Deliverable D3.4

Tools and techniques for collecting evidence of technical and organisational measures – v1

Editor(s):	Anže Žitnik
Responsible Partner:	XLAB
Status-Version:	Final – v1.1
Date:	30.09.2022
Distribution level (CO, PU):	PU

Project Number:	952633
Project Title:	MEDINA

Title of Deliverable:	D3.4 – Tools and techniques for collecting evidence of technical and organisational measures – v1
Due Date of Delivery to the EC	31.10.2021

Workpackage responsible for the Deliverable:	WP3 – Tools to gather evidences for high-assurance cybersecurity certification
Editor(s):	Anže Žitnik (XLAB)
Contributor(s):	Angelika Schneider, Immanuel Kunz, Florian Wendland (FhG), Franz Berger (Fabasoft), Aleš Černivec (XLAB)
Reviewer(s):	Artsiom Yautsiukhin (CNR), 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. There will be three iterations of the tool integration, an initial prototype, reflecting an early stage of integration in the technical framework (D3.4), the second release will be based on a refinement of the technical architecture (D3.5), finally the third iteration will reflect the implementation of the use cases (D3.6). 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
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	18.01.2021	First draft version of the ToC	Anže Žitnik (XLAB)
	24.08.2021	ToC updated	Anže Žitnik (XLAB)
	20.09.2021	Clouditor description added	Angelika Schneider (FhG)
	22.09.2021	Updates in Section 3	Immanuel Kunz, Angelika Schneider (FhG)
	22.09.2021	Updates in Section 3	Aleš Černivec, Anže Žitnik (XLAB)
	30.09.2021	Updates in Section 3	Angelika Schneider (FhG)
	05.10.2021	Updates in Section 3	Anže Žitnik (XLAB)
	07.10.2021	Added content in Section 5	Franz Berger (Fabasoft)
	07.10.2021	Added content in Section 4	Florian Wendland (FhG)
v0.2	08.10.2021	Minor content and comments added	Anže Žitnik (XLAB)
	11.10.2021	Updated Section 5	Franz Berger (Fabasoft)
	16.10.2021	Updates in Section 3	Anže Žitnik (XLAB)
v0.3	20.10.2021	Added contents to Sections 1 and 2. Formatting and other refinements throughout the document.	Anže Žitnik (XLAB)
v0.4	21.10.2021	Added Executive Summary and Conclusions.	Anže Žitnik (XLAB)
v0.5	2.11.2021	Addressing comments from the internal review	Immanuel Kunz (FhG), Franz Berger (Fabasoft), Anže Žitnik (XLAB)
v0.6	4.11.2021	Minor changes after 2nd internal review	Anže Žitnik (XLAB), Florian Wendland, Immanuel Kunz (FhG)
v1.0	8.11.2021	Ready for submission	Leire Orue-Echevarria (TECNALIA)
v1.01	17.8.2022	Comments from EU review implemented. Ready for internal review	Anže Žitnik (XLAB), Immanuel Kunz (FhG)
v1.02	16.9.2022	Addressed all comments received in the internal QA review	Anže Žitnik (XLAB), Immanuel Kunz (FhG)
v1.1	30.09.2022	Ready for submission	Cristina Martínez (TECNALIA)

Table of contents

List of tables

List of figures

Terms and abbreviations

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
AWS	Amazon Web Services
BSI	Bundesamt für Sicherheit in der Informationstechnik
CI/CD	Continuous Integration / Continuous Deployment
CLI	Command Line Interface
CPG	Code Property Graph
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
EUCS	European Cybersecurity Certification Scheme for Cloud Services
GA	Grant Agreement to the project
GDPR	General Data Protection Regulation
gRPC	Google Remote Procedure Call
HIPAA	Health Insurance Portability and Accountability Act
HTTP	HyperText Markup Language
IaaS	Infrastructure as a Service
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
K8S	Kubernetes
KPI	Key Performance Indicator
LSP	Language Server Protocol
Nmap	Network Mapper
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
REST	Representational State Transfer
RPC	Remote Procedure Calls
TLS	Transport Layer Security
UI	User Interface
YAML	Yet Another Markup Language
XML	Extensible Markup Language
VAT	Vulnerability Assessment Tools

Executive Summary

This deliverable presents the initial design, architecture, and implementation state of the evidence gathering components, being 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 Codyze), and assessment of organisational measures with document analysis. It gives an overview of how these components relate and interact between themselves and the rest of the MEDINA framework.

For each component, this document describes its purpose and scope, the (current and planned) coverage of the 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.

Other deliverables, closely related and worth reading for better understanding of the work presented herein, are D3.1 [2] and D5.1 [3]. D3.1 is the result of Task 3.1 and Task 3.5, which deal with evidence gathering methodology, the integration of evidence gathering tools into MEDINA (T3.1) and maintaining the trustworthiness of evidence (T3.5). While this document contains the technical details about the implementation of evidence gathering tools, deliverable D3.1 explains the methodology behind the choice and design of evidence gathering components and some related state of the art. A further analysis of the EUCS requirements' coverage with all MEDINA tools is also presented in D3.1 (Section 4.6). The external architecture of components and the relationship with all other MEDINA tools is further described in the scope of the overall MEDINA architecture in D5.1, which also lists all MEDINA functional and technical requirements, elicited in WP5.

The presented components currently have the initial prototypes implemented and ready to be (to some degree) integrated with other components of the MEDINA framework. Some requirements of the components are already fully or partially satisfied by the presented prototypes. An overview of requirement satisfaction is presented in Appendix A. There is currently no implementation of the component for assessment of organisational measures, but the methodology for its implementation and the initial architecture are presented.

Based on the work, described in this deliverable, the components will be integrated into the MEDINA framework in the scope of Work Package 5. This is the first iteration of the deliverable coming from Tasks 3.2, 3.3, and 3.4. The second version of this report with the updated components will be delivered with D3.5 [4] in project month 24 (October 2022), and the final version with D3.6 [5] in month 30 (April 2023).

1 Introduction

This deliverable is the initial result of Task 3.2, Task 3.3, and Task 3.4. It reports on the internal design and architecture, as well as the current implementation state of the tools and components, being developed in the scope of these tasks. Closely related to this document is the deliverable D3.1 [2], which is the result of Task 3.1 and Task 3.5 and contains additional details about the methodology used for the development of components described herein and their interactions. D3.1 also provides an analysis of the EUCS requirements' coverage with the MEDINA evidence gathering tools. The overall MEDINA architecture providing further details about the interaction between components is the result of WP5 and is presented in deliverable D5.1 [3], which also lists the requirements defined for all the technical components of MEDINA. Both mentioned related deliverables, D5.1 and D3.1, present the basis for the technical work reported in this document.

1.1 About this deliverable

The goal of this deliverable is to present the design and implementation of MEDINA evidence gathering components. This is a report on the initial prototype reflecting an early stage of implementation and integration of these components and is the first of three iterations of deliverables, 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 policies and processes documentation.

1.2 Document structure

This document is organised in the following sections:

1. *Introduction* gives the context for the results, reported in this document, its scope, structure, and mentions the relationship to other work in the MEDINA project.
2. *Evidence Management Tools High-level Architecture* gives an overview of the components, described in this document, and presents the architecture and relations between them.
3. *Security Assessment of Cloud Infrastructure* reports on the design and implementation of Clouditor, Wazuh, and Vulnerability assessment tools (VAT). 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* reports on the design and implementation of CloudPG and Codyze, components for cloud application source code analysis and provision of related technical evidence.
5. *Assessment of Organisational Measures* gives a report on the preliminary design and implementation state of the component for providing evidence of organisational measures based on the analysis of CSP's documentation in various forms.

Finally, Section 6 (*Conclusions*) summarizes and briefly comments on the reported results.

2 Evidence Management Tools High-level Architecture

This section gives an overview of the high-level architecture of MEDINA WP3 components, which result in the 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. Figure 1 shows the architecture and data workflow among WP3 and other related components. The following sections of this document present the technical details and the (current) state of implementation of the evidence gathering components. Each section also lists the MEDINA requirements (elicited in WP5) and their current coverage by the implemented tools. The current satisfaction of MEDINA requirements by all the described tools is presented in an overview in Appendix A.

Tools for collecting evidence about technical measures from cloud infrastructure are described in Section 3. Clouditor (see Section 3.1) is connected to the cloud interface and collects evidence about the secure configuration of cloud resources. Wazuh (see Section 3.2) is installed in the CSP's cloud infrastructure and monitors the security state of the individual (virtual) machines. Vulnerability Assessment Tools (see Section 3.3) are also installed in the CSP's infrastructure and periodically scan the configured servers and networks for vulnerabilities. The operation of Wazuh and VAT is inspected by the Wazuh and VAT Evidence Collection component which produces evidence and forwards them to the Security Assessment component, implemented as part of Clouditor. The Security Assessment component assesses the received evidence based on the target values, coming from the certification specification and CSP's configuration. For each evidence object, Security Assessment outputs a security assessment result with the information whether the addressed metric measured on the particular evaluation resource (e.g., computing resource, process, policy) is compliant or not compliant. Technical evidence, obtained from the analysis of cloud applications' source code is gathered by Codyze (see Section 4) which includes a Security Assessment part and thus outputs assessment results directly to the Orchestrator. Evidence about technical measures can also be collected by CSP-native components, which can contain their own security assessment or connect to Clouditor's Security Assessment.

The component for organisational evidence gathering and processing (see 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. This evidence is also communicated to the rest of the MEDINA framework through Clouditor's Security Assessment.

The Orchestration component (also implemented as part of Clouditor) is a central point for gathering evidence objects and their assessment results. The Orchestrator stores these data in the respective databases and makes it available to other components (e.g., Continuous Certification Evaluation, Compliance Dashboard UI). Evidence and assessment results are also forwarded to the Evidence trustworthiness management component which uses blockchain technologies to ensure the authenticity of data when retrieved at a later stage.

The overall architecture of the MEDINA framework is further presented in D5.1 [3].

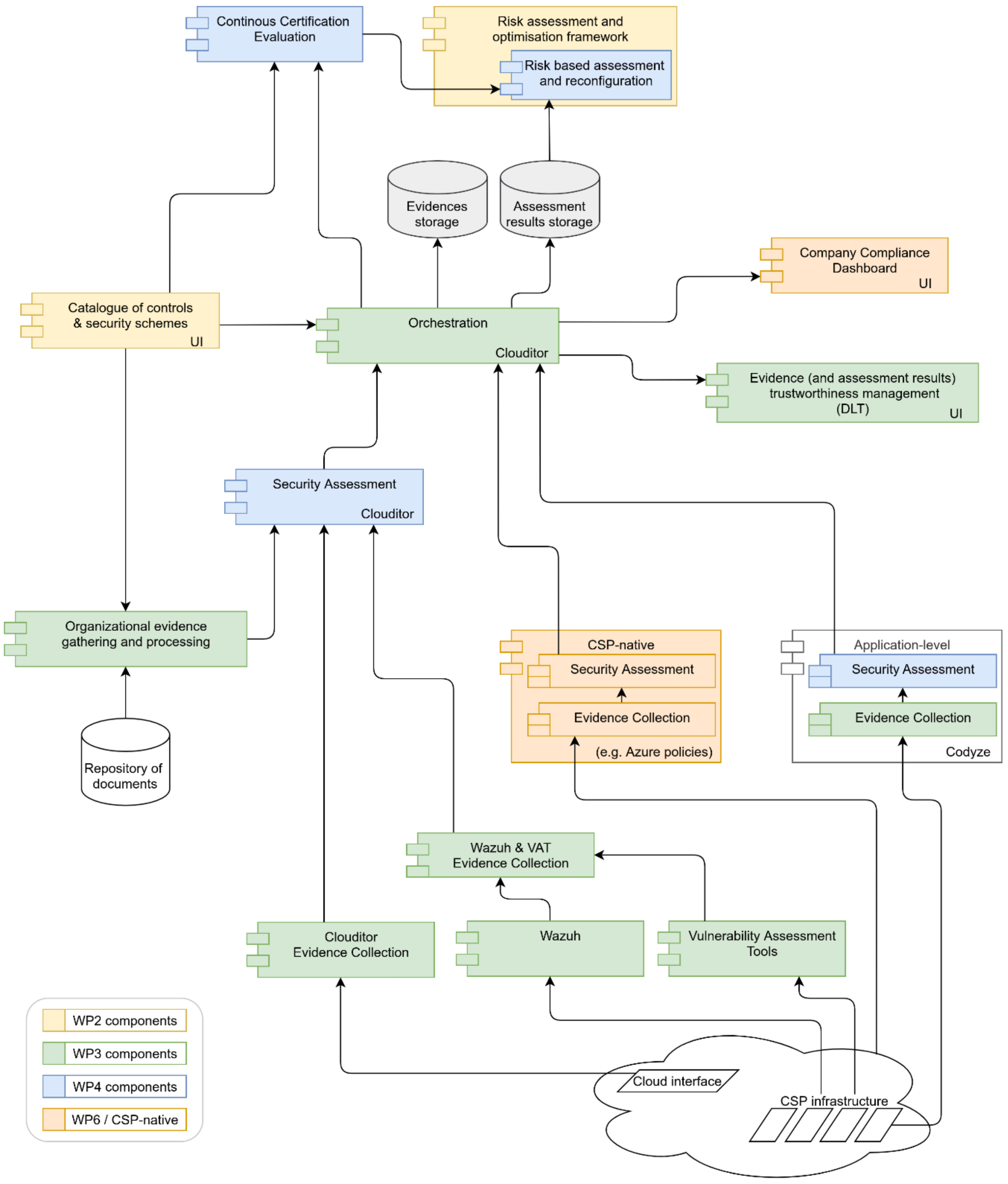

Continous Certification Evaluation
Risk assessment and optimisation framework
Risk based assessment and reconfiguration
Evidences storage
Assessment results storage
Company Compliance Dashboard
UI
Catalogue of controls & security schemes
UI
Orchestration
Clouditor
Evidence (and assessment results) trustworthiness management (DLT)
UI
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. Architecture of WP3 and directly related components (source: [2])

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 (cloud configuration, virtual machines and containers, or software running in them). The following subsections present the individual respective components, all being developed in the scope of Task 3.2.

