Deliverable D3.1

CDT design and abstractions

Editor	D. Canavese (CNR)
Contributors	CNR, ONE, PLX, POLITO
Version	1.0
Date	26th February 2026
Distribution	PUBLIC (PU)
Classification	UNCLASSIFIED (U)

This project has received funding from the European Union's Horizon Europe Research and Innovation programme under grant agreement No. 101168144. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Cybersecurity Competence Centre. Neither the European Union nor the granting authority can be held responsible for them.

Authors

CNR	CONSIGLIO NAZIONALE DELLE RICERCHE
D. Canavese	
ONE	ONESOURCE
J. Proença	
PLX	PLAIXUS
M. Karamousadakis	
POLITO	POLITECNICO DI TORINO
D. Bringhenti, G. Bachiorrini	

Reviewers

SPHYNX	SPHYNX
L. Koumakis	
UBI	UBITECH
S. Menesidou	

Copyright and Disclaimer

Funded by
The European Union

Version History

Rev. N	Description	Author	Date
0.1	Initial version of the table of contents	D. Canavese (CNR)	15/12/2025
0.2	Completed chapters 2 to 6 and the list of acronyms	D. Canavese (CNR), J. Proença (ONE), M. Karamousadakis (PLX), D. Bringhenti, G. Bachiorrini (POLITO)	04/02/2026
0.3	Completed executive summary, introduction and conclusion	D. Canavese (CNR)	06/02/2026
0.4	Integrated the reviewers' recommendations	D. Canavese (CNR)	24/02/2026
1.0	Final formatting	M. Repetto (CNR)	26/02/2026

List of Acronyms

Acronym	Meaning
API	Application Programming Interface
CBOR	Concise Binary Object Representation
CIDR	Classless Inter-Domain Routing
CLI	Command Line Interface
CTI	Cyber Threat Intelligence
CTXD	ConTeXt Discovery
CVE	Common Vulnerabilities and Exposures
CVSS	Common Vulnerability Scoring System
DHP	Data Handling Pipeline
GUI	Graphical User Interface
HTTP	HyperText Transfer Protocol
ID	IDentifier
IoT	Internet of Things
IP	Internet Protocol
IPFIX	IP Flow Information Export
JSON	JavaScript Object Notation
MISP	Malware Information Sharing Platform
MQTT	Message Queuing Telemetry Transport
NSG	Network Security Group
OpenC2	Open Command and Control
OS	Operating System
RCLI	Remote Command Line Interface
REST	REpresentational State Transfer

SBoM	Software Bill of Materials
SCG	Service Context Graph
SCTP	Stream Control Transmission Protocol
SDK	Software Development Kit
SLPF	StateLess Packet Filter
SQL	Structured Query Language
STIX	Structured Threat Information eXpression
TAMELESS	Threat & Attack ModEL Smart System
TCP	Transmission Control Protocol
TLP	Traffic Light Protocol
UDP	User Datagram Protocol
VEC	Visibility, Enrichment, and Contextualization
VM	Virtual Machine
YAML	YAML Ain't Markup Language
XBoM	eXtended Bill of Materials
XML	eXtensible Markup Language

Executive Summary (CNR)

This deliverable reports on the work carried out under *Work Package 3 "Cybersecurity Digital Twin"* of the MIRANDA project during the first half of the project. WP3 focuses on the design and implementation of core services that enable context awareness, monitoring, enforcement, attack and threat modelling, and data handling within the MIRANDA platform. The activities described in this document have been conducted as part of Tasks 3.1, 3.2, and 3.3, which are planned to conclude at Month 18 (M18). As such, this deliverable provides an intermediate consolidation of the results achieved so far.

Task 3.1 "Service Context Graph," addressed the development of services for context discovery, monitoring, and enforcement based on the OpenC2 command-and-control protocol. Context discovery capabilities support the automated identification of infrastructure components, their characteristics, and their interconnections, providing a shared and up-to-date view of the operational environment. Monitoring services enable visibility into system and network behavior through network flow monitoring and file and log collection, supporting both operational and security-related analysis. Enforcement services provide standardized mechanisms for controlling security functions, including stateless packet filtering and remote command-line execution, across heterogeneous environments.

Task 3.2 "Attack and Threat Modelling" focused on attack and threat modelling, supported by the integration of cyber threat intelligence through dedicated threat feeds. Threat feeds aggregate and normalize information from heterogeneous sources, enabling the ingestion of vulnerability and threat data in a uniform representation. The attack & threat modelling component instead supports structured reasoning on vulnerabilities, threats, and their propagation across interconnected services. The resulting framework enables the derivation of attack paths and the analysis of multi-step attack scenarios, contributing to proactive and predictive security analysis within the MIRANDA platform.

Task 3.3 "Visibility, Enrichment, Contextualization" defined a data handling pipeline capable of ingesting, normalizing, and processing heterogeneous security telemetry. This layer provides a flexible abstraction for managing data-processing workflows and supports the integration of diverse data sources, thereby facilitating the delivery of enriched and contextualized information to higher-level MIRANDA components.

Overall, the work presented in this deliverable establishes the main WP3 components from Tasks 3.1, 3.2, and 3.3, and their design abstractions required for the subsequent integration, validation, and use-case-driven evaluation phases of the project. The results achieved during the first half of the project provide a solid basis for completing the remaining developments and validating them within the integrated MIRANDA platform.

Table of Contents

List of Figures

List of Tables

1. INTRODUCTION

This deliverable presents the work carried out in *Work Package 3 "Cybersecurity Digital Twin" (WP3)* of the MIRANDA project during the first half of the project. WP3 focuses on the design and implementation of the core services enabling context awareness, monitoring, enforcement, and cyber threat reasoning within the MIRANDA platform. Figure 1 shows an overview of MIRANDA's architecture. The components described in this document have a solid red border, while their internal data models have a red dashed border.

