MEDINA* is available as README in the public MEDINA repository[64]. In addition, the *Codyze* library and MARK are documented at the *Codyze* website[65] and at their respective GitHub repositories[66,67].

[64] https://git.code.tecnalia.com/medina/public/codyze
[65] https://www.codyze.io/
[66] https://github.com/Fraunhofer-AISEC/codyze
[67] https://github.com/Fraunhofer-AISEC/codyze-mark-eclipse-plugin

11 Appendix D: Cloud Property Graph - Installation instructions and User manual

11.1 Installation instructions

Note that the installation instructions may change with the advancement of the tool, so consider the installation details in the README file on the GitHub repository[68]. The following instructions and the following manual are partly copied from this file:

1. Clone the git repository `git@github.com:clouditor/cloud-property-graph.git`
2. Set the JAVA_HOME variable to Java 11
3. Install jep[69]
4. For usage of experimental language, e.g., go
    a. Checkout Fraunhofer AISEC - Code Property Graph and build by using the property -Pexperimental: `./gradlew build -Pexperimental`
    b. The libcpgo.so must be placed somewhere in the java.library.path[70].
        i. Under Linux in /lib/. `sudo cp ./cpg-library/src/main/golang/libcpgo.so /lib/`
        ii. And Mac in ~/Library/Java/Extensions.
5. To build, the graph classes need to be built from the Ontology definitions by calling `./build-ontology.sh`. Then build using `./gradlew installDist`.

11.2 User Manual

Start neo4j using `docker run -d --env NEO4J_AUTH=neo4j/password -p7474:7474 -p7687:7687 neo4j`, or `docker run -d --env NEO4J_AUTH=neo4j/password -p7474:7474 -p7687:7687 neo4j/neo4j-arm64-experimental:4.3.2-arm64` on ARM systems.

Run `cloudpg/build/install/cloudpg/bin/cloudpg`. This will print a help message with any additional needed parameters. The root path is required, and the program can be called as follows: `cloudpg/build/install/cloudpg/bin/cloudpg --root=/x/testprogramm folder1/ folder2/ folder 3/`

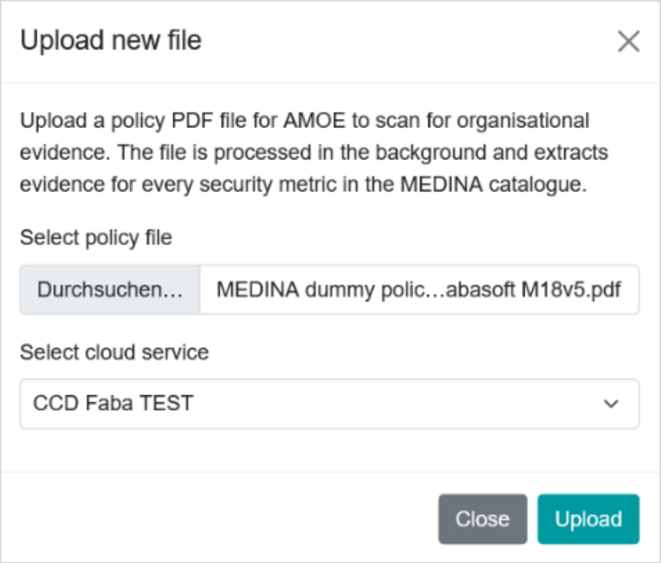
[68] https://github.com/clouditor/cloud-property-graph/blob/main/README.md

[69] Follow the instructions at https://github.com/Fraunhofer-AISEC/cpg#python

[70] For further information see: https://github.com/Fraunhofer-AISEC/cpg#usage-of-experimental-languages

12 Appendix E: AMOE User Manual

To use the *AMOE* GUI, start by clicking on the "Upload new file" button. A file upload dialog box will then appear, as shown in Figure 13.



Upload new file

Upload a policy PDF file for AMOE to scan for organisational evidence. The file is processed in the background and extracts evidence for every security metric in the MEDINA catalogue.