3.1 Cloudditor

3.1 Clouditor

3.1.1 Implementation

3.1.1.1 Functional description

Clouditor is an open-source continuous cloud assurance tool. Its main goal is to continuously evaluate if cloud resources are configured in a secure way and if they comply with security requirements defined by security requirement catalogues, for example ENISA EUCS [1]. As such, it implements several components of the MEDINA framework, including evidence gathering, security assessment of evidence, and the orchestrator.

Clouditor currently supports evidence gathering in Amazon Web Services (AWS), Microsoft Azure, and Kubernetes. The resource configurations in these platforms are checked by the use 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

Many other checks are implemented for different resource types in different services, like networking, storage, and computing. The tool is therefore platform-independent and does not require agents on the target platform. Instead, resource properties are queried via API calls that are provided by the respective platform.

To implement the components in a MEDINA-compliant way, Clouditor is subdivided into microservices which conform to the components defined in MEDINA:

- the *discovery service* discovers resources and collects their properties
- the *assessment service* assesses the resource properties against defined metrics
- the *orchestrator service* is a central component for managing connections and interactions between different components

An overview of the current Clouditor architecture is shown in Figure 2.

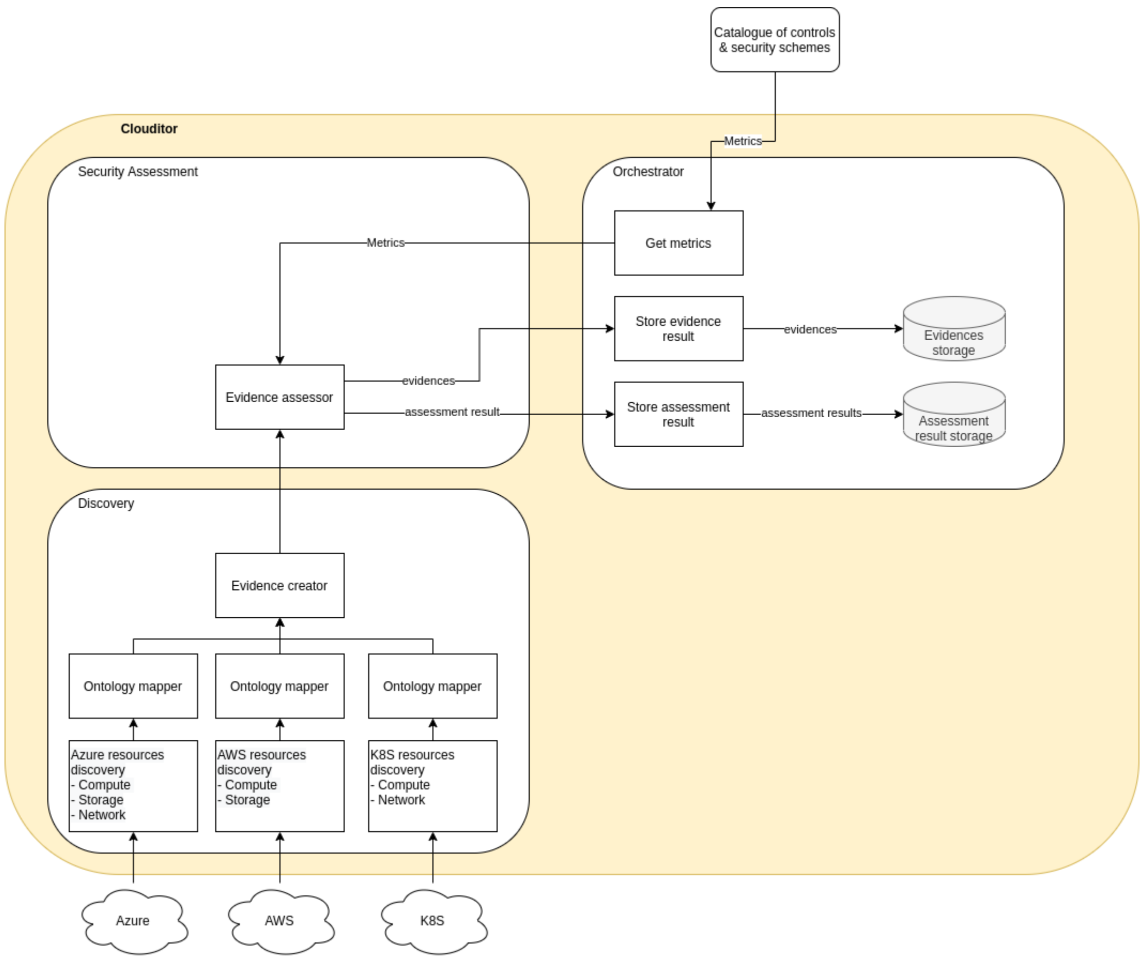


Figure 2. Overview of the Clouditor architecture: It is divided into three main components which connect to various cloud APIs to discover resource properties, to a central catalogue of controls and metrics to assess the properties against, and to the evaluation component of WP4 (not presented in this figure).

The relevant requirements from Deliverable D5.1 are listed below and a brief description of how they are implemented is given. The requirements are listed per Clouditor component.

Requirements for the discovery component

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 interval in the discovery component is set to 5 minutes and can only be changed in the source code. In the future, this will become customizable configuration through a configuration file or UI.

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 Clouditor discovery component sends the evidence to the Clouditor security assessment component via its offered APIs.

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

The Clouditor discovery component collects the evidence of cloud workloads from different CSPs (Azure, AWS, K8S). As illustrated in Figure 2, 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. These will be extended in future iterations.

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 Clouditor discovery component sends the evidence to the Clouditor security assessment by using the provided APIs. Measurement results are substituted by assessment results which are generated in the security assessment component.

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	Not implemented

The initial version of the security assessment component does not retrieve the target values from a central repository. Future releases will implement a connection to the central repository of target values.

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 result by using the provided orchestrator APIs.

Requirements for the orchestrator component

Requirement id	**ECO.01**
Short title	Provision of Interfaces

Requirement id	**ECO.01**
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	Implemented

Interfaces are provided by RPC (Remote Procedure Call) APIs with gRPC[1]. Currently, the assessment tools send assessment results accompanied by the evidence they are based on. In a future release, we foresee a separate component for storing the evidence. This modification is intended to prevent the orchestrator, which may reside on an external device from the user's perspective, from obtaining potential sensitive evidence.

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

Currently, the *discovery* component (evidence collection) is triggered via a CLI (Command Line Interface). Then the collected evidence are sent to the security assessment and the generated assessment results are then sent to the orchestrator which stores them in a database. In the future, the selected assurance level will be addressed via the selected metrics that should be assessed.

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	Partly implemented

The *orchestrator* currently stores the evidence in-memory, i.e., no further security mechanism is needed. As described above, a new component, the *evidence store*, is foreseen. It then has the responsibility to securely transfer the evidence to the evidence storage, which is currently planned to be located at the orchestrator side as well.

Requirement id	**ETM.01**
Short title	Trustworthiness of evidence
Description	The evidence orchestrator must integrate reasonable safeguards for guaranteeing the trustworthiness of collected evidence.
Implementation state	Not implemented

The initial version of the Orchestrator component does not implement a functionality to store checksums of evidence. The implementation is planned for the next iteration.

Requirement id	**ETM.02**
Short title	Transmission of evidence checksums

[1] https://grpc.io/

Description	The evidence orchestrator should integrate a Ledger client that stores checksums of evidence in a DLT.
Implementation state	Not implemented

The initial version of the Orchestrator component does not implement a functionality to store checksums of evidence. The implementation is planned for the next iteration.

3.1.1.1.1 Fitting into overall MEDINA Architecture

Figure 1 in Section 2 shows the integration of the Clouditor within the overall MEDINA architecture. The three microservices representing components are mapped to the MEDINA framework are as follows:

- the *discovery* service of Clouditor represents the *evidence collection* component in the MEDINA framework,
- the *assessment* service represents the *security assessment* component in the MEDINA framework and
- the *orchestrator* service represents the *orchestrator* component in the MEDINA framework.

The Clouditor (security) assessment component processes the evidence of the Clouditor evidence collection tool as well as evidence of other evidence collection tools, e.g. Wazuh. The orchestrator component processes the results of the Clouditor assessment component as well as results of other security assessment tools, e.g., Codyze.

3.1.1.2 Technical description

In the following the technical description of the Clouditor's components within the MEDINA framework is provided. 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., for adding an evidence collection tool for a new CSP. The discovery, security assessment and orchestrator are such modular components that represent microservices. Like all parts in Clouditor, these services are written in Go and communicate among themselves via the gRPC protocol. The three microservices representing components in the MEDINA framework are as follows:

- the *discovery* service of the Clouditor represents the *evidence collection* component in the MEDINA framework,
- the *assessment* service represents the *security assessment* component in the MEDINA framework and
- the *orchestrator* service represents the *orchestrator* component in the MEDINA framework.

An overview of the components and the data flows in the Clouditor prototype is shown in Figure 2.

Since the architecture is defined by its components and connections between them, interface snippets of the individual components are provided below. For the detailed specification see the

./proto folder within the *Clouditor* repository[2]. The specification is defined in the *Protocol Buffer Version 3 Language Specification*. In addition, see the *./openapi* folder containing for each component the corresponding auto generated *.yaml files which follow the OpenAPI description for REST APIs.*

Discovery interface

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.

Security Assessment interface

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 evidences collection tool.

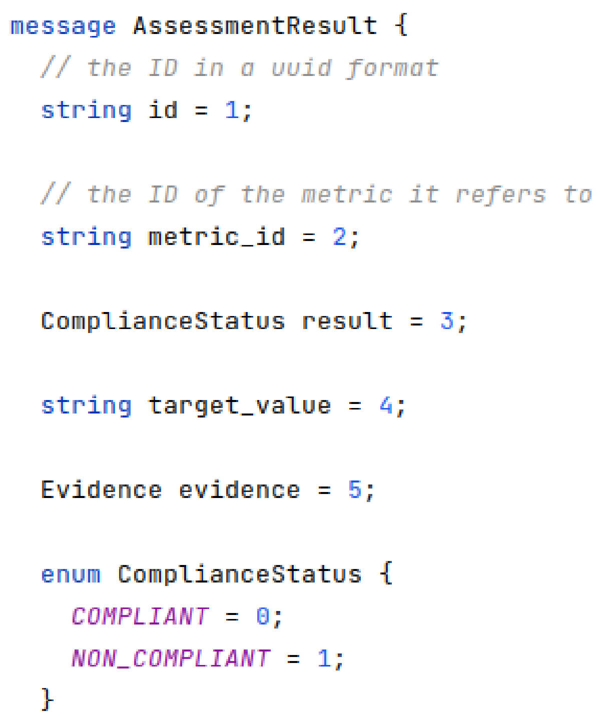
Orchestrator interface

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.

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

Function Name	Parameters	Return Type	Description
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.
StoreAssessmentResult/ StoreAssessmentResults	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 (Metric)	Returns the metric with the given metric id.
ListMetrics	-	metrics (Metric)	Returns a list of all metrics provided by the *catalogue of metrics and security schemes.*

As an example, an excerpt of the *AssessmentResult* message (type) is shown in Figure 3. It consists of a unique identifier (*id),* the corresponding metric id (*metric_id)*, target value (*target_value*) compliance status (*result)* and the corresponding *evidence*.



```
message AssessmentResult {
  // the ID in a uuid format
  string id = 1;

  // the ID of the metric it refers to
  string metric_id = 2;

  ComplianceStatus result = 3;

  string target_value = 4;

  Evidence evidence = 5;

  enum ComplianceStatus {
    COMPLIANT = 0;
    NON_COMPLIANT = 1;
  }
```

Figure 3. An example excerpt of the AssessmentResult message. Note that it is a Protobuf definition where the assigned numbers do not represent actual values, but field numbers

Note that this is the current state, and it is planned to de-couple the evidence storage from the orchestrator and to let the *discovery* forwarding the evidence directly to the new introduced component *evidence store* which is storing and returning evidence to/from the evidence storage. Furthermore, we plan to provide a monolithic prototype version of the Clouditor for a future release that bundles all microservices.