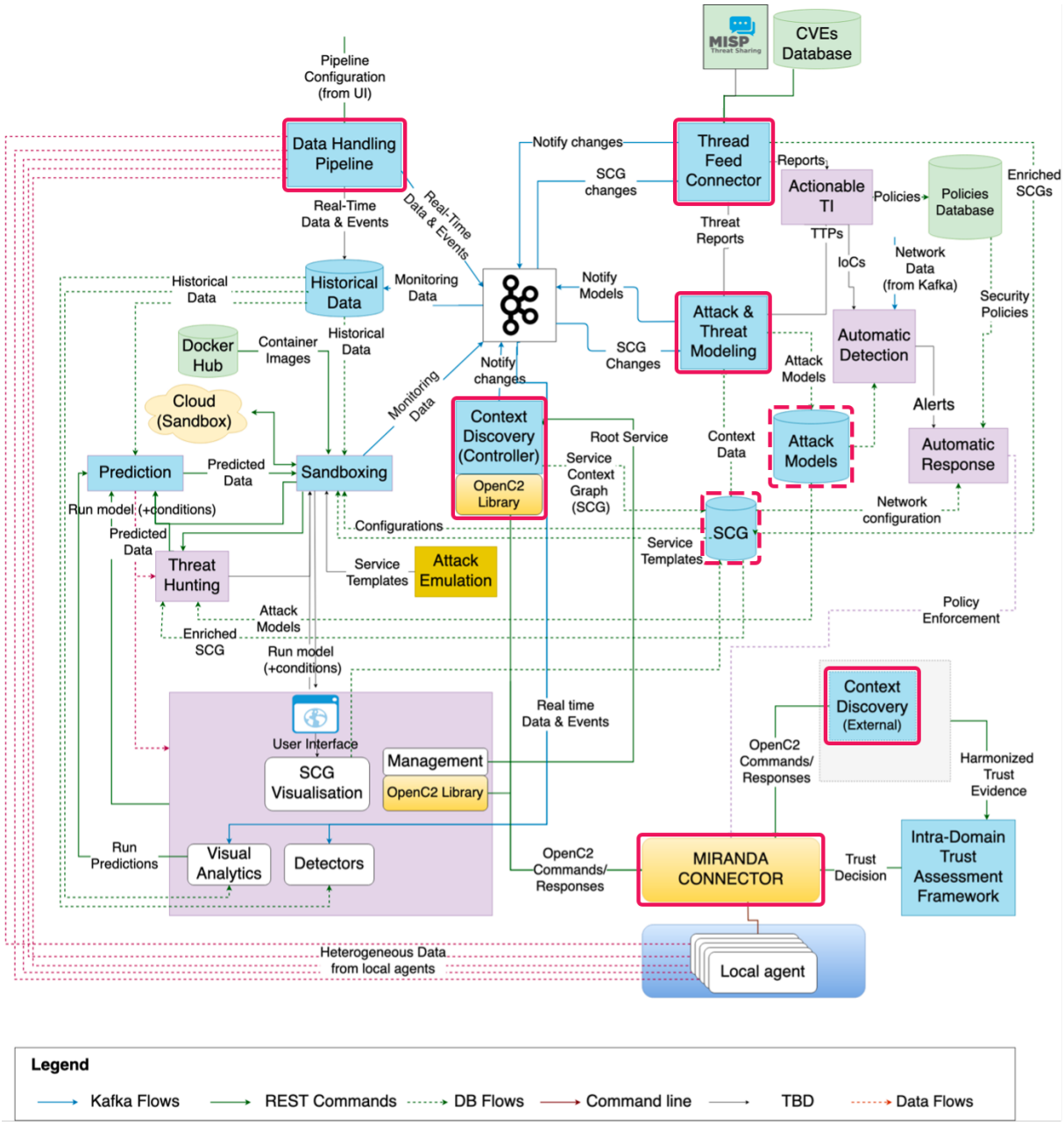

Figure 1. General architecture of MIRANDA.

This document reports on the activities performed in *Tasks* 3.1, 3.2, and 3.3.*Task 3.1 "Service Context Graphs"* addresses the development of service-level capabilities for context discovery,

monitoring, and enforcement, providing standardized abstractions and control mechanisms for heterogeneous infrastructures through the Context Discovery components, the MIRANDA Connector, and its actuators. *Task 3.2 "Attack and Threat Modelling"* focuses on attack representation, leveraging cyber threat intelligence and contextual information to derive attack paths and support proactive security analysis via the Attack & Threat Modelling and Threat Feeds modules. Finally, *Task 3.3, "Visibility, Enrichment, Contextualization"* is devoted to the Data Handling Pipeline module that enables the ingestion, normalization, and processing of heterogeneous security telemetry across the entire computing continuum. These tasks are scheduled to conclude at Month 18. Accordingly, this deliverable reports the design choices, architectures, and implementations developed during the first half of the project, providing an intermediate consolidation of the results achieved so far.

Table 1 summarizes the components described in this deliverable, their tasks, their locations in this document, and their requirements (described in *Deliverable D2.1 "Concept, State- of-the-Art, Requirements, and System Architecture"*).

Table 1. Summary of the components described in the deliverable.

Component	Task	Sections	Satisfied requirements
Context discovery	T3.1	2	• FR-A-1.8 – Displaying service context graphs • FR-A-1.10 – Navigating service context graphs • FR-B-1.6 – Service types • FR-B-1.7 – Service relationships • FR-B-1.11 – Context API • FR-B-1.13 – Context discovery implementation • NON-FR-B-1.16 – Time to discover the context
MIRANDA connector	T3.1	3 and 4	• FR-A-1.11 – Controlling security functions • FR-A-1.27 – Packet filtering • FR-B-1.2 – Interface to security functions • FR-B-1.8 – Security properties • NON-FR-B-1.24 – Latency of commands to security functions
Threat feeds	T3.2	5.1	• FR-B-1.46 – System context and data export/import
Attack modelling	T3.2	5.2	• FR-A-1.9 – Displaying vulnerabilities and threats
Data handling pipeline	T3.3	6	• NON-FR-B-1.35 – Adaptive data handling pipelines

This deliverable is complemented by *Deliverable D2.3, "Integrated MIRANDA platform"*, which serves as a twin report. While this document focuses on the design abstractions and functional capabilities of WP3 components, deliverable D2.3 provides more technical and integration-oriented details, including a detailed description of the satisfied functional and non-functional

requirements, module interfaces, inter-component communication mechanisms, and validation activities within the integrated MIRANDA platform.

The document is structured as follows. After this introduction, the core WP3 components are presented grouped by categories. The components developed in Task 3.1 are discussed in Chapters 2, 3, and 4. They report respectively the context discovery mechanisms, followed by the monitoring and enforcement services, managed by the MIRANDA connector. The modules developed in Task 3.2 are instead presented in Chapter 5 and discuss the threat feeds and attack and threat modelling tools. Chapter 6 instead details the data handling pipeline, the component implemented in Task 3. Finally, Chapter 7 presents the conclusions and final remarks.

2. CONTEXT DISCOVERY

This chapter introduces the Con*TeXt Discovery* (CTXD) service, a system that, once deployed in a virtualized network, can automatically analyze the infrastructure to discover available network services, their features (e.g., the operating system of a virtual machine), and their interconnections. This information, the *context*, is then stored in a central location accessible by other services as well. This data is crucial to the proper operation of many other MIRANDA systems, such as threat-hunting and attack & threat modelling services. More information about how these other components are communicating with the CTXD service is described in Deliverable 2.3 Integrated MIRANDA platform.

The standard OpenC2 language does not natively support the discovery of such contextual information, as it lacks targets that can expose service-level attributes such as operating systems, hostnames, or connectivity details. The CTXD bridges this gap, enabling systematic, automated context acquisition as a foundation for informed security orchestration, a key pillar in the MIRANDA platform.

Figure 2 reports the high-level architecture of this component.

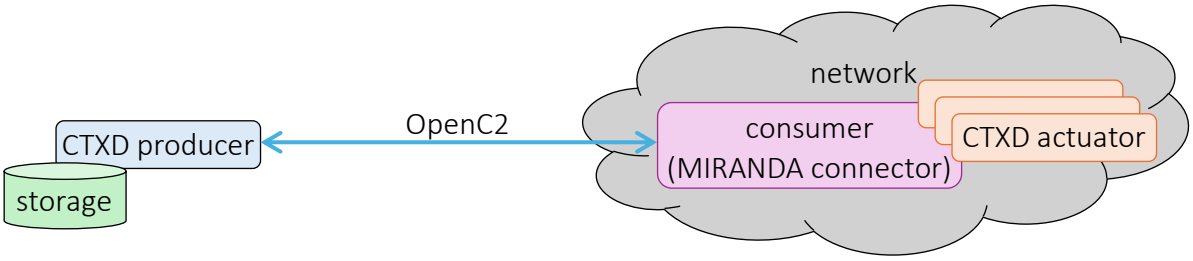

Figure 2. General architecture of the CTXD service.

The target network hosts a consumer, a component that acts as an OpenC2 server. On the one side, the job of the consumer is to leverage one or more CTXD OpenC2 actuators, the modules responsible for discovering services and their interconnections in the target network. On the other side, a CTXD producer (i.e., a client) can issue OpenC2 commands to query the appropriate CTXD actuator via the consumer and, optionally, store the obtained context.