Select policy file

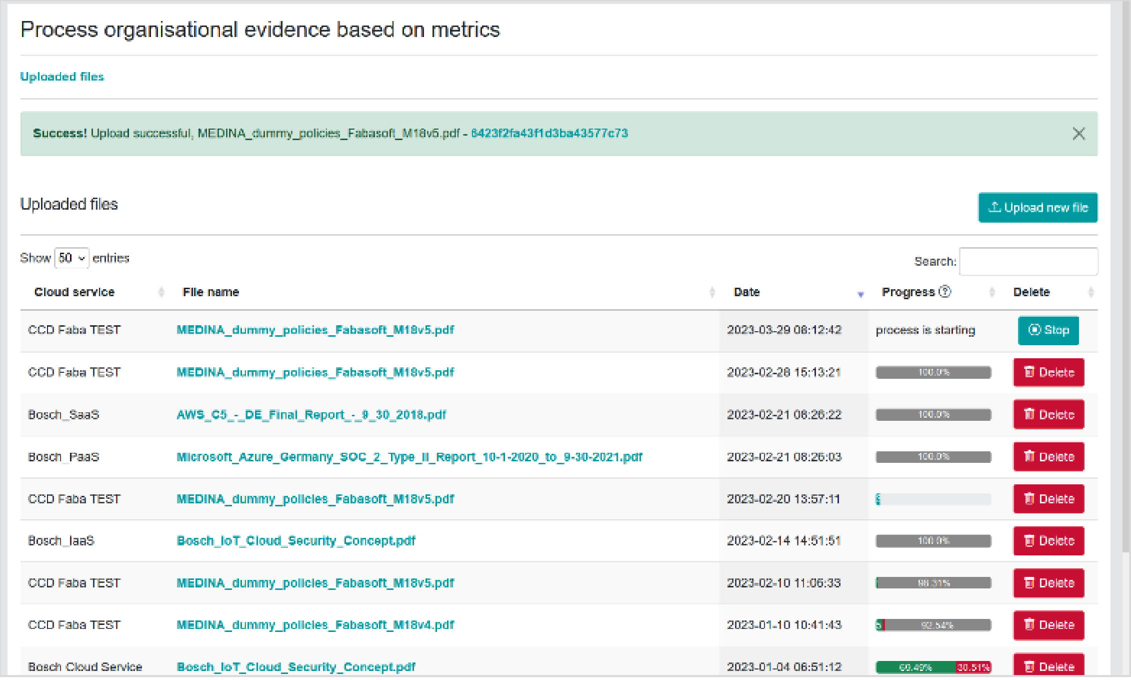
Durchsuchen… MEDINA dummy polic…abasoft M18v5.pdf

Select cloud service

CCD Faba TEST

Figure 13. AMOE file upload dialog

Select a policy PDF document to upload and the cloud service (id) that should be connected to. Then click on "Upload".



Process organisational evidence based on metrics

Uploaded files

Success! Upload successful, MEDINA_dummy_policies_Fabasoft_M18v5.pdf - 8423f2fa43f1d3ba43577c73

Uploaded files

Show 50 entries

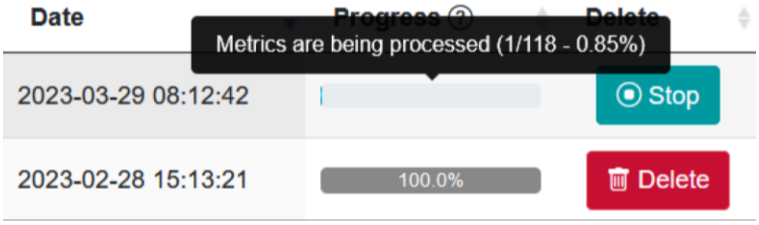
Search:

Cloud service	File name	Date	Progress ⓘ	Delete
CCD Faba TEST	MEDINA_dummy_policies_Fabasoft_M18v5.pdf	2023-03-29 08:12:42	process is starting	Stop
CCD Faba TEST	MEDINA_dummy_policies_Fabasoft_M18v5.pdf	2023-02-28 15:13:21	100.0%	Delete
Bosch_SaaS	AWS_C5_-_DE_Final_Report_-_9_30_2018.pdf	2023-02-21 08:26:22	100.0%	Delete
Bosch_PaaS	Microsoft_Azure_Germany_SOC_2_Type_II_Report_10-1-2020_to_9-30-2021.pdf	2023-02-21 08:26:03	100.0%	Delete
CCD Faba TEST	MEDINA_dummy_policies_Fabasoft_M18v5.pdf	2023-02-20 13:57:11		Delete
Bosch_IaaS	Bosch_IoT_Cloud_Security_Concept.pdf	2023-02-14 14:51:51	100.0%	Delete
CCD Faba TEST	MEDINA_dummy_policies_Fabasoft_M18v5.pdf	2023-02-10 11:06:33	98.31%	Delete
CCD Faba TEST	MEDINA_dummy_policies_Fabasoft_M18v4.pdf	2023-01-10 10:41:43	5 92.54%	Delete
Bosch Cloud Service	Bosch_IoT_Cloud_Security_Concept.pdf	2023-01-04 06:51:12	69.49% 30.51%	Delete

Figure 14. AMOE landing page after file upload

The evidence extraction process is started in the background. It can take some time until every organisational metric has been processed. The process can be stopped by clicking on the turquoise "stop" button. Files and their linked evidence can be deleted by clicking on the red "Delete" button.

Figure 15 depicts the progress of the background evidence extraction process. On hovering, the details are shown.

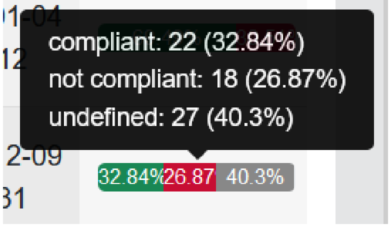


Metrics are being processed (1/118 - 0.85%)

Date	Progress	Delete
2023-03-29 08:12:42		Stop
2023-02-28 15:13:21	100.0%	Delete

Figure 15. AMOE evidence extraction progress

Figure 16 shows the status overview. This is displayed for every file after the evidence extraction process has finished. The details are shown by hovering with the mouse. Green indicates the number of assessment results set to compliant, red the number set to not compliant and grey marks where no status has been set (undefined).