3.1.1.2.2 Description of components

This section presents the tools provided by Clouditor, describing how they have been and will be further developed to meet the MEDINA requirements.

Evidence Collection

The functionality of the *discovery* can be divided into 3 parts:

- Fetching relevant properties of cloud resources,
- Creation of evidence objects, and
- Forwarding this evidence to the security assessment component.

Within the *discovery* service, the *discovery* package is located at the top-level. Its purpose is to communicate with other services/components (in this case the Assessment component). In a first step, this service establishes a connection to the 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 [6], 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 can be discovered. Through the *Kubernetes* API *compute* and *network* resources are currently discoverable.

Security Assessment

The Clouditor's assessment tool (security assessment in the MEDINA terminology) is responsible for evaluating incoming evidence and sending the produced assessment results to the orchestrator.

Evidence is received from components which are collecting properties of a cloud service, e.g., the evidence collection tools from Clouditor or Wazuh. As mentioned above, Clouditor now follows a microservice architecture. As a result, also the assessment is a service that other tools can use to send their evidence for evaluation. Such evidence collection tools only need to implement the given public API in gRPC to send evidence as Protocol Buffer messages. Using REST over HTTP is another option for evidence collecting tools that do not use gRPC. However, the gRPC approach allows to send evidence in a stream which can significantly increase the throughput. For all defined remote procedure calls, see the API definition in Section 3.1.1.2.1.

In the previous version of Clouditor, a dedicated policy rule language was defined for assessing evidence. Since no other tools outside the Clouditor tool suite needed to be connected to it, this approach was sufficient. In MEDINA, however, various microservices are used to decouple the individual applications so that new or existing components (e.g., Wazuh) can be smoothly integrated with them. To simplify the definition of policies, a more commonly used policy language, Rego from Open Policy Agent (OPA)[3], was introduced instead. OPA introduced Rego as a uniform declarative policy language. A policy written in Rego is asserting an input (e.g., an evidence) against user-specific constraints (target values and operators). This is also the place where the cloud resource ontology (see D2.3 [6]) comes into play: Since evidence provided by the evidence collecting tools adhere to it, the Rego policies only need to specify rules based on properties following the format of the ontology. E.g., see Figure 4 for a policy which checks if the algorithm used for encrypting data at rest of the input is higher than the given target value. Figure 5 shows the user-specific constraint that the algorithm has to be at least 256. The input is illustrated in Figure 6: 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, these policies will be stored as metrics in the catalogue of controls and security schemes component [7] and fetched by the orchestrator. 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. Since WP3 is now in the phase of implementing and testing the metrics as well as the security assessment, the metrics deduced so far are currently stored directly with the assessment. In a next step, these metrics are first outsourced to the orchestrator and after that iteration finally stored in the metrics catalogue.

[3] 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 4. 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 5. Sample data that will be provided in the future by the central catalogue of metrics and target values.

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

Figure 6. A sample excerpt of an 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 D4.1 [8]).

Orchestrator

The orchestrator is a central component in the MEDINA framework and acts as a central management component for launching WP3 components and orchestration of dataflows between components. As such, it also manages the interaction between work packages as shown in Figure 1. The orchestrator offers APIs to store and retrieve data and manage assessment tools. The APIs are defined in gRPC such that the components only need to 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. An overview of the provided interfaces can be found in 3.1.1.2.1. Furthermore, data is forwarded to the respective components, e.g., WP2 or WP4. Its interactions with other components and its functionalities are summarized in the following:

- **Security assessment:** 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.
- **Catalogue of metrics and security schemes:** The orchestrator also acts as the central interface to the catalogue of metrics and security schemes which is developed in the

scope of WP2 (see deliverable D2.1 [7]). As such, it is responsible for providing relevant metrics to the assessment component. Note that this integration has not been developed yet.

- **Distributed Ledger Technology (DLT):** The orchestrator stores the checksums of the evidence and assessment results in the DLT. The implementation of the trustworthiness management system component is reported in D3.1 [2]. Currently, it is not yet decided which component calculates the checksums and the forwarding is not yet implemented.
- **Continuous certification evaluation:** The orchestrator forwards the assessment results to the *continuous certification evaluation* component which is developed in WP4 (see deliverable D4.1 [8]). Currently, the connection to the component is not yet implemented.
- **Data storages:** The orchestrator stores the evidence as well as the assessment results into the associated storages. Currently, it is not yet decided which kind of database technology shall be used, since the selection also depends on the evaluation within Task 4.2, which discusses possible technologies for storing evidence securely. Until this is decided, the prototype stores data in memory.

3.1.1.2.3 Technical specifications

The prototype is written in Go (version 1.16). A selection of key libraries is shown in the following and a full list of used libraries can be found in the Github repository[2].

- *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 DB or a postgres DB 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[2].

3.1.2.1 Package information

The structure of the important folders and a brief description is shown in Table 1.

Table 1. Overview of the package structure

Folder	Description
api/	This folder contains 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 Installation instructions

The full up-to-date installation instructions can be found in the README at the Clouditor Github repository[2]. A copy of the recent version is given in Appendix B.

To build the Clouditor the Gradle build tool[4] 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 2 commands:

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

The engine could 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. The detailed instructions can be found in the Readme file at the Github respository[2].

- 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

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

- Install and use the CLI for running the Clouditor at runtime

3.1.2.3 User Manual

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

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

Usage:
  cl [command]

Available Commands:
  assessment      Assessment result commands
  completion      Generate completion script
  discovery       Discovery commands
  help            Help about any command
  login           Log in to Clouditor
  metric          Metric 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 –help*.

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

3.1.2.4 Licensing information

Clouditor is licensed under the Apache License 2.0.

3.1.2.5 Download

The Clouditor source code can be found in the Clouditor Github respository[2].

Source code of the individual Clouditor services can also be accessed in the MEDINA GitLab here:

- Evidence Collection: https://git.code.tecnalia.com/medina/public/cloud-evidence-collector
- Security Assessment: https://git.code.tecnalia.com/medina/public/security-assessment
- Orchestrator: https://git.code.tecnalia.com/medina/public/orchestrator

3.1.3 Advancements within MEDINA

Several modifications and features have been implemented in Clouditor within the first year of the MEDINA project:

- The project has completely been 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 D2.3 [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[5] 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.

3.1.4 Limitations and future work

Discovery (Evidence Collection)

The *discovery* service, which currently collects evidence from Microsoft Azure systems, is limited by the access rights that are given in the 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 the *discovery* service. Also, the evidence collection is limited by the information that the cloud provider APIs implement: If certain encryption property, for example, would not be implemented by an API, the evidence collection for that property would not be possible. Since *discovery* adds ontological terms to the evidence, also limitations of the ontology need to be taken into account. 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.

Furthermore, the collection is currently limited to a small set of Azure and AWS services, which we will expand in future iterations.

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, at the time of writing, in version 0.42.2; 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.

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

The Security Assessment also currently only applies a small set of metrics which we will expand according to the KPIs.

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.

The main limitation regarding the Orchestrator is that it currently can only manage one cloud service. In the next iteration, we will therefore add support for multiple cloud services, as well as for multiple certification frameworks and assurance levels. Also, the integration of the Orchestrator with some of the other MEDINA components still needs to be finished in a future iteration.

3.2 Wazuh

3.2.1 Implementation

3.2.1.1 Functional description

Wazuh[6] 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. Wazuh agents can run on many different platforms, such as Windows, Linux, Mac OS X, AIX, Solaris, and HP-UX. Unlike Clouditor, Wazuh is not primarily connected to the cloud interfaces, but its agents are installed directly on the (virtual) machines of the monitored infrastructure.

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:

- 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 [9] or VirusTotal [10], 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.

[6] https://wazuh.com/

- 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 we are planning to provide in order to satisfy listed requirements 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 MEDINA. The controls that can be satisfied with Wazuh relate to malware protection, logging, threat analytics, and automatic monitoring (alerting). The initial analysis of EUCS requirements covered by Wazuh is further described in D3.1 [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-05.3**. The 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 OPS-05.1*". **OPS-05.1** states: "*The CSP shall deploy malware protection, if technically feasible, on all systems that support delivery of the cloud service in the production environment, according to policies and procedures*". According to the descriptions of these requirements, the conditions for regarding a machine compliant with OPS-05.3 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 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.

Related requirements

Below is the collection of requirements (from D5.1 [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.
Implementation state	Partially implemented

Integration with the Evidence Collector component is implemented for a limited number of evidence types (metrics). Continuous collection is implemented, but the collection intervals are currently configurable only manually.

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	Partially implemented

Interface between the Wazuh & VAT Evidence Collector and Clouditor (providing the security assessment capabilities) is currently in a prototype state.

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 with the Evidence Collector allows MEDINA to verify the malware detection state and gather evidence about it, helping to verify the compliance with certain controls of standards.

3.2.1.1.1 Fitting into overall MEDINA Architecture

Wazuh is integrated with the rest of the MEDINA framework through the 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 (if required) the security events, gathered by Wazuh, are queried by the Evidence Collector, which forwards such information to Clouditor (security assessment) in the form of evidence. Positioning of Wazuh in the architecture among the WP3-related components is shown in Figure 1.

3.2.1.2 Technical description

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 as the agents.

The server includes the Wazuh 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 Wazuh to any external component.

High level architecture of Wazuh is depicted in Figure 7 below. 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.

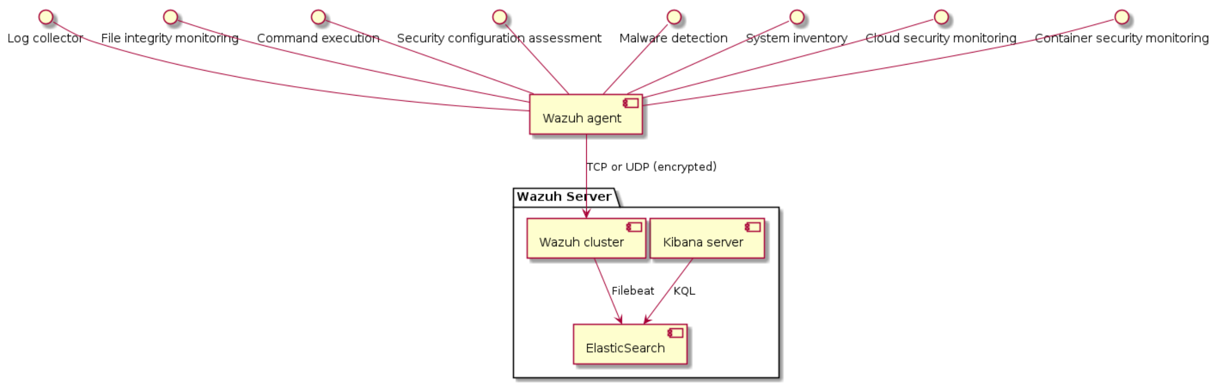


Figure 7. High-level Wazuh's architecture.

3.2.1.2.2 Description of components

Agents communicate with the server using Rsyslog. Wazuh is plugged into MEDINA with the Wazuh & 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. It also includes two-way communication with the security assessment component (Clouditor). This is depicted below in Figure 8.

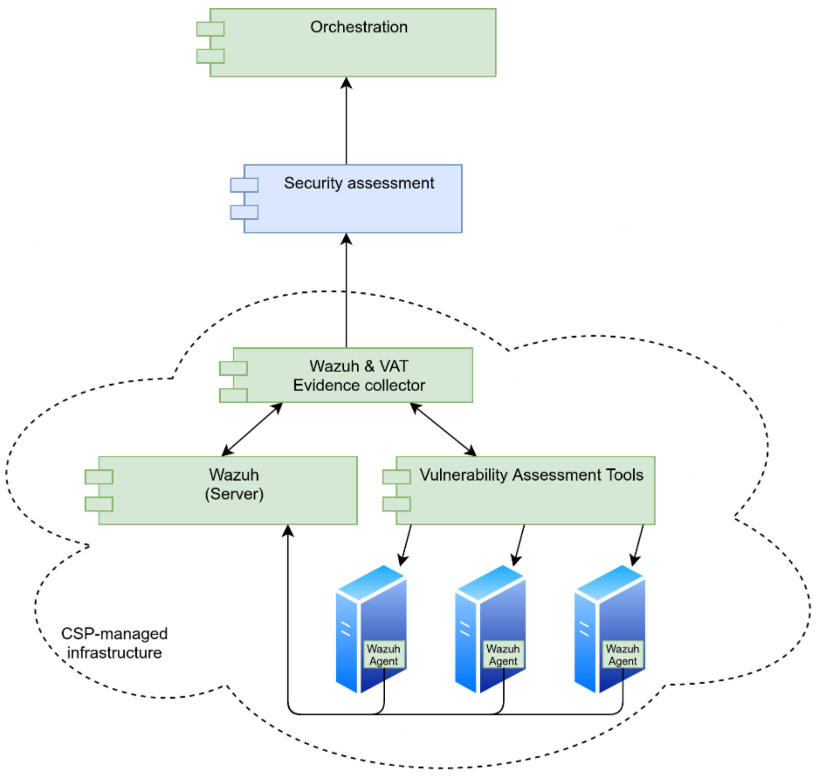


Figure 8. HIgh-level schema of Wazuh, VAT, and related components.