2.1 Data model

Figure 3 reports the data model used by the context discovery module. Since its goal is to describe a network, what is hosting and how the various nodes are interconnected, it is centered around two key concepts: services and links. A *service* is any digital resource that can be identified and connected to other resources. This includes software, applications, repositories, infrastructure, networks, devices, and anything that is potentially vulnerable to cyberattacks. Its description encompasses details such as location, subservices, owners, and a release version. The most relevant information about the service is embedded within the *service type*, as these details are specific to each service type.

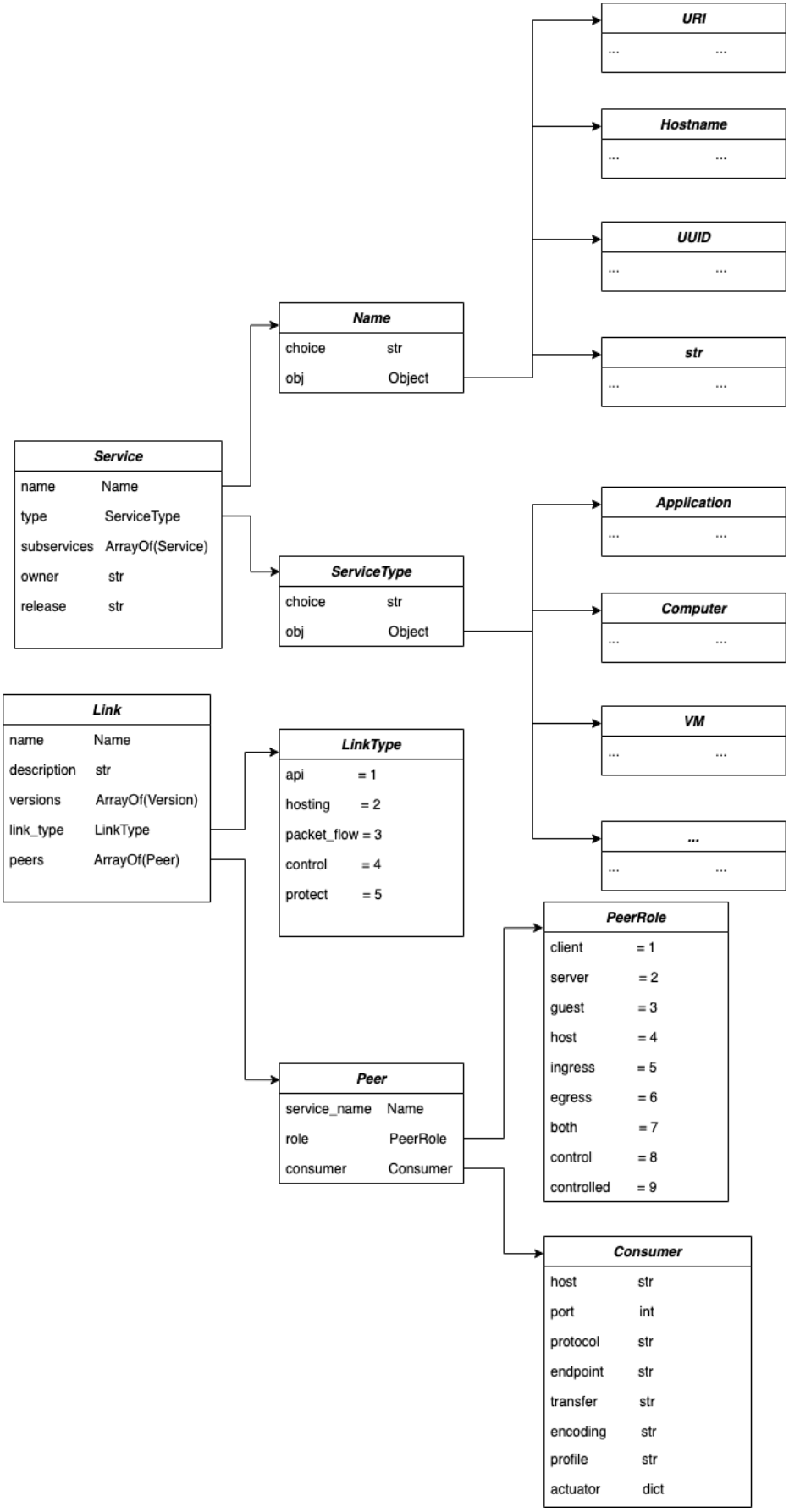

Figure 3. Context discovery data model.

Currently, the CTXD module supports the following service types:

- a software application of any kind, including an operating system;
- a computer, that is a collection of an OS and its software — it can be hosted on either a physical device or a virtualized one;
- a virtual machine, a container, or a Kubernetes pod;
- an IoT device;
- a web application;
- a cloud environment or a physical or virtual network.

The second key concept is a *link*, which represents a relationship between two or more services. It defines the relationship between one name, which is always described by the current actuator, and one or more peers. These *peers* might be described by the same or a different actuator. If described by a different actuator, the peer object may include a consumer object indicating where the entire service description can be retrieved. This object will be absent if the consumer does not know where to find the description or if no CTXD consumer exists to describe the service, making it a hidden node in the chain. It is the responsibility of the producer performing discovery to query another consumer to build the complete service chain, as no caching is involved. Like the service object, the descriptive elements of a link are found in its type. Currently, the CTXD system supports the following link types:

- an API link indicating that two services are using a (web) API to communicate;
- a hosting link representing that a service is hosting another one (e.g., a computer hosting an OpenStack environment);
- a control link indicating that a service is controlling another one (e.g., Kubernetes is controlling its pods);
- a protect link representing that a service is protecting another one (e.g., a firewall is protecting a network service);
- a packet flow link indicating a generic (non-API) exchange of data packets.

More information about the CTXD data model can be found on the otupy documentation page.[1]

2.2 Actuators

The CTXD actuators are fully compliant with the OpenC2 standard and leverage the otupy library.[2] They represent the functional interfaces that allow the retrieval of detailed contextual information about services, their execution environments, security functions, and inter-service relationships, supporting a recursive discovery process across heterogeneous infrastructures.

All the CTDX actuators implement the same OpenC2 profile, named `x-ctxd`, specifying the common capabilities of all the context discovery actuators, which are:

- all the CTXD actuators support the `query` command with two Boolean arguments:
  - `name_only`: when set, retrieves only the service and link names, and otherwise the CTXD actuator returns the full details — this option is useful mostly when debugging, since returning the full context can be very time consuming;

[1] https://otupy.readthedocs.io/en/latest/ctxd/ctxd.html
[2] https://github.com/mattereppe/otupy

    - `cached`: this option toggles returning the cache context results — this parameter can speed up the execution of the command, however, any recent change to the target network might not be returned;
- the result of the query is the *context*, an object containing four fields:
    - `services`: an array of the services in the target network containing full details;
    - `links`: an array of the links connecting the services;
    - `service_names`: an array containing the names of the services;
    - `link_names`: an array containing the names of the links.

The CTXD module offers five actuators that support different virtualization technologies, as shown in Table 2.

Table 2. List of the supported CTXD actuators.

CMS	Discovered services	Status
OpenStack	OpenStack root services (e.g., Nova), hypervisors, VMs, SLPF firewalls	Ready
Kubernetes	Kubernetes root services, containers, pods, namespaces, nodes, SLPF firewalls	Ready
Docker	Docker root services, VMs, containers	Under revision
Proxmox	Proxmox root services, nodes, VMs, containers	Under revision
Azure	Azure root services, nodes, Kubernetes namespaces, Kubernetes pods, SLPF firewalls	Ongoing

A central feature of all the CTXD actuators is their support for recursive service discovery. By collecting and exposing information about how a service connects to its peers, the CTXD module enables the system to progressively explore the environment's topology, starting from a single known service. Each discovered service can, in turn, be queried for its own attributes and peer relationships, yielding a directed graph that captures both the system's components and their interactions.