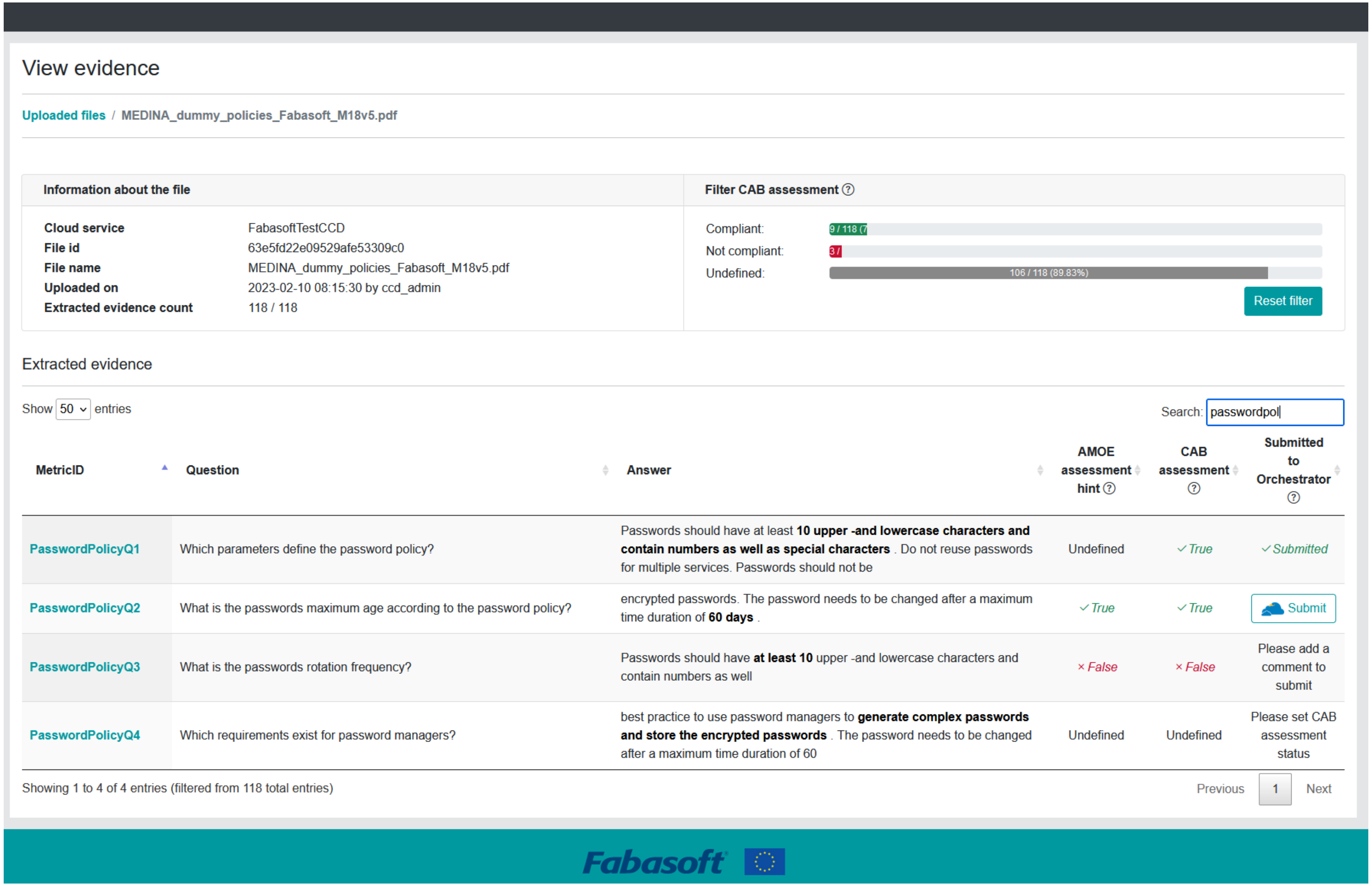


compliant: 22 (32.84%)
not compliant: 18 (26.87%)
undefined: 27 (40.3%)

Figure 16. AMOE assessment status overview per document

To view the evidence results of an uploaded file, click on a row of the respective table or the filename of the list, as depicted in Figure 14. The overview, as depicted in Figure 17, opens. This overview contains meta data of the uploaded file, filter, and search options. In case an assessment result has been set, it can be submitted to the *Orchestrator* directly from this view. Otherwise, click on a row to get to the detailed view for the extracted evidence.

Figure 18 depicts the detailed view of the extracted evidence. The linked requirement is shown on the top. This is followed by the metric meta data, extracted answer, assessment hint and options to set the assessment result and comment.



View evidence

Uploaded files / MEDINA_dummy_policies_Fabasoft_M18v5.pdf

Information about the file

Cloud service	FabasoftTestCCD
File id	63e5fd22e09529afe53309c0
File name	MEDINA_dummy_policies_Fabasoft_M18v5.pdf
Uploaded on	2023-02-10 08:15:30 by ccd_admin
Extracted evidence count	118 / 118

Filter CAB assessment ?

Compliant:	9 / 118 (7
Not compliant:	3 /
Undefined:	106 / 118 (89.83%)