3.2.1.2.3 Technical specifications

The prototype's implementation mainly consists of:

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

3.2.2 Delivery and usage

3.2.2.1 Package information

The package is delivered in a repository, containing all the needed deployment and configuration scripts for installing Wazuh and the Evidence Collector. For demonstrative purposes and replicating a deployment on CSP's infrastructure, the process creates four virtual machines: a Wazuh server, two Wazuh agents, and a machine with the Evidence Collector.

The tree structure is as follows:

```
.
├── Makefile
├── README.md
├── ansible
│   ├── clamav
│   │   └── tasks
│   │       └── install-clamav.yml
│   ├── custom-integration
│   │   ├── files
│   │   │   ├── custom-integration
│   │   │   └── custom-integration.py
│   │   └── tasks
│   │       └── main.yml
│   ├── docker
│   │   ├── credentials
│   │   │   ├── credentials.yml
│   │   │   └── vars.yml
│   │   └── tasks
│   │       └── main.yml
│   ├── globals
│   │   ├── globals.yml
│   │   └── vars.yml
│   ├── provision-agents.yml
│   ├── provision-evidence-collector.yml
│   ├── provision-managers.yml
│   └── provision.yml
└── environments
    └── vagrant-full-setup
        ├── Vagrantfile
        ├── inventory.txt
        └── vagrant-full-setup.mk
```

The 'ansible' directory contains all available ansible roles. The 'environments' directory contains available environments towards which the ansible playbook can be executed.

3.2.2.2 Installation instructions

Requirements:

- Vagrant 2.2.14
- Ansible 2.9.16

To setup the demo, simply provision the Wazuh server, Wazuh agents, and Evidence Collector machines:

```
$ make create provision
```

3.2.2.3 User Manual

To access the Wazuh UI, navigate your browser to: https://192.168.33.10:5601[7] and login with the default credentials (*admin:changeme*). Navigate to "Wazuh" section on the left hand-side.

You should see 2 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 [11] component is open source, licensed with a modified GPLv2 license[8].

The deployment scripts for the MEDINA proof-of-concept and the Wazuh & VAT Evidence Collector, developed by XLAB, are licenced with the open-source Apache 2.0 licence. All source code repositories contain a LICENSE file in their root directories.

3.2.2.5 Download

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

- https://git.code.tecnalia.com/medina/public/wazuh-vat-evidence-collector
- https://git.code.tecnalia.com/medina/public/wazuh-deploy

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, an analysis of the EUCS requirements was conducted with regards to Wazuh to determine which of the requirements can be verified or satisfied by using Wazuh. Architecture of integrating Wazuh with MEDINA was defined along with the design for the Evidence Collector component. A prototype of the Evidence Collector was implemented that connects with Wazuh and produces evidence for metrics about malware protection use on the targeted machines.

3.2.4 Limitations and future work

In the current state (project month 12), evidence gathering with Wazuh is only possible for a very limited number of metrics. The supported metrics relate to the (draft) EUCS requirement OPS-05.3. During further course of the project, this limitation will be progressively removed by adding support for other metrics.

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.

[7] This URL points to the VM that is created by running the mentioned deployment scripts
[8] https://github.com/wazuh/wazuh/blob/master/LICENSE

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

3.3 Vulnerability Assessment Tools

3.3.1 Implementation

3.3.1.1 Functional description

Vulnerability Assessment Tools (VAT) act as a vulnerability scanning and detection framework. It 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.

These 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, 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 can be gathered about their fulfilment. EUCS requirements covered by VAT include several from the vulnerability detection and management categories, usage of encrypted communication protocols, separation of networks and monitoring new devices on the network, etc. Coverage of EUCS requirements by VAT is also described in deliverable D3.1 [2].

Additional innovation of VAT itself is the modularity and flexibility of the VAT framework. Beside the included vulnerability detection tools, users can 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. Beside the detection of vulnerabilities and provision of related evidence, the framework thus also enables implementation of custom detectors to produce other evidence types or monitor other CSP-specific networking metrics.

Related requirements

Below is the collection of requirements (from D5.1 [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.
Implementation state	Partially implemented

VAT framework enables configuration of the scanning tasks and continuous scanning with configurable intervals. The integration with Evidence Collector and other components is not yet implemented.

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	Not yet implemented

The interface for communication between VAT and the Evidence Collector is not yet 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	Partially implemented

VAT includes several vulnerability scanners and a framework for their orchestration and automated running of the scans. Possibility to add custom vulnerability scanning scripts is also implemented. The integration with the Evidence Collector and other MEDINA components is not yet implemented.

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 8. VAT scans the monitored machines inside the CSP's infrastructure, which is communicated to the Wazuh & 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.

3.3.1.2 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 comprises of several microservices and is presented below in Figure 9. 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 a 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 & VAT Evidence Collection component (also see Figure 8).

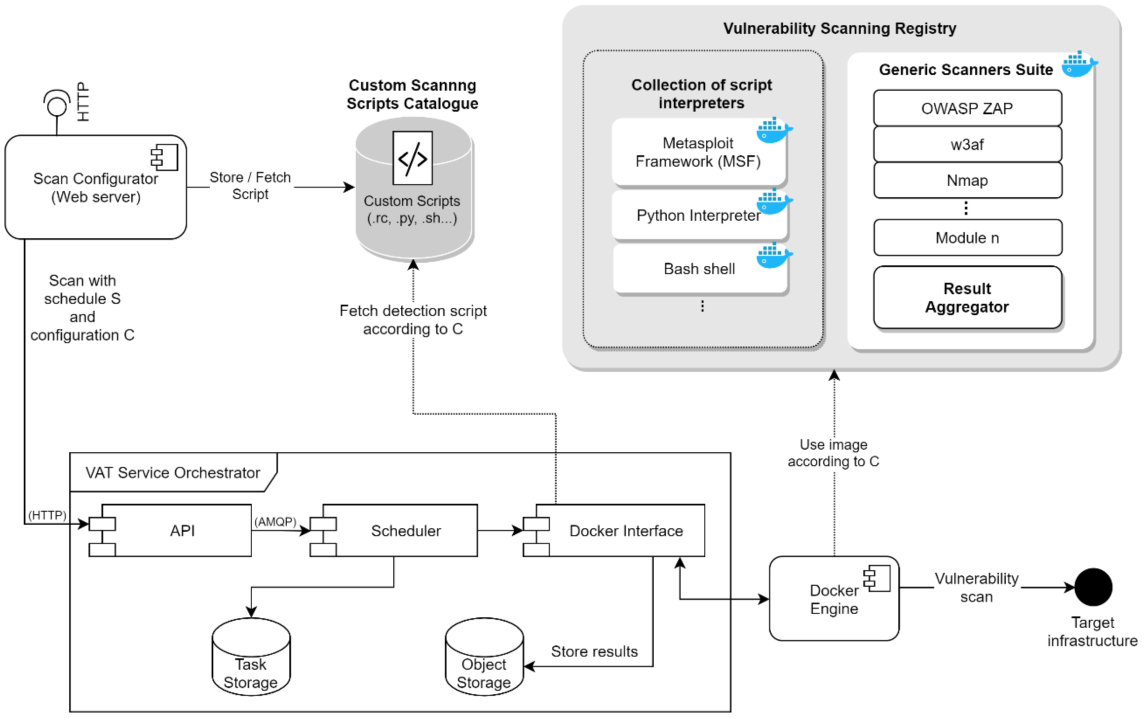


Figure 9. Internal architecture schema of Vulnerability Assessment Tools.

3.3.1.2.2 Description of components

Components, comprising Vulnerability Assessment Tools, are described below.

Scan Configurator is 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.

These 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 is 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 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.

Beside the Generic Suite, it also holds several Docker images [15] of **script interpreters** that can run user-provided custom scanning scripts. These 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.

VAT Service Orchestrator 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 the figure 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 & VAT Evidence Collector** component (shown in Figure 8), 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 is the component responsible for triggering scanning tasks according to their configured schedules. **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]). Cscan package (part of the open-source Faraday vulnerability scanning platform [21]) along with several additional Python and Bash scripts are included for triggering the scanning modules according to configuration parameters. The Result Aggregator is written in Python and outputs a JSON file containing outputs of all the scanning modules used.

Custom script interpreters are separate Docker images with the respective tools.

Communication among the core components is carried out with AMQP through a RabbitMQ [17] server. The API component exposes an HTTP REST API.

3.3.2 Delivery and usage

3.3.2.1 Package information

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

3.3.2.2 Installation instructions

Deployment scripts are provided using Vagrant [22] and Ansible [23] in the "vat-deployment" repository. Two machines will be provisioned:

- "VAT management" containing the VAT Service Orchestrator, Scan Configurator, and the catalogue of custom scripts,

- "VAT runtime" with the Docker engine for running the scanning tasks and the necessary images installed (Vulnerability Scanning Registry).

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

```
make create provision
```

3.3.2.3 User Manual

Navigate your internet browser to the IP address of the management machine and login with the default credentials: test-user, test-password. You will be able to 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 2.0 licence.

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

- OWASP ZAP (Apache Licence) [13]
- w3af (GPLv2) [12]
- Nmap (modified GPLv2)[9] [14]
- Faraday (GPLv3) [21]
- Metasploit (BSD) [16]

3.3.2.5 Download

VAT deployment demo repository is available at MEDINA's public GitLab:

https://git.code.tecnalia.com/medina/public/vat-deploy

The Generic Scanning Suite source code repository is also available in MEDINA's public GitLab:

https://git.code.tecnalia.com/medina/public/vat-genscan

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.

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 detection of vulnerabilities as well as for scheduling of various actions connected to defence and attacks of infrastructure in a controlled and enclosed cyber range environment. In the first 12 months of MEDINA, an analysis was made to determine the EUCS requirements that are feasible to be verified or satisfied by VAT. 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 integration with MEDINA components. The

[9] https://nmap.org/npsl/

Evidence Collector component was developed with a specific module to interact with VAT and produce relevant evidence.

3.3.4 Limitations and future work

In the current (month 12) state of implementation, VAT is not yet fully integrated with the Evidence collector, thus providing evidence with VAT is not possible.

As described above, VAT is composed of multiple modules: several vulnerability scanners and 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 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 or malware detection tools deployed on certain systems and to monitor their results. By using 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.

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 are described in the following subsections.

4.1 Cloud Property Graph

Cloud Property Graph (CloudPG) is another tool, developed within the first year of the MEDINA project. It combines source code analysis with infrastructure analysis. To that end, a library for static code analysis, the cpg[10], has been extended with analysis logic for cloud workloads. The implementation of this tool is being developed in GitHub[11].

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.

A scientific paper about this approach and tool has been written and accepted at the IEEE International Conference on Cloud Computing 2021 (CLOUD), but not yet published at the time of writing.

4.1.1 Implementation

4.1.1.1 Functional description

The cpg creates a property graph of source code that is enhanced by the CloudPG with information about the current resource configurations, as well as data flows between resources.

Figure 10 shows an excerpt of a graph generated by the Cloud Property Graph. This example 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.

Using such a graph 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 later using pre-defined metrics and target values.

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 D2.3 [6], and the security properties it defines. Consequently, security-relevant concepts can be analysed across source code and infrastructure configurations.

4.1.1.1.1 Fitting into overall MEDINA Architecture

The CloudPG is a separate service implementing an evidence gathering component. In the current state, it is not yet integrated with other components. Two possible approaches exist for

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

its integration: First, its output is adjusted such that it provides evidence in the required MEDINA format to the Clouditor assessment service. In this case the CloudPG's graph-based analysis results (see Figure 10) 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.