2.3 Producer-consumer interaction

The CTXD actuators are fully compliant with the OpenC2 standard and can thus be used with any custom consumer-type consumer; however, in the MIRANDA project, we developed a generic consumer, namely the *MIRANDA connector*, which can expose any actuator via an OpenC2 server using HTTP or MQTT as the underlying transport protocol, encapsulating the OpenC2 commands and responses, which can be encoded in JSON, XML, YAML, or CBOR format. More information about the MIRANDA connector can be found in Deliverable 2.3 Integrated MIRANDA platform.

On the other side, a CTXD producer (i.e., a client) can issue OpenC2 commands to query the appropriate CTXD actuator via a consumer. The standard CTXD producer can be executed in two different ways:

- As a standalone CLI application that retrieves the context from a CTXD actuator and writes the results to a storage space once.
- As a web service (implemented via Flask[3]) that exposes several web APIs — these methods can be used to manage how the producer can keep continuously updating the stored context when the target network changes.

The context can be:

- Stored in a MongoDB collection containing multiple serialized objects;
- Sent to a Kafka topic — this is particularly useful when multiple services want to ingest the context and be notified when it is updated;
- Saved to a file containing JSON data.

The context discovery algorithm follows the following recursive exploration strategy, implemented by a Producer and one or more Consumers:

1. *Initial query*. The Producer sends a `query` command to a known Consumer, specifying what CTXD Actuator to use. The command may optionally restrict the scope to specific services or links, or request only symbolic information (names) to reduce response size.
2. *Local context collection*. Upon receiving the command, the Consumer, via the chosen Actuator, gathers information about all active services under its responsibility, their attributes and service types, the links connecting them to other services, and the OpenC2 endpoints and profiles available for interaction. The Consumer then returns this information in the OpenC2 response to the Producer, encoded according to the CTXD data model.
3. *Peer identification*. From the response, the Producer extracts the connection details (transfer protocol, encoding, endpoint) of the service peer and links identified by the CTXD actuator. This information enables the Producer to establish communication with previously unknown network services.
4. *Recursive expansion*. The Producer repeats the query context operation for each newly discovered peer. This process continues iteratively, allowing the Producer to traverse the infrastructure and build a global view of the system incrementally.
5. *Continuous Update*. The collected information is aggregated into a directed graph representation of the infrastructure. Because CTXD queries can be reissued at runtime, the algorithm supports dynamic environments and can detect changes such as service failures, removals, or topology modifications.

2.4 XBOM support

As part of the project activities, the CTXD service is continually extended and updated to enhance its ability to provide comprehensive, security-relevant contextual awareness of complex digital infrastructures. The original CTXD profile focuses on the discovery of services, their execution environment, security functions, and inter-service relationships, enabling the

[3] https://flask.palletsprojects.com/en/stable/

automated construction of a global system view through recursive OpenC2 queries. Within the project, CTXD is evolving from a topology- and service-oriented discovery mechanism into a richer context provider, capable of exposing also the software and hardware composition, and their dependencies, leveraging the concept of XBOM.

MIRANDA will introduce the support for Software Bill of Materials (SBOM) and eXtended Bill of Materials (XBOM) within the CTXD framework, leveraging the CycloneDX[4] standard as a common representation format. By integrating CycloneDX-based XBOM data into CTXD responses, the Consumer can expose detailed information about software components, dependencies, versions, and associated metadata for each discovered service. This extension will enable the Producer to correlate topological context with software supply chain visibility, better supporting the MIRANDA use cases and their security operations, such as vulnerability impact analysis, and automated security orchestration.

[4] https://cyclonedx.org/

3. MONITORING SERVICES

Task 3.1 has developed several monitoring services that enable visibility into an infrastructure's behaviour by combining insights on communication patterns with detailed event-level information. Together, they support the detection of anomalies, performance issues, and security-relevant events, as well as post-event analysis. This section describes the inner workings of both the network flow monitoring and log collection components.

3.1 Network flow monitoring

Network flow monitoring is a key capability for achieving scalable, continuous visibility across networked systems. By aggregating packets into flows and summarizing communication characteristics, flow-based monitoring provides a compact yet expressive representation of network activity, suitable for both operational monitoring and security analysis. Compared to packet-level inspection, this approach significantly reduces data volume and processing overhead, while still preserving essential information about traffic patterns, endpoints, and protocol usage.

To this end, the network flow monitoring service uses a structured control model based on explicit lifecycle management of monitoring sessions. Each session represents a self-contained monitoring configuration, defined by parameters such as the set of monitored interfaces, the flow export format, exporter behaviour, and destination collectors. Sessions can be instantiated, queried, and terminated independently, enabling fine-grained control and parallel monitoring activities tailored to different use cases.

3.1.1. Profile

The Network Flow Monitoring Profile defines the standardized control semantics required to configure, manage, and observe network flow monitoring functions in a uniform and interoperable manner. It formalizes how external controllers can interact with flow-monitoring capabilities, regardless of the specific technologies or implementations used at the execution layer.

From an interoperability perspective, the profile decouples the control plane from backend-specific details. While different actuators may use different flow-monitoring technologies, the profile ensures that control commands and responses follow a consistent structure and semantics.

At its core, the profile provides a structured abstraction for the lifecycle management of *flow monitoring sessions*. A monitoring session represents a logically independent instance of a flow collection and export, characterized by a coherent set of configuration parameters and an associated runtime state. Through the profile, controllers can explicitly request the creation, activation, termination, and inspection of such sessions, enabling dynamic and adaptive monitoring strategies.

The profile defines a constrained, well-scoped set of actions that enables managing each monitoring session individually, providing fine-grained control. Table 3 lists the supported actions and their description.

Table 3. List of the supported network flow monitoring actions.

Action	Description
query	request the supported capabilities
start	begin a flow monitoring session
stop	stop a flow monitoring session

The `query` action allows to retrieve the supported features of a specific. Controllers can then ask a monitoring component to determine supported flow formats, export protocols, configurable information elements, and interface types.

The `start` and `stop` actions instead allow you to begin or terminate a monitoring session on a specific network interface; additional filters (e.g., on IP addresses or port numbers) can be used to restrict monitoring to specific traffic types.

The profile also enforces semantic validation of configuration parameters. Monitoring session requests are checked for internal consistency, completeness, and compatibility with the underlying actuator's supported capabilities. Invalid or unsupported configurations are rejected with explicit feedback, preventing partial or inconsistent deployments and improving overall system robustness.

3.1.2. Actuators

The Network Flow Monitoring Profile is supported by three actuators, listed in Table 4.

Table 4. List of the supported network flow monitoring actuators.

Tool	Storage	Status
Packetbeat	Elasticsearch, Logstash, local files	Ready
nprobe	any NetFlow/IPFIX collector, local files	Ready
fprobe	any NetFlow/IPFIX collector	Ongoing

All the actuators use an embedded SQLite database as a lightweight, reliable persistence layer to maintain the monitoring system's runtime state. The database stores structured information related to configured flow monitors, including their identifiers, associated exporters and collectors, selected interfaces, flow formats, and current operational status. In addition, it

tracks the lifecycle of active monitoring sessions, enabling consistent management of start, stop, and query operations across restarts and failures.

An actuator manages one or more flow monitoring sessions, each corresponding to an active or configured instance of flow collection and export. For each session, the actuator is responsible for instantiating the required monitoring processes, applying the specified configuration parameters, supervising execution, and handling controlled termination. The actuator maintains a runtime representation of each session, which is kept up to date in the SQLite database.