Reset filter

Extracted evidence

Show 50 entries

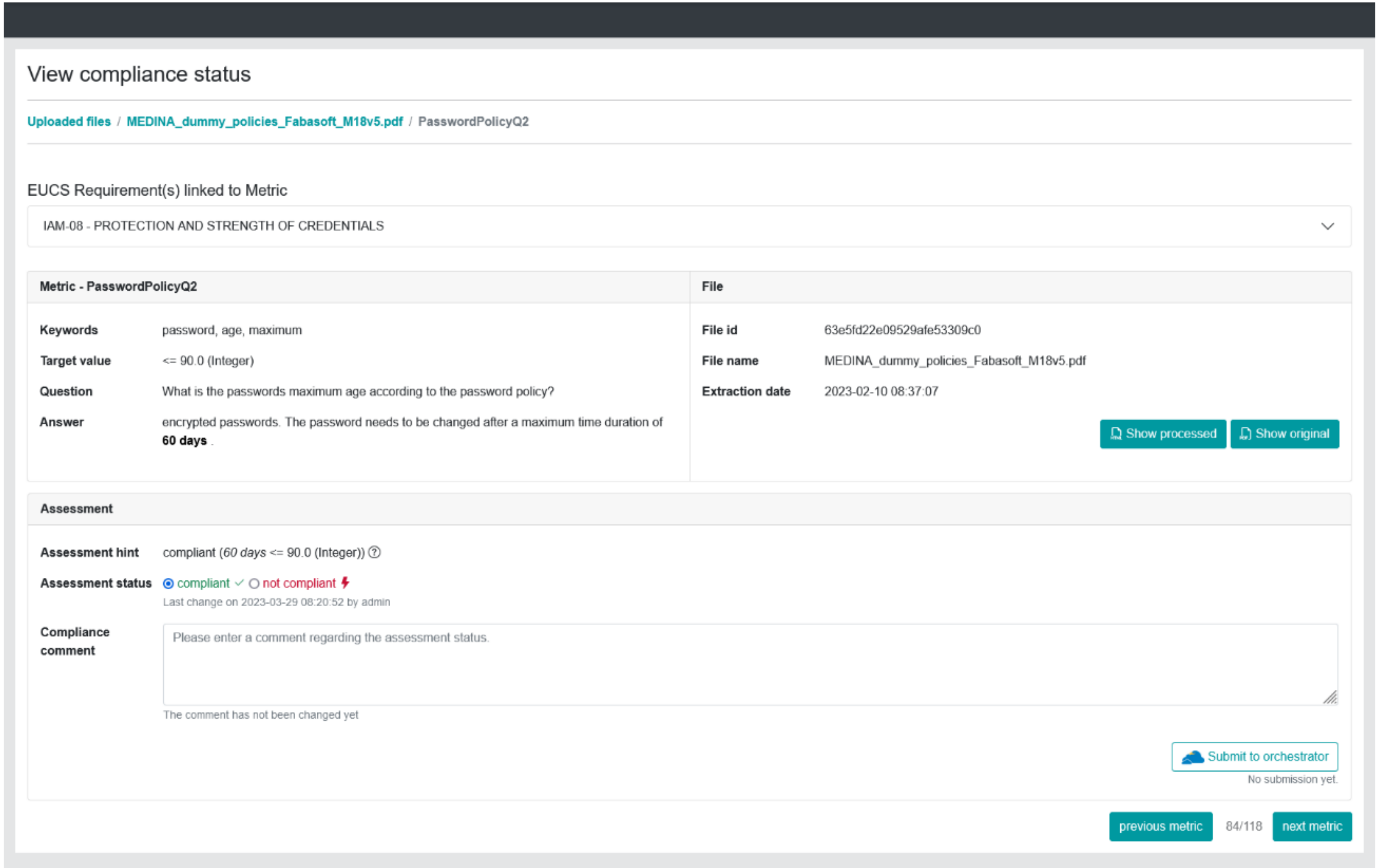
Search: passwordpol

MetricID	Question	Answer	AMOE assessment hint ?	CAB assessment ?	Submitted to Orchestrator ?
PasswordPolicyQ1	Which parameters define the password policy?	Passwords should have at least **10 upper -and lowercase characters and contain numbers as well as special characters** . Do not reuse passwords for multiple services. Passwords should not be	Undefined	✓ True	✓ Submitted
PasswordPolicyQ2	What is the passwords maximum age according to the password policy?	encrypted passwords. The password needs to be changed after a maximum time duration of **60 days** .	✓ True	✓ True	Submit
PasswordPolicyQ3	What is the passwords rotation frequency?	Passwords should have **at least 10** upper -and lowercase characters and contain numbers as well	× False	× False	Please add a comment to submit
PasswordPolicyQ4	Which requirements exist for password managers?	best practice to use password managers to **generate complex passwords and store the encrypted passwords** . The password needs to be changed after a maximum time duration of 60	Undefined	Undefined	Please set CAB assessment status

Showing 1 to 4 of 4 entries (filtered from 118 total entries)

Previous 1 Next

Figure 17. AMOE overview of extracted evidence and meta data linked to the uploaded file



View compliance status

Uploaded files / MEDINA_dummy_policies_Fabasoft_M18v5.pdf / PasswordPolicyQ2

EUCS Requirement(s) linked to Metric

IAM-08 - PROTECTION AND STRENGTH OF CREDENTIALS

Metric - PasswordPolicyQ2	
Keywords	password, age, maximum
Target value	<= 90.0 (Integer)
Question	What is the passwords maximum age according to the password policy?
Answer	encrypted passwords. The password needs to be changed after a maximum time duration of **60 days** .