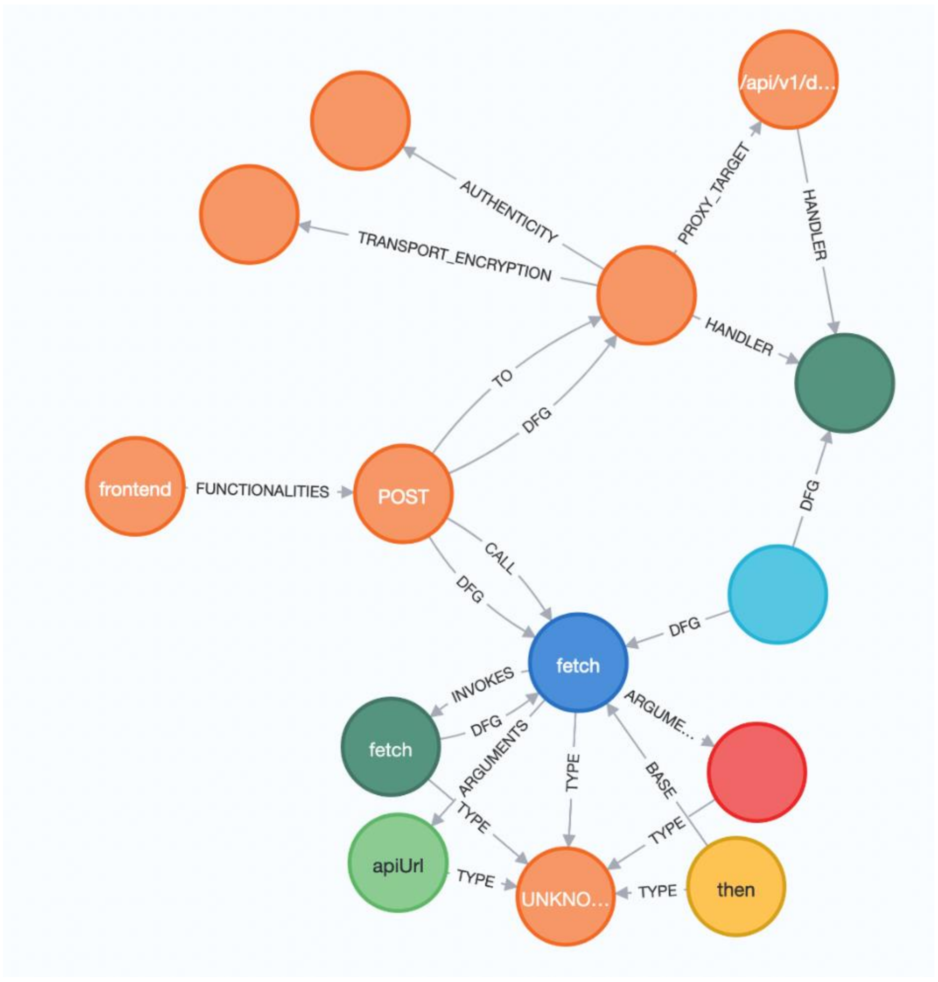


Figure 10. An excerpt from the graph generated by the Cloud Property Graph

4.1.1.2 Technical description

4.1.1.2.1 Prototype Architecture

The CloudPG employs a straightforward architecture. First it uses the cpg library to build a code property graph of available 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[12] 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

4.1.1.2.2 Description of components

The CloudPG is not divided into separate components. Yet, 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.

4.1.2 Delivery and usage

4.1.2.1 Package

The tool is not yet available as a Docker image. It currently needs to be installed as described below.

4.1.2.2 Installation

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[13]. 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, follow the instructions at https://github.com/Fraunhofer-AISEC/cpg#python
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. (For further informaton see https://github.com/Fraunhofer-AISEC/cpg#usage-of-experimental-languages)
        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.

4.1.2.3 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

[12] https://pkg.go.dev/github.com/rs/zerolog
[13] https://github.com/clouditor/cloud-property-graph/blob/main/README.md

follows: *cloudpg/build/install/cloudpg/bin/cloudpg --root=/x/testprogramm folder1/ folder2/ folder 3/*

4.1.2.4 Licensing

The tool is licensed under the Apache 2.0 license.

4.1.2.5 Download

The project is available as an open-source project on GitHub[14].

4.1.3 Advancements within MEDINA

The Cloud Property Graph is based on the cpg which is a project that is developed independently of MEDINA. The CloudPG's additions described above, however, have completely been developed within the MEDINA project. In its approach of combining source code analysis with infrastructure analysis, it complements Codyze which is described in the next section.

The CloudPG is not yet integrated with the overall MEDINA architecture, since its output is a graph which is currently not assessable with the Clouditor assessment service. A possible integration approach is to extend it with a custom assessment logic that is integrated with the orchestrator.

4.1.4 Limitations and future work

The approach we have 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, its integration with the MEDINA framework, i.e., with the Security Assessment or Orchestrator should be addressed in future work.

4.2 Codyze

By presenting the structure and implementation of Codyze, this section reports on the results of Task 3.3.

4.2.1 Implementation

4.2.1.1 Functional description

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. High-level security requirements from catalogues like ENISA EUCS 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 high-level requirement is sufficiently realized in software.

Codyze uses the MARK DSL [25] 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

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

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 can analyse source code written in C/C++ and Java. Moreover, it ships with MARK rules for cryptographic libraries Bouncy Castle for Java and Botan for C++. Thus, source code can be checked if cryptographic operations with Bouncy Castle or Botan are properly implemented and thereby attest state-of-the-art cryptography of sufficient strength.

As Codyze analyses source code, 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 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.

The relevant requirements from D5.1 [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	Partially implemented

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	Not implemented

Currently, the interface is not implemented in Codyze.

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	Implemented

Codyze is able to 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	Not implemented

While Codyze is able to verify security requirements defined in the MARK DSL, it is not yet able to verify MEDINA-related requirements, e.g., written in Rego.

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

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.

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	Partly implemented

Source code is provided as part of the CI/CD pipeline.

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	Partly implemented

Codyze supports the programming languages C/C++ and Java. In addition, Codyze ships with rules for cryptographic libraries for Bouncy Castle and Botan. The underlying components can parse source code in JavaScript, Python and Go. However, the analysis and evaluation in Codyze is not yet 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	Partially implemented

Codyze is provided as a container image. It can be integrated as a validation step in CI/CD pipelines. The integration, usage and suggested configuration must still be documented.

4.2.1.2 Fitting into overall MEDINA Architecture

Figure 1 in Section 2 shows the integration of Codyze within the overall MEDINA architecture (see Application-level evidence collection and security assessment).

Codyze assesses the source code of cloud application and ensures compliance to security requirements catalogues like ENISA EUCS within applications. Thereby, it ensures that the individual applications and services at CSPs comply with security requirement catalogues. Other tools in the MEDINA framework such as Clouditor ensure that applications and services are consumed in a secure manner.

4.2.2 Technical description

4.2.2.1 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 source code of cloud application and services. Therefore, there are no server components nor agents.

In Figure 11, the architecture of Codyze is depicted. Codyze provides two frontends. One is an implementation of the Language Server Protocol. This interface is mainly used by developers during the development process. It can provide immediate feedback to a developer if source code changes comply to defined security requirements. The second interface is 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.

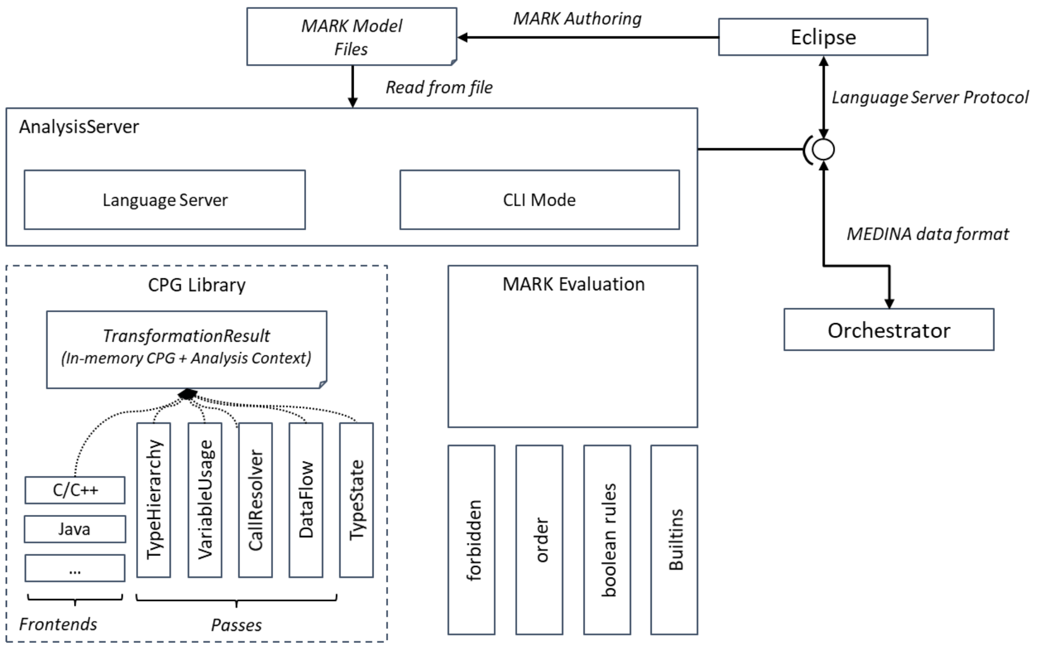


Figure 11. Codyze architecture

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 via the Language Server Protocol. In addition, findings are generated as console output. This output will be submitted to the orchestrator of the MEDINA framework in the specified data format.

4.2.2.2 Description of components

Codyze does not have separate components. It uses libraries as dependencies to implement its functionality. Codyze itself is a single, self-contained tool.

4.2.2.3 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 as specified by the German BSI. These rules are specific to cryptographic libraries Bouncy Castle and Botan.

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.

4.2.3 Delivery and usage

4.2.3.1 Package information

The project structure of Codyze with the important folders is presented with short descriptions in Table 2.

Table 2. Overview and description of package structure for Codyze

Folder	Description
docs/	Content and templates for static website content published at https://www.codyze.io/. Among others, the website contains documentation on Codyze and MARK. It describes how the tools are installed, used and worked with.
plugins/vscode	A plugin for Visual Studio Code that calls out to Codyze using LSP to analyse source code while a programmer develops an application or service. The analysis results are displayed in Visual Studio Code like compiler warnings and problems. This interface is meant for direct interaction with developers.
src/	Contains the source code for Codyze including MARK files shipped with Codyze releases.
src/dist/mark/	Predefined set of MARK files with entities and rule specifications that Codyze will validate when analysing source code.
src/main/	Source code implementing Codyze with its user interface and the analysis and evaluation engine.
src/test/	Unit tests for Codyze

Codyze uses the MARK DSL to specify entities and rules. MARK itself has its own code repository. The MARK package is structured as described in Table 3.

Table 3. Overview and description of package structure for Codyze's subcomponent MARK

Folder	Description
de.fraunhofer.aisec.mark/	Contains the Xtext grammar for MARK as well as the generated parser.
de.fraunhofer.aisec.mark.feature/	Bundles MARK as an installable feature for the Eclipse IDE.
de.fraunhofer.aisec.mark.ide/	IDE components for Eclipse IDE like customized editors.
de.fraunhofer.aisec.mark.ui/	UI components for Eclipse IDE to write MARK files. It provides content assist, code completion, quick fixes and other IDE features.
de.fraunhofer.aisec.mark.updatesite/	Defines the configuration for an Eclipse IDE update site. From this update site, Eclipse IDE can retrieve the MARK feature and install it into an Eclipse IDE.

4.2.3.2 Installation instructions

The full up-to-date installation instructions are described by the README in the Codyze GitHub repository and in the MARK GitHub repository. In addition, the website for Codyze -- https://www.codyze.io/ -- provides documentation with installation instructions. In particular, the latter contains special installation instructions for the various plugins into Eclipse IDE.

Codyze uses the build tool Gradle. The source code ships with the Gradle wrapper such that Gradle can be executed without prior installation. The wrapper is a shell and Windows batch file that downloads, installs and executes Gradle.

To build Codyze, one can run the following command:

- `./gradlew[.bat] clean build test installDist`

The command cleans out previous compilation files (`clean`), compiles the sources (`build`), runs any unit test (`test`) and creates an installable binary (`installDist`). Afterwards, the executable Codyze binary can be found 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 help looks like:

```
$ ./codyze --help
Usage: codyze [-hV] [--enable-go-support] [--enable-python-support] [--no-good-findings]
              [-o=<file>] [-s=<path>] [--timeout=<minutes>] [-m=<path>[,<path>...]]... (-c | -l |
              -t) [[--typestate=<NFA|WPDS>]] [[--analyze-includes] [--includes=<includesPath>[:|;
              <includesPath>...]]...]
Codyze finds security flaws in source code
  -s, --source=<path>       Source file or folder to analyze.
  -m, --mark=<path>[,<path>...]
                            Loads MARK policy files
  -o, --output=<file>       Write results to file. Use - for stdout.
      --timeout=<minutes>   Terminate analysis after timeout
                              Default: 120
      --no-good-findings    Disable output of "positive" findings which indicate correct
                              implementations
      --enable-python-support
                            Enables the experimental Python support. Additional files need to be
                              placed in certain locations. Please follow the CPG README.
      --enable-go-support   Enables the experimental Go support. Additional files need to be placed
                              in certain locations. Please follow the CPG README.
  -h, --help                Show this help message and exit.
  -V, --version             Print version information and exit.
Execution mode
  -c                        Start in command line mode.
  -l                        Start in language server protocol (LSP) mode.
  -t                        Start interactive console (Text-based User Interface).
Analysis settings
      --typestate=<NFA|WPDS>
                            Typestate analysis mode
                            NFA:  Non-deterministic finite automaton (faster, intraprocedural)
                            WPDS: Weighted pushdown system (slower, interprocedural)
Translation settings
      --analyze-includes    Enables parsing of include files. By default, if --includes are given,
                              the parser will resolve symbols/templates from these include, but not
                              load their parse tree.
      --includes=<includesPath>[:|;<includesPath>...]
                            Path(s) containing include files. Path must be separated by :
                              (Mac/Linux) or ; (Windows)
```

An example usage of Codyze is described in the README of the GitHub repository as well as on the Codyze website.