The Packetbeat[5] actuator operates as a passive network sensor that observes traffic on selected network interfaces and extracts flow-level information without interfering with packet forwarding or application behaviour. It supports generating NetFlow/IPFIX-compatible records by aggregating packets into flows based on configurable keys, such as source and destination addresses, transport protocols, and ports. Primarily, Packetbeat can export data to Elasticsearch[6], and Logstash[7], but can also be configured to output data to local files for offline analysis or archival purposes.

The nprobe[8] actuator supports multiple export destinations and operates as a flexible flow exporter. Its primary function is to generate and export flow records using standard protocols such as NetFlow v5 and v9 and IPFIX, which are transmitted to remote flow collectors for storage, analysis, and correlation. nprobe can export flows to commercial and open-source collectors, including ntopng[9], as well as to generic NetFlow/IPFIX receivers deployed in security monitoring and traffic analysis platforms. In addition, nprobe supports exporting flow data to local files for offline processing and forensic analysis, and can be integrated with external processing pipelines through sockets or intermediate collectors.

Similarly, the fprobe[10] actuator is a lightweight flow exporter that generates network flow records from live traffic and forwards them to external flow collectors using standard protocols. It supports exporting flow data primarily via NetFlow v5 and v9, enabling interoperability with a wide range of NetFlow-compliant collectors. Typical deployment scenarios involve exporting flows to centralized collectors, such as ntopng or other NetFlow/IPFIX analysis platforms, where data storage, aggregation, and visualization occur. Due to its minimal resource footprint and focus on flow generation rather than analysis, fprobe is well-suited for high-throughput or resource-constrained environments, where it acts exclusively as an exporter and relies on external collectors for downstream processing.

3.2 File collection and log monitoring

File collection and log monitoring are essential capabilities for achieving visibility into system behavior, security events, and application-level activities across distributed infrastructures.

[5] https://www.elastic.co/beats/packetbeat
[6] https://www.elastic.co/elasticsearch
[7] https://www.elastic.co/logstash
[8] https://www.ntop.org/products/netflow-probes/nprobe/
[9] https://www.ntop.org/products/traffic-analysis/ntopng/
[10] https://fprobe.sourceforge.net/

Logs and structured files produced by operating systems, services, and security components constitute a primary source of information for threat detection and forensic analysis.

Rather than relying on fixed configurations or ad hoc scripts, this service uses a standardized control approach that enables external controllers to manage file and log-based monitoring functions uniformly.

Monitoring sessions are modeled as independent entities, each associated with a specific set of files or directories, export destinations, and collection parameters. The data model captures the lifecycle state of each session, along with its configuration and execution metadata, providing a consistent representation of the monitoring state.

3.2.1. Profile

The File Collection and Log Monitoring Profile defines the standardized control semantics for configuring, managing, and observing file-based monitoring functions. The profile specifies how controllers can interact with file collection capabilities through a uniform set of actions, targets, arguments, and responses, abstracting implementation- and platform-specific details.

The profile centers on the lifecycle management of *file collection sessions*. Each session represents a logical monitoring configuration that defines which files or directories are collected, how data is exported, and under what conditions collection occurs. Through the profile, controllers can start new monitoring sessions, stop existing ones, and query the status and configuration of active or configured sessions.

Table 5 lists the supported actions by this component.

Table 5. List of the supported file collection and log monitoring actions.

Action	Description
query	request the supported capabilities
start	begin a file collection session
stop	stop a file collection session

The `query` action retrieves information about active or configured monitoring instances, including their identifiers, configuration parameters, and current operational status.

The `start` action is instead employed to activate a new file or log monitoring instance based on the parameters provided by a producer, such as source paths, collection format, export destination, and runtime options. Upon execution, the actuator initializes the corresponding monitoring process and records its state to enable subsequent management operations. The `stop` action is used to terminate an active monitoring instance previously started by the same producer, ensuring a controlled shutdown and proper release of system resources.

By decoupling control semantics from concrete implementations, the File Collection and Log Monitoring Profile enables uniform management of file-based monitoring functions across diverse environments. This abstraction allows higher-level services to reason consistently about

file collection and log monitoring, regardless of the specific tools or technologies used at the execution layer.

3.2.2. Actuators

Currently, we support only Filebeat as the underlying tool for collecting files and logs, as show in Table 6.

Table 6. List of the supported file collection and log monitoring actuators.

Tool	Storage	Status
Filebeat	Elasticsearch, Logstash	Ready

The Filebeat[11] actuator offers a lightweight, open-source log shipper developed by Elastic. It is designed to efficiently collect, parse, and forward log data from various sources to a centralized data store or processing system, such as Elasticsearch or Logstash. It operates with minimal resource overhead, making it suitable for deployment on production systems with high availability requirements. It supports a variety of input types and can handle log rotation, multiline events, and structured logging formats.

Like the network flow monitoring service, the Filebeat actuator uses a SQLite database. For each file collection and log monitoring session, the actuator creates a record in the database that stores the monitoring process status and its configuration parameters. The actuator maintains a runtime representation of each session, which is kept up to date in the SQLite database.

[11] https://www.elastic.co/beats/filebeat

4. ENFORCEMENT SERVICES

Task 3.1 developed several services to enforce cybersecurity across a network. In this Chapter, we document the stateless packet filter and the remote command-line interface components used to configure firewalls and issue generic commands remotely.

4.1 Stateless packet filtering

In the following paragraphs, the internal workings, design, implementation, and validation of a unified control framework that implements the OpenC2 StateLess Packet Filtering (SLPF) profile and supports a variety of actuators across multiple firewall technologies. The work demonstrates how OpenC2 can be effectively employed to achieve homogeneous, interoperable, and automatable control of diverse firewall platforms within a common management architecture.

The primary objective of this activity, executed in Task 3.1 Service Context Graphs, is to demonstrate the feasibility and effectiveness of OpenC2 as a unifying control mechanism for stateless packet filtering systems across heterogeneous environments. To this end, the SLPF service offers the following features:

- it supports multiple firewall technologies through dedicated actuator implementations;
- it ensures strict compliance with the OpenC2 Language Specification and the SLPF Actuator Profile;
- we validated the correctness, interoperability, and performance of the proposed solution through systematic testing (documented in Deliverable 2.3).

By following the OpenC2 general architecture, a specific SLPF actuator is managed by a consumer (the MIRANDA connector in this project) which receives a variety of firewall-related commands using the OpenC2 protocol via an ad-hoc producer.

4.1.1. Profile

All SLPF actuators conform to an OpenC2 actuator profile, which defines the common language and actions they support, allowing a producer to configure an SLPF system independently of a specific firewall technology. Table 7 lists the supported SLPF actions.

The `query` command returns a list of supported features by a specific actuator. For instance, a specific firewall instance might not support IPv6 addresses, but only IPv4 addresses. This allows a producer to better send valid commands to a specific actuator.When adding a new firewall rule, the producer can request the insertion of an allow or deny action coupled with the usual 5-tuple selector found in traditional packet filter containing:

- a source IPv4 or IPv6 address in CIDR format;
- a source port number;
- a destination IPv4 or IPv6 address in CIDR format;
- a destination port number;
- a protocol such as TCP, UDP, or SCTP.

Table 7. List of the supported SLPF actions.

Action	Description
allow	insert a rule for permitting a traffic exchange
deny	insert a rule for denying a traffic exchange
delete	remove a rule
query	return the state/settings of the packet filtering device
update	update the firewall configuration or its rule sets

In addition, a rule can be marked as persistent or not. A *persistent* rule will be reapplied when the firewall device reboots; a non-persistent one instead is temporary and will disappear after the reboot. Firewall rules can also have some optional temporal features. The profile allows the insertion of a rule that can optionally specify:

- a time when the rule must be enabled (e.g., starting from 2 PM this Friday);
- a time when the rule must be disabled;
- or a temporal duration (e.g., 2 hours).