File	
File id	63e5fd22e09529afe53309c0
File name	MEDINA_dummy_policies_Fabasoft_M18v5.pdf
Extraction date	2023-02-10 08:37:07

Assessment

Assessment hint compliant (*60 days* <= 90.0 (Integer)) ?

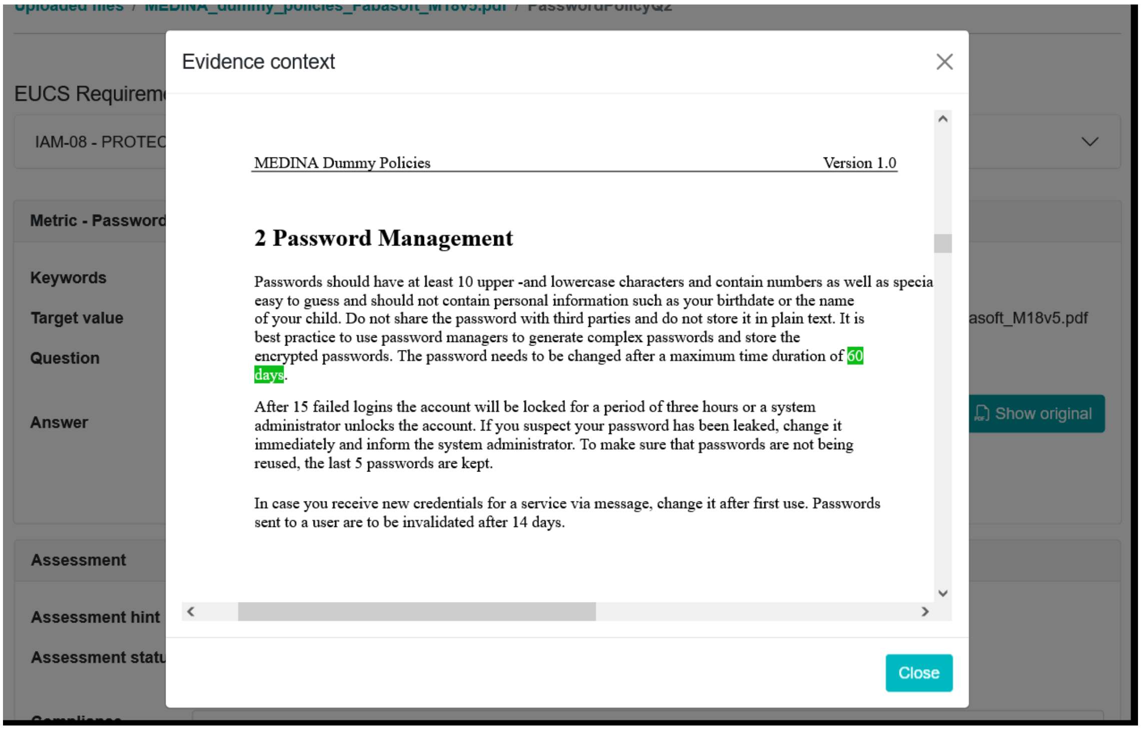
Assessment status ● compliant ✓ ○ not compliant
Last change on 2023-03-29 08:20:52 by admin

Compliance comment Please enter a comment regarding the assessment status.
The comment has not been changed yet

No submission yet.

Figure 18. AMOE view of organisational evidence

Figure 19 shows the processed HTML version of the document. The extracted evidence is highlighted in green.



MEDINA Dummy Policies Version 1.0

2 Password Management

Passwords should have at least 10 upper -and lowercase characters and contain numbers as well as specia easy to guess and should not contain personal information such as your birthdate or the name of your child. Do not share the password with third parties and do not store it in plain text. It is best practice to use password managers to generate complex passwords and store the encrypted passwords. The password needs to be changed after a maximum time duration of 60 days.