In addition to building Codyze from source, the GitHub repository provides build artifacts and release binaries. Releases are distributed as Zip archives and can be downloaded directly from the Codyze GitHub repository. The archive has the same structure as the `{project-dir}/build/install/codyze/` folder. Moreover, Codyze is provided as a container image and can be run by container runtimes like Docker. The container image will execute the Codyze script when the container image is run.

MARK uses the build tool Maven. The source code ships with the Maven wrapper such that Maven can be executed without prior installation. The wrapper is a shell and Windows batch file that downloads, installs and executes Maven.

To build MARK, one can run the following command:

- `./mvnw[.cmd] clean install`

The command cleans out previous compilation files (`clean`) and compiles the source code and installs it into the local Maven repository (`install`). Afterwards, Codyze can retrieve MARK as a library dependency.

Usually, MARK does not need to be built from source. MARK is a library dependency for Codyze. It is pulled in when building Codyze either from the local repository, as an artifact automatically built from source or from Maven Central repository.

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

4.2.3.3 User Manual

A user manual for Codyze and MARK can be found at the Codyze website -- https://www.codyze.io/.

4.2.3.4 Licensing information

Codyze and its DSL library MARK are licensed under the Apache License 2.0.

4.2.3.5 Download

The Codyze source code can be found in the Codyze GitHub repository:

- https://github.com/Fraunhofer-AISEC/codyze

The MARK source code can be found in the MARK GitHub repository:

- https://github.com/Fraunhofer-AISEC/codyze-mark-eclipse-plugin

The Codyze Wrapper which integrates it with the MEDINA framework can be found in the public MEDINA GitLab:

- https://git.code.tecnalia.com/medina/public/codyze

4.2.4 Advancements within MEDINA

Codyze has not yet been further developed in the scope of MEDINA. Within the upcoming tool integration efforts, however, it will be adapted to the MEDINA data model and integrated with Clouditor.

4.2.5 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. A limitation that occurs in the integration of existing tools like Codyze is that metric details, e.g., target values, that shall be changed cannot currently be conveyed to these tools. Future work should therefore explore the possibilities to enhance MEDINA's data model in that regard.

5 Assessment of Organisational Measures

This section presents the technical design of the MEDINA component for the assessment of organisational measures. This is the result of activities in the scope of Task 3.4. This section describes the initial design for the component, while the methodology is further described in D3.1 [2]. The detailed technical implementation of the component will be presented in the subsequent versions of this report.

5.1 Functional description

The main goal of this component is to aid in continuously determining the compliance status to a selected set of organisational requirements from security catalogues like ENISA EUCS. Deliverable D3.1 Section 2.3 [2] describes the underlying problems and possible solutions that build the foundations of this task's research goal and prototypical implementation. The main problem identified so far is related to the fact that different CSPs use different kinds of evaluation of single organisational measures and various types of evidence documents, thus additional research is required before implementing the component. This chapter provides an overview of the interactions of involved (sub)-components and the technical details for the research prototypes.

The component is a tool to process documents that reflect the raw, unprocessed evidence for security requirements. Depending on the document type (e.g., image, text document) and content, this component will use different subcomponents to extract "audit-relevant" information or provide a status of the evidence – e.g., is it compliant/similar to a predefined evidence document. Using predefined metrics (like for the technical assessment components in MEDINA) for the organisational requirements – paired with this component's information extraction functionalities - the organisational requirements can be transformed into automatic assessable structure. This renders at least a subsection of the organisational requirements technical (from an evaluation/assessment point of view). Of course, this needs additional research and will most certainly only work in a subset of the organisational requirements – and could be CSP dependent, or at least be dependent on the infrastructure status (how much is already automated, digitalized).

5.1.1 Fitting into overall MEDINA Architecture

Figure 2 shows the integration of this component within the MEDINA architecture. The component will interact with Clouditor to integrate the assessment results in the continuous assessment cycle. If possible, for a document category, the evidence information is passed to Clouditor in a way similar to the input of a CSP specific or MEDINA evidence collection component. Otherwise, the component internal assessment result will be passed in a meaningful way to be compatible to the rest of the MEDINA architecture.

If feasible, metrics and/or assessment rules will be retrieved from the catalogue of controls & security schemes (see [7] for more information) to be used for the assessment and for extracting the relevant information.

5.2 Technical description

The following subsections describe the envisioned technical details of the component.

5.2.1 Prototype architecture

Figure 12 depicts the component's abstract interactions with other MEDINA components. The component interacts with the catalogue of controls & security schemes to retrieve data like requirements and metrics. From the repository of documents, the raw evidence to be processed will be retrieved, processed and either stored in a subcomponent or prepared for (external) assessment. The extracted information will be prepared for assessment and provided to Clouditor or similar assessment tools. The other component's connections were omitted for simplicity. Figure 13 depicts the current envisioned architecture of the "Organizational evidence gathering and processing" component which is briefly described in the following section.

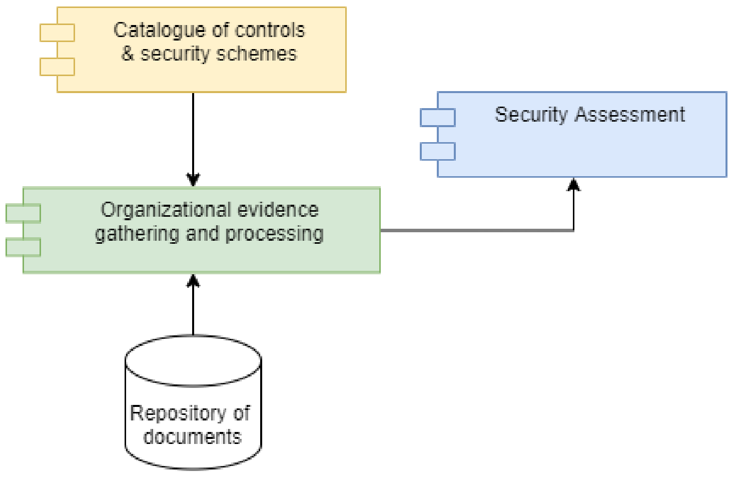

Catalogue of controls & security schemes
Security Assessment
Organizational evidence gathering and processing
Repository of documents

Figure 12. Abstract schema of the organizational evidence gathering and processing component

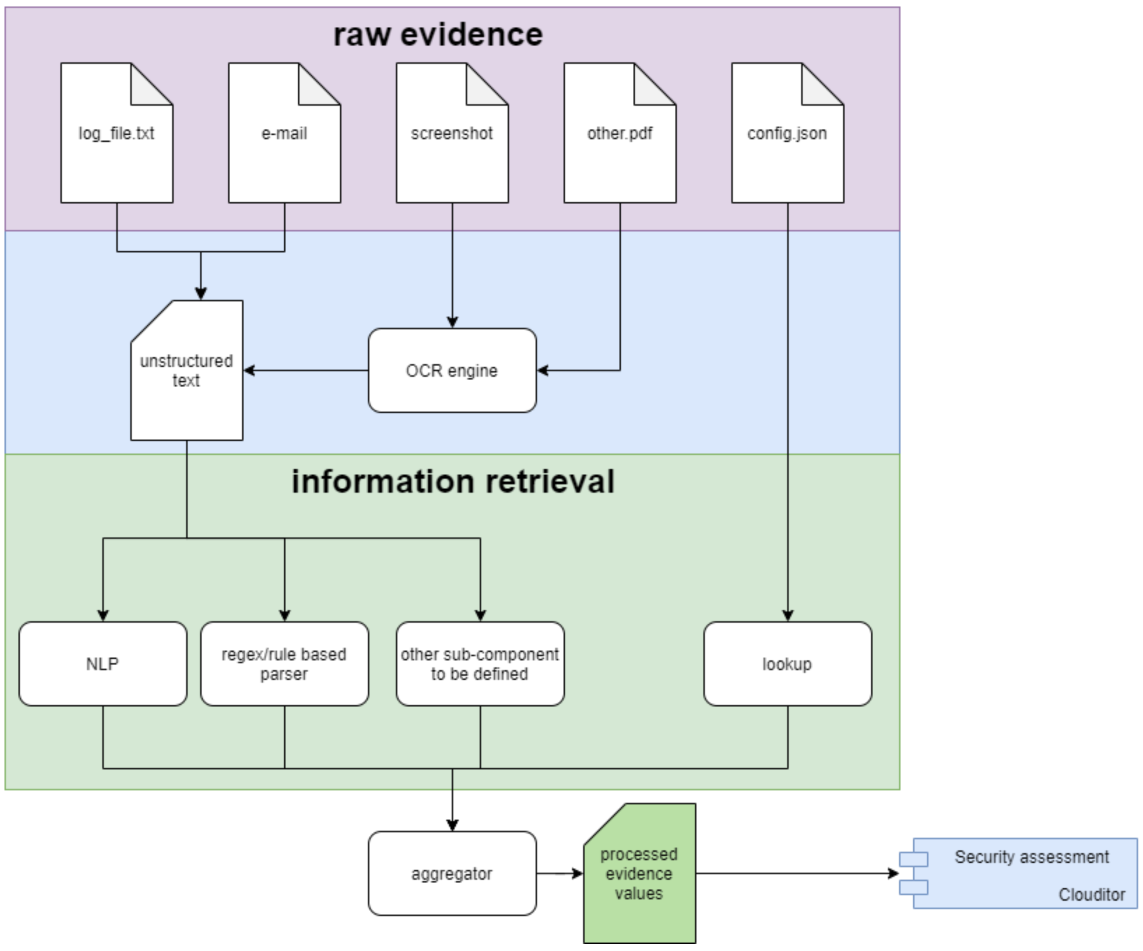


Figure 13. Prototype architecture

5.2.2 Description of components

Figure 13 depicts the prototype architecture which will be improved and detailed as the research experiment results in Task 3.4 become available, which will be reported in the next versions of this deliverable, namely D3.5 [4] and D3.6 [5].

The top purple section of the figure depicts possible input formats (e.g., from the repository of evidence documents, or direct CSP input documents). Raw evidence refers to the unprocessed evidence documents, which cannot be used for direct technical assessment. The middle section depicts a necessary transformation step to handle the data in the information retrieval sub-components. The bottom green layer depicts possible "information retrieval" sub-components still to be determined by concrete experiments and evidence data, and which will be reported in subsequent versions of this deliverable.

5.2.3 Technical specifications

The prototype will be developed in Python and will probably depend on common machine learning and data science libraries like Pandas, NumPy, scikit-learn and others like OpenCV – depending on the subtask. The component will be containerized to be able to be integrated in the MEDINA CI/CD infrastructure. If required by the subcomponents a NoSQL database like MongoDB can be used for internal document and assessment result storage. The service will provide an API supported by a Python Flask server.

6 Conclusions

In this deliverable D3.4, which is the initial output of Task 3.2, Task 3.3, and Task 3.4, we presented the technical report about the design, architecture, and current implementation states of MEDINA evidence gathering components. The components follow the overall MEDINA framework approach and are aligned with the requirements gathered in the scope of WP5. This deliverable presents the relation of the presented components with the other parts of the MEDINA framework and details the individual components' internal structure, their subcomponents, and information about their technical implementation.

The components presented in this document include three tools supporting the security assessment of cloud infrastructure (Clouditor, Wazuh, and Vulnerability Assessment Tools), a pair of tools for assessing the security and compliance of cloud application's source code (Codyze and CloudPG), and a component for the assessment of organisational measures based on analysis of CSP's documentation. At this point in the project, the components, based on some background works, already have working prototypes that can be (partially) integrated with some other MEDINA components and satisfy some of their respective requirements as expressed in D5.1 [3] as well as to comply with the needs of EUCS requirements of assurance level high, the goal of MEDINA. There is currently no implemented prototype of the component for assessment of organisational measures, but the methodology for its production, as well as the envisioned design and architecture are described. An overview of the current satisfaction of MEDINA requirements by the implemented tools is presented in Appendix A.

The components, presented in this deliverable, will be integrated into the MEDINA framework in WP5. The subsequent iterations of this report will give reports on the updated components in month 24 (D3.5 [4]) and month 30 (D3.6 [5]).

7 References

[1] European Union Agency for Cybersecurity (ENISA), “EUCS – Cloud Services Scheme (draft),” 22 December 2020. [Online]. Available: https://www.enisa.europa.eu/publications/eucs-cloud-service-scheme.