4.1.2. Actuators

Our implementation supports four firewall technologies, all of which use the same SLPF profile for consistency. Table 8 lists the supported actuators.

Table 8. List of the supported SLPF actuators.

Firewall	Security function	Status
iptables	CLI	Ready
OpenStack	Security groups	Ready
Kubernetes	Network policies	Ready
Azure	Network security groups	Ongoing

iptables[12] is a widely used Linux packet-filtering framework that operates directly in the kernel. Its rule-based model, organized into tables and chains, closely aligns with the abstractions provided by the SLPF profile. Allow and deny actions are mapped to ACCEPT, DROP, or REJECT rules, while SLPF targets and arguments are translated into iptables match criteria and rule

[12] https://www.netfilter.org/projects/iptables/index.html

parameters. Under the hood, the iptables firewall is actually managed by the actuator using the Linux CLI (Command Line Interface).

The OpenStack SLPF actuator uses OpenStack Security Groups[13], which provide virtual firewalling for cloud instances. Rules are defined declaratively and applied to virtual network interfaces, supporting ingress and egress traffic control. Within the SLPF Actuator, OpenC2 commands are translated into Security Group rule operations through the OpenStack APIs, enabling consistent policy management across multi-tenant cloud environments.

Similarly, the Kubernetes actuator uses Kubernetes Network Policies to regulate traffic between pods and namespaces in containerized environments. Although their declarative and label-based model differ from traditional firewall rule sets, the SLPF profile can still be mapped to Network Policy constructs for ingress and egress filtering. The Kubernetes actuator handles the creation, update, and deletion of Network Policy resources in response to OpenC2 commands.

Finally, the Microsoft SLPF actuator controls the Azure NSGs[14] (Network Security Groups), which implement packet filtering at the virtual network level within the Azure cloud. NSG rules closely resemble traditional firewall rules and support priority ordering, protocols, ports, and address ranges. The Azure actuator leverages the Azure SDK to manage NSG rules in compliance with the SLPF profile.

All SLPF actuators use an SQL database to store the currently active firewall rules. The content of such a database is kept up to date as each OpenC2 command is successfully executed. This is used to ensure consistent behaviour across multiple firewall systems that might not natively support all features offered by the actuators. In practice, the SQL database is primarily used to persist firewall rulesets (e.g., when a firewall is rebooted, the actuator reacts and reconfigures the SLPF with the rules stored in the database) and to enforce temporal rules by configuring the filtering device rule when a time trigger occurs.

4.2 RCLI

The Remote Command-Line Interface (RCLI) addresses the need for a standardized and interoperable mechanism to remotely control heterogeneous security components that expose command-line interfaces as their primary management layer. In many operational environments, especially those involving legacy systems, the command-line remains the most expressive and flexible control interface. However, direct interaction through native shells poses significant challenges for automation, security, and integration within coordinated cyber defence workflows.

Within this context, the RCLI approach leverages the Open Command and Control (OpenC2) paradigm to abstract command execution into a controlled, machine-to-machine interaction model. By encapsulating command invocation, process lifecycle management, and artifact handling within a well-defined actuator profile, RCLI enables secure, auditable, and vendor-agnostic remote control of command-line-driven security functions. The RCLI Profile defines the semantic constraints, data model, and command structure required to represent

[13] https://docs.openstack.org/nova/latest/user/security-groups.html
[14] https://learn.microsoft.com/en-us/azure/virtual-network/network-security-groups-overview

command-line interactions using the OpenC2 language. Rather than introducing new OpenC2 actions, the profile constrains and combines existing actions, targets, and arguments to safely model operations such as command execution, process control, and file or artifact management.

4.2.1. Profile

The profile introduces dedicated targets to represent runtime entities typically associated with command-line execution, including processes, executable artifacts, and managed files. These targets are complemented by arguments that specify execution parameters, environment settings, and contextual metadata.

A key characteristic of the RCLI Profile is its emphasis on controlled execution. Commands are not treated as opaque shell strings but are subject to validation, authorization, and profile-level enforcement. This design reduces the attack surface traditionally associated with remote shell access, while preserving sufficient expressiveness to manage complex security tools and scripts.

Table 9 lists the supported actions by the RCLI profile.

Table 9. List of the supported RCLI actions.

Action	Description
copy	transfer a file to a remote storage
delete	delete a remote file
query	request information about the actuator features
start	launch a process
stop	terminate a process

The `copy` and `delete` actions can be used to upload or delete a file to a remote controller server. This is typically used to upload configuration files needed by a command.

The `start` and `stop` actions are used to launch or terminate a remote process. Note that the remote command to be executed is checked: if a dangerous command is detected, the RCLI will forbid its execution (such as executing `rm -fr /`).

Finally, the `query` action is used to inspect the characteristics supported by an actuator (e.g. the list of forbidden CLI commands).

4.2.2. Actuator

Currently, we support only the Linux RCLI actuator, as shown in Table 10.

The RCLI Actuator is the concrete component responsible for interpreting and executing OpenC2 commands conforming to the RCLI Profile. It acts as a mediation layer between the OpenC2 Consumer and the underlying Linux operating system.

Table 10. List of the supported RCLI actuators.

Tool	Supported OS	Status
Linux RCLI	Linux	Ready

Note that, to safeguard the integrity of the machine running the actuator, several security checks are performed to prevent abuse of the RCLI actuator to execute arbitrary, potentially dangerous code. These checks currently include:

- White-list approach for executing only trusted commands. This list can be configured so that each actuator is customized for its deployment.
- Flag checks. The RCLI actuator checks if a command is executed with some forbidden flags.
- Path checks. The actuator will abort the execution if it receives an unexpected file or folder name as input. For instance, the Linux command kill does not accept a path as input, and doing so will prevent it from executing.
- Rate limiting. The actuator supports a per-minute rate limit on commands.

Finally, the Linux RCLI actuator stores each executed command and uploaded file in a SQL database for auditing purposes and to ensure ownership for historical reasons.

5. ATTACK AND THREAT MODELLING

This section introduces the main components of the Twinning Layer of the MIRANDA available to other components of the MIRANDA infrastructure. Task 3.2, Attack Modelling, uses the CTI provided by the Threat Feeds together with the information obtained through Context Discovery to perform threat analysis using formal methods. The objective of this task is to produce Attack Paths and to support other MIRANDA components in enabling proactive countermeasures and predictive security analysis.

5.1 Threat feeds

The Threat Feed Connector aggregates cyber threat intelligence from heterogeneous sources (e.g. CVE, MISP, STIX 2.1) into a unified REST interface accessible to other MIRANDA components. Each feed employs distinct data formats and severity scales, requiring normalisation algorithms that preserve semantic fidelity while enabling cross-feed analysis. The connector exposes endpoints for keyword-based queries, temporal filtering, and cached retrieval, supporting both local parsing and distributed query routing to external workers.

All parsers transform source data into a common schema comprising identifier, source, title, description, publication timestamp, severity, reference URLs, and raw data. The main interoperability challenge is severity normalisation across incompatible formats. CVE feeds use CVSS scores (9.0–10.0 → Critical, 7.0–8.9 → High, 4.0–6.9 → Medium, <4.0 → Low). MISP employs threat levels (1 → High, 2 → Medium, 3 → Low, 4 → Unknown). STIX maps confidence values to severity labels, while AlienVault TLP (Traffic Light Protocol) markings (Red → High, Amber → Medium, Green/White → Low) indicate sensitivity. The unified schema enables correlation across feeds without requiring consumers to understand feed-specific formats.