After 15 failed logins the account will be locked for a period of three hours or a system administrator unlocks the account. If you suspect your password has been leaked, change it immediately and inform the system administrator. To make sure that passwords are not being reused, the last 5 passwords are kept.

In case you receive new credentials for a service via message, change it after first use. Passwords sent to a user are to be invalidated after 14 days.

Figure 19. Show processed evidence (HTML view)

13 Appendix F: Wazuh and VAT Evidence Collector - Readme and installation instructions

13.1 Readme

The latest installation and configuration instructions can be found in the README in the public MEDINA repository[71].

Dependant components: Wazuh, ClamAV, VAT

Wazuh and VAT Evidence Collector generates evidence using information acquired from *Wazuh* and *Vulnerability Assessment Tool* APIs. These components should be installed and configurated in accordance with instructions given in the relevant repositories.

Wazuh Agents also require the *ClamAV* tool to be installed on their machines (to successfully cover all the requirements).

Required component versions:

- *Wazuh*: “v4.1.5”,
- *ClamAV*: “latest”,
- *VAT*: “latest.

See “wazuh-deploy”[72] for further details on how to set up *Wazuh* & *ClamAV*.[73]

See “vat-deploy”[74] for relevant information regarding *VAT* installation.

Wazuh evidence collector

Wazuh evidence collector uses *Wazuh*'s API[75] to access information about information and configurations of manager and system agents. As an additional measure to ensure correct configuration of ClamAV[76] (if installed on machine) we also make use of Elasticsearch's API [77]to directly access collected logs. Elastic stack is one of the *Wazuh*'s required components (usually installed on the same machine as the *Wazuh* server, but can be standalone as well).

VAT evidence collector

VAT evidence collector uses the *VAT* API to create w3af[78] & OWASP[79] scans and retrieve their results. These are later processed and forwarded to *Clouditor* (Assessment Interface).

[71] https://git.code.tecnalia.com/medina/public/wazuh-vat-evidence-collector
[72] https://git.code.tecnalia.com/medina/public/wazuh-deploy
[73] Note that the “wazuh-deploy” repository is deprecated and its information regarding Wazuh-VAT Evidence Collector configuration could be incomplete. However, information regarding Wazuh configuration is still up-to-date.
[74] https://git.code.tecnalia.com/medina/public/vat-deploy
[75] https://documentation.wazuh.com/current/user-manual/api/reference.html
[76] https://www.clamav.net/
[77] https://www.elastic.co/guide/en/elasticsearch/reference/current/search.html
[78] http://w3af.org/
[79] https://owasp.org/

13.2 Installation instructions & use

Using docker

1. Set up your *Wazuh* & *VAT* development environment. Use the "Wazuh Deploy" repository[80] to create and deploy Vagrant box with all the required components[81].

2. Clone this repository.

3. Build Docker image:

~~~
```shell
$ make build
```
~~~

4. Run the image:

~~~
```shell
$ make run
```
~~~

Note: See the "Environment variables" section for more information about configuration of this component and it's interaction with *Wazuh, Clouditor*, etc.

Local environment

1. Set up your *Wazuh* & *VAT* development environment. Use "Wazuh Deploy" repository[80] to create and deploy Vagrant box with all required components[81].

2. Clone this repository.

3. Install dependencies:

~~~
```shell
$ pip install -r requirements.txt
```
~~~

4. Set environment variables:

~~~
```shell
$ source .env
```
~~~

5. a) Install Redis server locally:

~~~
```shell
$ sudo apt-get install redis-server
```
~~~

Note: To stop Redis server use `/etc/init.d/redis-server stop`

b) Run Redis server in Docker container:

~~~
```shell
$ docker run --name my-redis-server -p 6379:6379 -d redis
```
~~~

In this case also comment-out server start command in 'entrypoint.sh':

~~~
```shell
~~~

[80] https://git.code.tecnalia.com/medina/public/wazuh-deploy

[81] Note: The Wazuh Deploy repository is not up to date! Use only for development

~~~~
#redis-server &
```
~~~~

6. Run ‘entrypoint.sh’:

~~~~
```shell
$ ./entrypoint.sh
```
~~~~

Note: This repository consists of multiple Python modules. When running Python code manually, the use of *-m* flag might be necessary.