[2] MEDINA Consortium, “D3.1: Tools and techniques for the management of trustworthy evidence - V1,” 2021.

[3] MEDINA Consortium, “D5.1: MEDINA requirements, Detailed architecture, DevOps infrastructure and CI/CD and verification strategy,” 2021.

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

[5] MEDINA Consortium, “D3.6 Tools and techniques for collecting evidence of technical and organisational measures-v3,” 2023.

[6] MEDINA Consortium, “D2.3: Specification of the Cloud Security Certification Language - V1,” 2021.

[7] MEDINA Consortium;, “D2.1 – Continuously certifiable technical and organizational measures and catalogue of cloud security metrics-v1,” 2021.

[8] MEDINA Consortium;, “D4.1 Tools and Techniques for the Management and Evaluation of Cloud Security Certifications,” 2021.

[9] Cisco, “ClamAV,” [Online]. Available: https://www.clamav.net/. [Accessed October 2021].

[10] Chronicle Security, “VirusTotal,” [Online]. Available: https://www.virustotal.com/. [Accessed October 2021].

[11] Wazuh Inc., “Wazuh,” [Online]. Available: https://wazuh.com/. [Accessed October 2021].

[12] “w3af,” [Online]. Available: http://w3af.org/. [Accessed September 2021].

[13] OWASP Foundation, “OWASP Zed Attack Proxy (ZAP),” [Online]. Available: https://owasp.org/www-project-zap/. [Accessed September 2021].

[14] “Nmap,” [Online]. Available: https://nmap.org/. [Accessed September 2021].

[15] Docker, Inc., “Docker,” [Online]. Available: https://www.docker.com/. [Accessed October 2021].

[16] Rapid7, “Metasploit,” [Online]. Available: https://www.metasploit.com/. [Accessed October 2021].

[17] VMware, Inc., “RabbitMQ,” [Online]. Available: https://www.rabbitmq.com/. [Accessed October 2021].

[18] MongoDB, Inc., “MongoDB,” [Online]. Available: https://www.mongodb.com/. [Accessed October 2021].

[19] OpenStack, “OpenStack Swift (Github repository),” [Online]. Available: https://github.com/openstack/swift. [Accessed October 2021].

[20] Google LLC, “Angular,” [Online]. Available: https://angular.io/. [Accessed October 2021].

[21] Faraday Security, “Faraday (Github repository),” [Online]. Available: https://github.com/infobyte/faraday. [Accessed October 2021].

[22] HashiCorp, Inc., “Vagrant,” [Online]. Available: https://www.vagrantup.com/. [Accessed October 2021].

[23] Red Hat, Inc., “Ansible,” [Online]. Available: https://www.ansible.com/. [Accessed October 2021].

[24] CYBERWISER.eu consortium, “CYBERWISER.eu,” [Online]. Available: https://www.cyberwiser.eu/. [Accessed October 2021].

[25] Fraunhofer AISEC, “MARK (Modeling Language for Cryptography Requirements and Guidelines) GitHub page,” [Online]. Available: https://github.com/Fraunhofer-AISEC/codyze-mark-eclipse-plugin. [Accessed October 2021].

Appendix A. MEDINA requirements implementation overview

Table 4 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 D5.1 [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 D5.1. Implementation status has three possible values represented in the table by colours:

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

This table will be updated and presented in the next versions of this report for easier comparison and progress tracking.

Table 4. Overview of requirements satisfaction according to current implementation of presented tools

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	🟧			
ETM.01	Trustworthiness of evidence	🟥			
ETM.02	Transmission of evidence checksums	🟥			
Wazuh (Gathering evidence from computing resources)					
TEGT.S.08	Provision of malware, intrusion, and vulnerability detection tools		🟩		

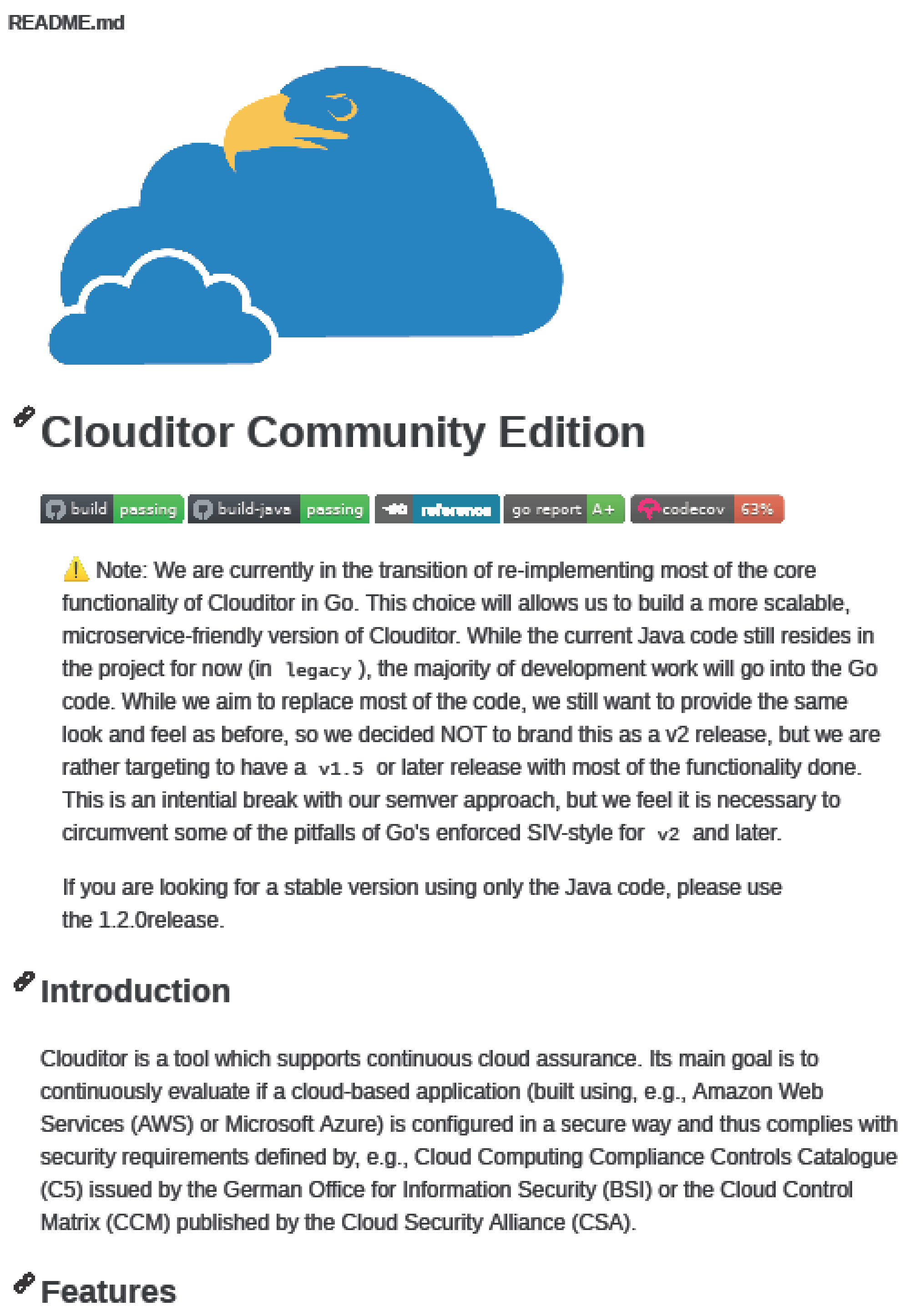
VAT (Gathering evidence from computing resources)		
TEGT.S.08	Provision of malware, intrusion, and vulnerability detection tools	🟨
Codyze + CloudPG (Gathering evidence from application source code)		
TEGT.S.02	Collect evidence from source code via CPG	🟥
TEGT.S.03	Implement information and data flow analysis	🟩
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	🟨
Organizational evidence gathering and management		
OEGM.01	Continuous collection of organizational evidence	🟥
OEGM.02	Provision to defined interfaces	🟥
OEGM.03	Usability for auditors	🟥
OEGM.04	Minimum evidence storage	🟥

In total, there are 30 requirements (if the common requirements are counted once for each component) related to the presented components. 7 (23%) of them are currently marked as fully implemented, 11 (37%) as partly implemented, and 12 (40%) as not implemented. The basic statistic of requirement coverage by component is presented in Table 5.

Table 5. Requirements satisfied by tool

Tool	Number of requirements	Fully implemented	Partially implemented	Not implemented
Clouditor	11	5	2	4
Wazuh	3	1	2	0
VAT	3	0	2	1
Codyze + CloudPG	9	1	5	3
Organisational evidence gathering	4	0	0	4

Appendix B. Clouditor README file



README.md

Clouditor Community Edition

build passing | build-java passing | reference | go report A+ | codecov 63%

> ⚠️ Note: We are currently in the transition of re-implementing most of the core functionality of Clouditor in Go. This choice will allows us to build a more scalable, microservice-friendly version of Clouditor. While the current Java code still resides in the project for now (in `legacy`), the majority of development work will go into the Go code. While we aim to replace most of the code, we still want to provide the same look and feel as before, so we decided NOT to brand this as a v2 release, but we are rather targeting to have a `v1.5` or later release with most of the functionality done. This is an intential break with our semver approach, but we feel it is necessary to circumvent some of the pitfalls of Go's enforced SIV-style for `v2` and later.
>
> If you are looking for a stable version using only the Java code, please use the 1.2.0release.

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

Usage

To run the Clouditor in a demo-like mode, with no persisted database:

```
docker run -p 9999:9999 clouditor/clouditor
```

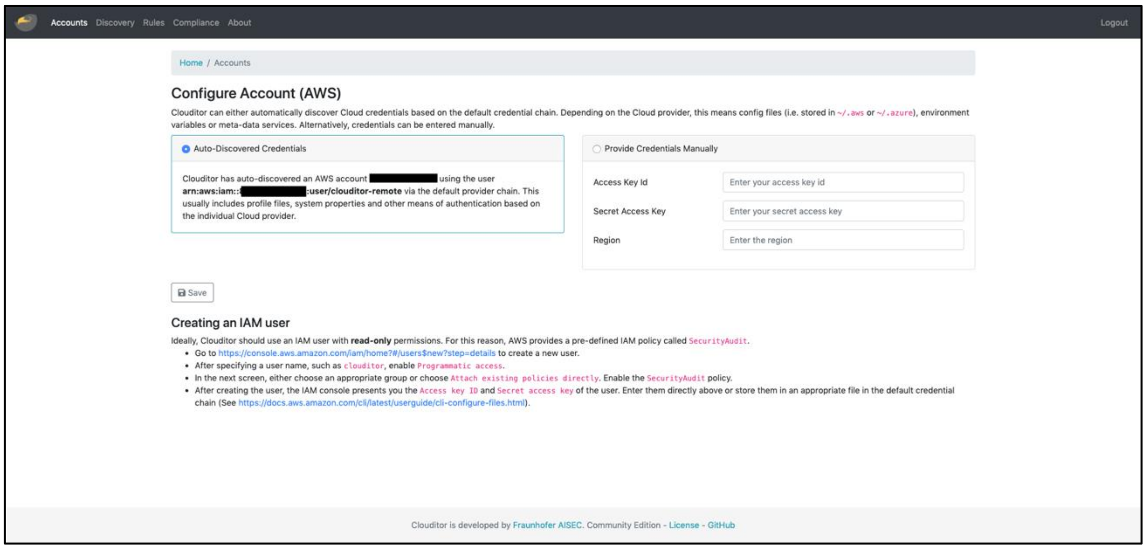
To enable auto-discovery for AWS or Azure credentials stored in your home folder, you can use:

```
docker run -v $HOME/.aws:/root/.aws -v $HOME/.azure:/root/.azure -p
9999:9999 clouditor/clouditor
```

Then open a web browser at http://localhost:9999. Login with user `clouditor` and the default password `clouditor` .

Screenshots

Configuring an account



Configure Account (AWS)

Clouditor can either automatically discover Cloud credentials based on the default credential chain. Depending on the Cloud provider, this means config files (i.e. stored in `~/.aws` or `~/.azure`), environment variables or meta-data services. Alternatively, credentials can be entered manually.

● Auto-Discovered Credentials

Clouditor has auto-discovered an AWS account [redacted] using the user **arn:aws:iam::[redacted]:user/clouditor-remote** via the default provider chain. This usually includes profile files, system properties and other means of authentication based on the individual Cloud provider.