The connector prioritises external components over local parsers through a four-stage decision tree. Queries first check cached storage. For uncached requests, the registry identifies healthy components and dispatches parallel queries. If components return zero results or are unavailable, the system falls back to local parsers. This architecture supports horizontal scaling whilst maintaining resilience through automatic failover. Components register via `POST /registerComponent`, providing network address and feed capabilities. A background process evicts stale entries, preventing routing to terminated workers.

Currently, threat data persists in file-backed JSON with content hashing. Before storing, the system computes a canonical hash (sorted keys, excluding raw field) and compares it against existing entries (duplicates are discarded). Write operations use atomic rename (temp file → final path) to prevent corruption. Read errors trigger backup creation and empty storage fallback, prioritizing availability over data preservation.

5.2 Attack modelling

The Attack Modelling component is a core element of the Twinning Layer within the MIRANDA architecture. It is responsible for modelling the services of the SCG, together with the vulnerabilities affecting them and the threats that may exploit such vulnerabilities. In addition,

the component captures the relationships among the various SCG components and systems. Based on this comprehensive model, it performs a structured threat analysis to formally derive the security properties of the services listed in the SCG.

The module represents a direct evolution of the TAMELESS (Threat & Attack ModEL Smart System) methodology, extending its expressive language and modelling capabilities and thereby enabling the identification and reasoning over multi-step attack kill chains. Multi-step attack kill chains can rapidly evolve into highly complex scenarios, combining seemingly unrelated conditions to enable high-impact attacks on the infrastructure. Such scenarios are often difficult to anticipate in advance by human security designers or administrators without the support of automated analysis tools. The enhanced capabilities of the Attack Modelling component allow it to address the requirements of highly dynamic and evolving modern network environments, corresponding to the target operational context of the MIRANDA infrastructure.

5.2.1. Internal Architecture

The internal architecture of the Attack Modelling component is illustrated in Figure 4.

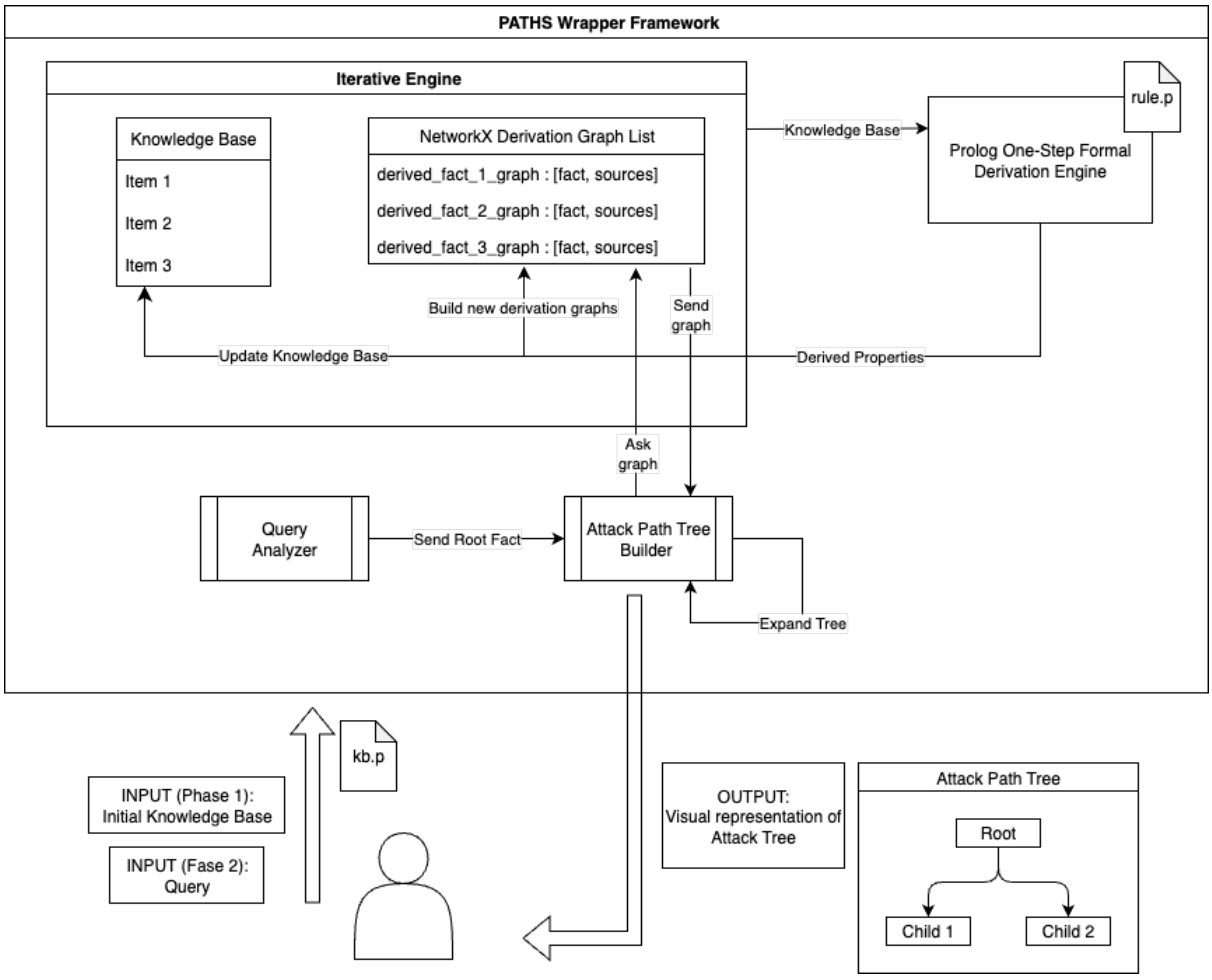

Figure 4. Architecture of the Attack Modelling component.

The architecture comprises several internal modules and engines. The primary ones are the following:

- One-Step Formal Derivation Engine: applies a defined set of formal derivation rules to infer the security properties of the entities represented in the knowledge base.
    - Formal Derivation Rules: a set of logical rules (shown in Figure 4 as *rule.p*) used by the derivation engine to compute security properties related to vulnerabilities, compromise, and malfunctioning conditions. These rules formally specify the inference conditions under which properties can be derived, how threats can propagate across entities, and how lateral movement can occur within the infrastructure.
    - Knowledge Base: the internal representation of the SCG within the Attack Modelling component (shown in Figure 4 as *kb.p*), enriched with information on vulnerabilities affecting its entities and systems, the threats capable of exploiting them, and the relationships among components (e.g., containment, control, and connectivity). The knowledge base is consumed by the derivation engine, which applies the formal derivation rules to it.
- Iterative Engine: interacts iteratively with the Formal Derivation Engine by submitting the Knowledge Base for analysis, updating it with newly derived facts, and maintaining an internal collection of local derivation graphs.
    - Local Derivation Graph: a graph structure that represents the derivation process of a fact produced by the Formal Derivation Engine
- Attack Path Builder: composes multiple local derivation graphs into an Attack Path Tree (or forest, in the presence of multiple roots), encoding the inferred derivation paths that model multi-step attack kill chains.

5.2.2. Operational Workflow

Figure 4 also hints at a high-level overview of the operational workflow of the Attack Modelling component. The workflow begins with the user uploading the knowledge base to the component, either through the available API or the GUI. The knowledge base is then analysed by the One-Step Formal Derivation Engine, which applies its set of formal derivation rules to infer new facts related to security properties and relationships among the entities represented in the model.