○ Provide Credentials Manually

Access Key Id	Enter your access key id
Secret Access Key	Enter your secret access key
Region	Enter the region

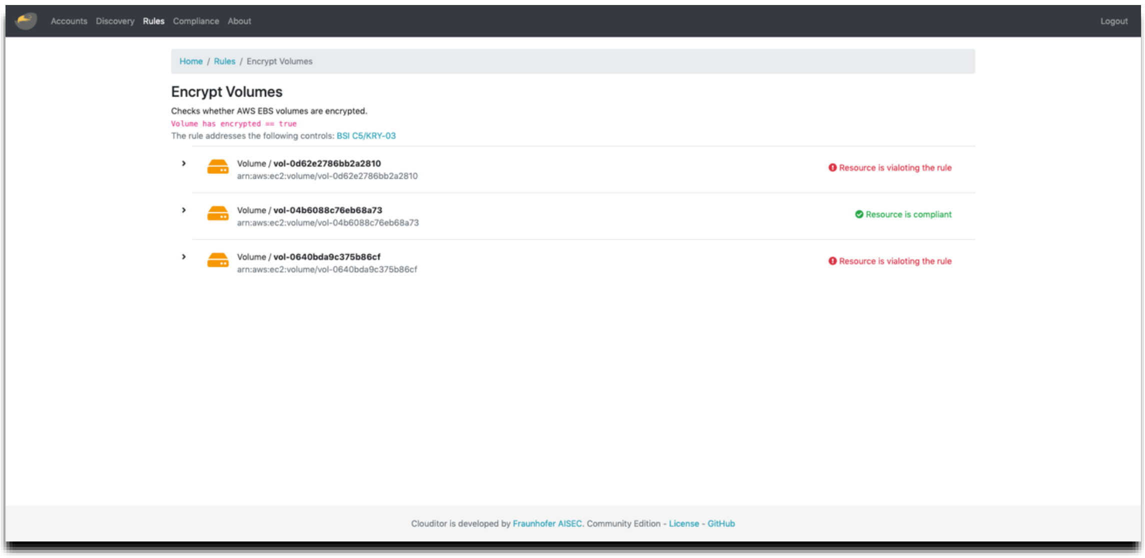
Save

Creating an IAM user

Ideally, Clouditor should use an IAM user with **read-only** permissions. For this reason, AWS provides a pre-defined IAM policy called `SecurityAudit`.

- Go to https://console.aws.amazon.com/iam/home?#/users$new?step=details to create a new user.
- After specifying a user name, such as `clouditor`, enable `Programmatic access`.
- In the next screen, either choose an appropriate group or choose `Attach existing policies directly`. Enable the `SecurityAudit` policy.
- After creating the user, the IAM console presents you the `Access key ID` and `Secret access key` of the user. Enter them directly above or store them in an appropriate file in the default credential chain (See https://docs.aws.amazon.com/cli/latest/userguide/cli-configure-files.html).

Clouditor is developed by Fraunhofer AISEC. Community Edition - License - GitHub


Accounts Discovery Rules Compliance About
Logout
Home / Rules / Encrypt Volumes

Encrypt Volumes

Checks whether AWS EBS volumes are encrypted.

`Volume has encrypted == true`

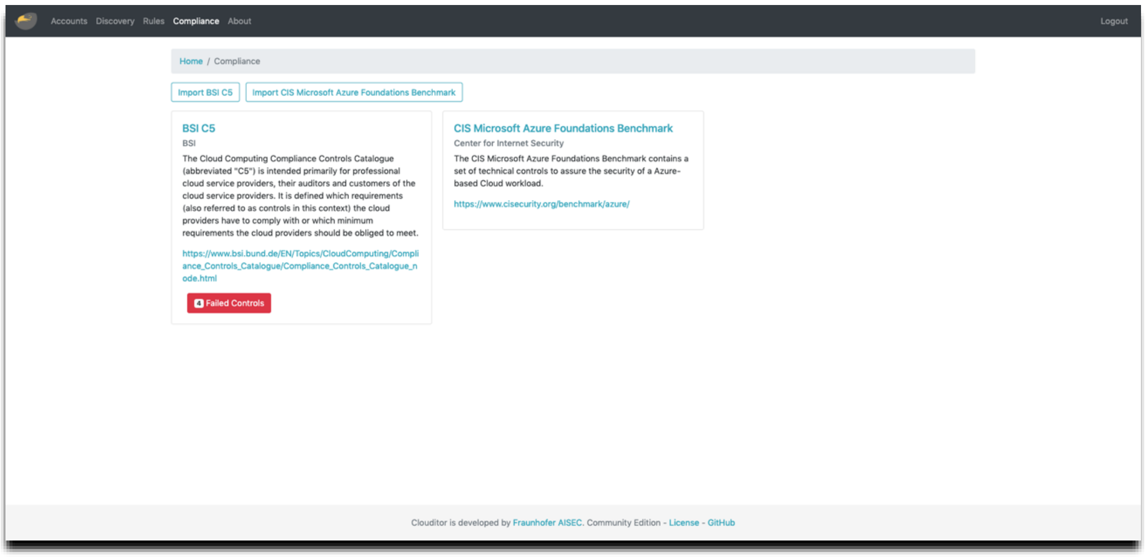
The rule addresses the following controls: BSI C5/KRY-03

Volume / **vol-0d62e2786bb2a2810**
arn:aws:ec2:volume/vol-0d62e2786bb2a2810
Resource is vialoting the rule

Volume / **vol-04b6088c76eb68a73**
arn:aws:ec2:volume/vol-04b6088c76eb68a73
Resource is compliant

Volume / **vol-0640bda9c375b86cf**
arn:aws:ec2:volume/vol-0640bda9c375b86cf
Resource is vialoting the rule

Clouditor is developed by Fraunhofer AISEC. Community Edition - License - GitHub


Accounts Discovery Rules Compliance About
Logout
Home / Compliance
Import BSI C5
Import CIS Microsoft Azure Foundations Benchmark

BSI C5

BSI

The Cloud Computing Compliance Controls Catalogue (abbreviated "C5") is intended primarily for professional cloud service providers, their auditors and customers of the cloud service providers. It is defined which requirements (also referred to as controls in this context) the cloud providers have to comply with or which minimum requirements the cloud providers should be obliged to meet.

https://www.bsi.bund.de/EN/Topics/CloudComputing/Compliance_Controls_Catalogue/Compliance_Controls_Catalogue_node.html

4 Failed Controls

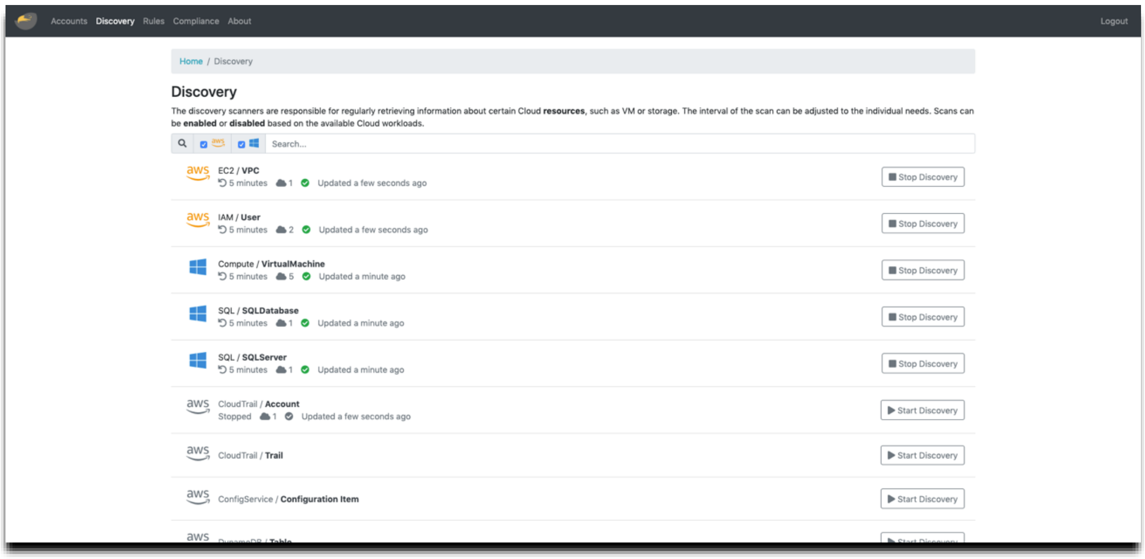
CIS Microsoft Azure Foundations Benchmark

Center for Internet Security

The CIS Microsoft Azure Foundations Benchmark contains a set of technical controls to assure the security of a Azure-based Cloud workload.

https://www.cisecurity.org/benchmark/azure/

Clouditor is developed by Fraunhofer AISEC. Community Edition - License - GitHub



Discovery

The discovery scanners are responsible for regularly retrieving information about certain Cloud **resources**, such as VM or storage. The interval of the scan can be adjusted to the individual needs. Scans can be **enabled** or **disabled** based on the available Cloud workloads.

EC2 / **VPC**
5 minutes 1 Updated a few seconds ago — Stop Discovery

IAM / **User**
5 minutes 2 Updated a few seconds ago — Stop Discovery

Compute / **VirtualMachine**
5 minutes 5 Updated a minute ago — Stop Discovery

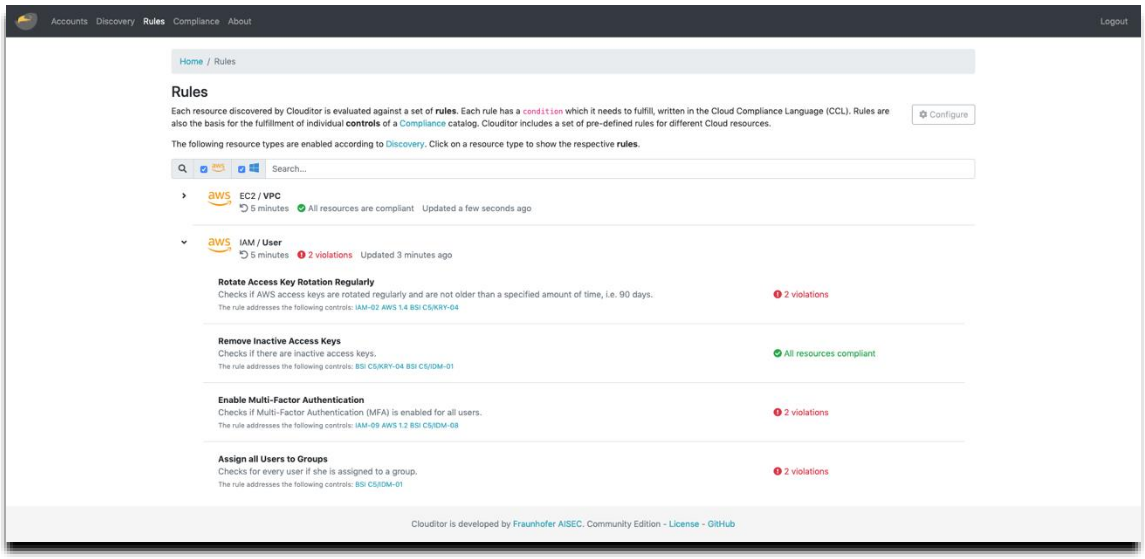
SQL / **SQLDatabase**
5 minutes 1 Updated a minute ago — Stop Discovery

SQL / **SQLServer**
5 minutes 1 Updated a minute ago — Stop Discovery

CloudTrail / **Account**
Stopped 1 Updated a few seconds ago — Start Discovery

CloudTrail / **Trail** — Start Discovery

ConfigService / **Configuration Item** — Start Discovery



Rules

Each resource discovered by Clouditor is evaluated against a set of **rules**. Each rule has a `condition` which it needs to fulfill, written in the Cloud Compliance Language (CCL). Rules are also the basis for the fulfillment of individual **controls** of a Compliance catalog. Clouditor includes a set of pre-defined rules for different Cloud resources.

The following resource types are enabled according to Discovery. Click on a resource type to show the respective **rules**.

EC2 / **VPC**
5 minutes ✓ All resources are compliant Updated a few seconds ago

IAM / **User**
5 minutes ● 2 violations Updated 3 minutes ago

Rotate Access Key Rotation Regularly
Checks if AWS access keys are rotated regularly and are not older than a specified amount of time, i.e. 90 days. — ● 2 violations
The rule addresses the following controls: IAM-02 AWS 1.4 BSI C5/KRY-04

Remove Inactive Access Keys
Checks if there are inactive access keys. — ✓ All resources compliant
The rule addresses the following controls: BSI C5/KRY-04 BSI C5/IDM-01

Enable Multi-Factor Authentication
Checks if Multi-Factor Authentication (MFA) is enabled for all users. — ● 2 violations
The rule addresses the following controls: IAM-09 AWS 1.2 BSI C5/IDM-08

Assign all Users to Groups
Checks for every user if she is assigned to a group. — ● 2 violations
The rule addresses the following controls: BSI C5/IDM-01

Clouditor is developed by Fraunhofer AISEC. Community Edition - License - GitHub

Build (gradle)

To build the Clouditor, you can use the following gradle commands:

```
./gradlew clean build
```

Build (Docker)

To build all necessary docker images, run the following command:

```
./gradlew docker
```

Build (Go components) - Experimental

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/googleapis/gnostic/apps/protoc-gen-openapi
```

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

```
go generate ./...
go build ./...
```

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

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

Alternatively, be sure to start a postgre 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`.