The newly derived facts, together with the updated knowledge base, are then passed to the Iterative Engine. The Iterative Engine integrates the new facts into the knowledge base and generates the corresponding local derivation graphs for each derived fact. The updated knowledge base is subsequently resubmitted to the One-Step Formal Derivation Engine, and this process iterates until no additional facts can be derived.

Once the iterative phase is completed, the component enters an interactive phase. During this phase, the user (through the API or GUI) can request the Attack Path Tree or Attack Path Forest associated with a specific fact or set of facts. Upon receiving a query, the Attack Path Builder retrieves from the previously generated local derivation graphs the graph whose root corresponds to the queried fact. In this graph, the leaves represent the source facts that have led to the derivation of the root fact.

For each leaf node, the Attack Path Builder searches the set of local derivation graphs to determine whether a graph exists whose root matches that leaf. When such a graph is found, its leaf nodes are chained to the corresponding leaf of the original graph.

This recursive graph expansion process continues until no further expansions are possible. The component then returns the complete Attack Path Tree (or Forest).

6. DATA HANDLING PIPELINE

This chapter details the design, architecture, and implementation of the Visibility, Enrichment, and Contextualization (VEC) framework within the MIRANDA platform. As the foundational layer for the project's data-centric capabilities, the VEC framework is responsible for the ingestion, normalization, and contextualization of heterogeneous security telemetry. It acts as the bridge between the raw, chaotic data streams generated by multi-ownership urban infrastructures and the sophisticated analytical engines, such as Threat Hunting and Prediction, that reside in the upper layers of the architecture.

The Data Handling Pipeline (DHP) is the core service responsible for orchestrating the lifecycle of data processing workflows. In the context of the MIRANDA ecosystem, where data originates from diverse sources ranging from IoT sensors and network probes to cloud logs and SIEM feeds, a rigid ingestion strategy is insufficient. The DHP addresses this by providing a dynamic, standards-based abstraction layer that decouples the definition of data flows from their execution engine.

6.1 Architectural Philosophy and Principles

The architectural design of the DHP is not merely a technical implementation of a log aggregator; it is a realization of the OpenC2 (Open Command and Control) standard applied to data engineering. The system is built upon four foundational principles that dictate its internal logic, ensuring interoperability and resilience across the computing continuum.

6.1.1. Principle 1: Abstraction of Intent

The primary driver for the DHP is the strict separation of **intent** from **implementation**. In traditional systems, data pipelines are often defined by engine-specific configuration files (e.g., `logstash.conf`, `fluentd.yaml`), which tightly couples the orchestration layer to the data plane. The DHP breaks this dependency. The MIRANDA Orchestrator (the Producer) issues abstract commands defining *what* needs to be done, for example, "ingest from Socket A, parse via Pattern B, and route to Storage C." It possesses no knowledge of the underlying technology. The translation of this intent into concrete execution logic is the sole responsibility of the **Actuator**. This abstraction ensures that the underlying processing engine can be swapped (e.g., from Logstash to Vector) without requiring changes to the Northbound API or the orchestration logic.

6.1.2. Principle 2: The Command/Target Paradigm

To ensure a deterministic API surface, the DHP rejects loose RESTful endpoints in favour of the strict OpenC2 grammar. Every interaction follows a structured "Action → Target" paradigm.

Action: The specific verb describing the operation (e.g., `start`, `stop`, `query`).

Target: The specific noun the action operates upon (e.g., `pipeline`, `pipeline_id`).

This paradigm enforces a rigid API contract. For example, the capability to create a workflow is strictly bound to the `start` action on the pipeline target, while the capability to terminate it is bound to the stop action on the `pipeline_id` target. This prevents ambiguity and simplifies the validation logic within the service.

6.1.3. Principle 3: Extensibility via Profiles

The core OpenC2 language defines common actions, but specialized domains require specific vocabularies. The DHP implements a custom OpenC2 Profile named **x-dpp (Data Processing Profile)**. This profile extends the standard language with domain-specific targets (e.g., `DataSource`, `TransformationRule`) and data structures. By defining these extensions formally within a dedicated namespace (`nsid: x-dpp`), the DHP ensures that its capabilities are discoverable, self-describing, and strictly typed, preventing conflicts with other MIRANDA control planes.

6.1.4. Principle 4: Statelessness and Resilience

Designed for deployment in containerized environments (e.g., Kubernetes), the DHP service is architected to be stateless. The application logic does not hold critical pipeline state in volatile memory. Instead, state is externalized to persistent stores: (i) An SQLite database serves as the source of truth for metadata and configurations; ii) a shared storage volume holds the active configuration files. This design guarantees that if the DHP container crashes or is rescheduled, it can immediately reconstruct the operational state of the system by querying the database, ensuring zero data loss regarding pipeline management.

6.2 Service Architecture

The DHP architecture is composed of distinct logical layers that separate the API interface, the abstract processing logic, and the concrete execution engine (as shown in Figure 5).

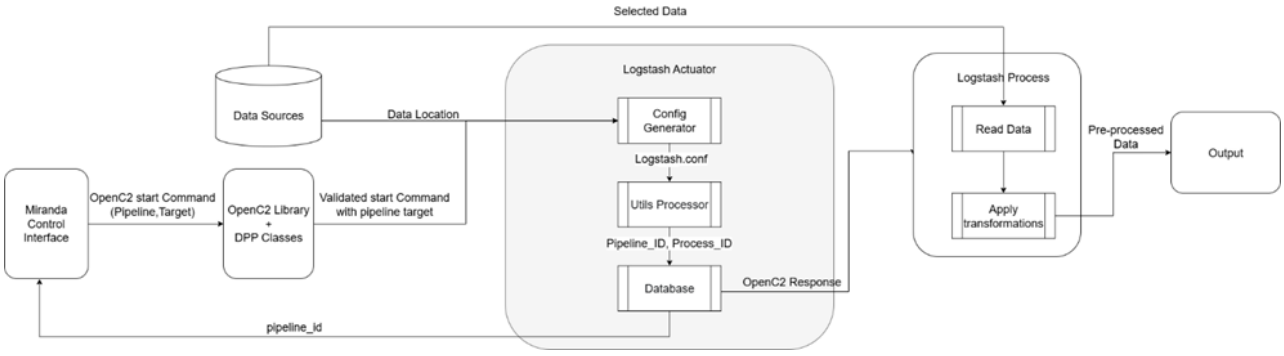

Figure 5: Detailed Architecture of the Data Handling Pipeline.

6.2.1. The Abstraction: DataProcessor

The entry point for all logic is the `DataProcessor` class. This component acts as the "switchboard" for the service, handling generic OpenC2 "boilerplate" tasks so that the specific engine implementation remains clean. Its responsibilities include:

- **Command Validation:** It invokes the validation logic to ensure incoming JSON payloads adhere strictly to the schema.
- **Actuator Targeting:** It implements security logic to check the `asset_id` field of incoming commands, ensuring that requests are routed to the correct specific DHP instance in a distributed deployment.
- **Action Routing:** It maps abstract OpenC2 Actions to internal method calls (e.g., routing `Action.QUERY` to `_query_features` or `_query_pipeline`).

6.2.2. The Implementation: LogstashActuator

The `LogstashActuator` inherits from the `DataProcessor` and serves as the translation engine for the current reference implementation. It is responsible for bridging the gap between the abstract OpenC2 objects and the Logstash Data Processing Engine. It manages the lifecycle of the underlying OS processes implicitly by manipulating configuration files in the shared volume and maintaining synchronization with the state database.

6.3 Data Model

The schema that governs all DHP interactions is defined in the `x-dpp` profile (as shown in Figure 6). This contains the complete definition of the custom nouns and data structures used by the system.

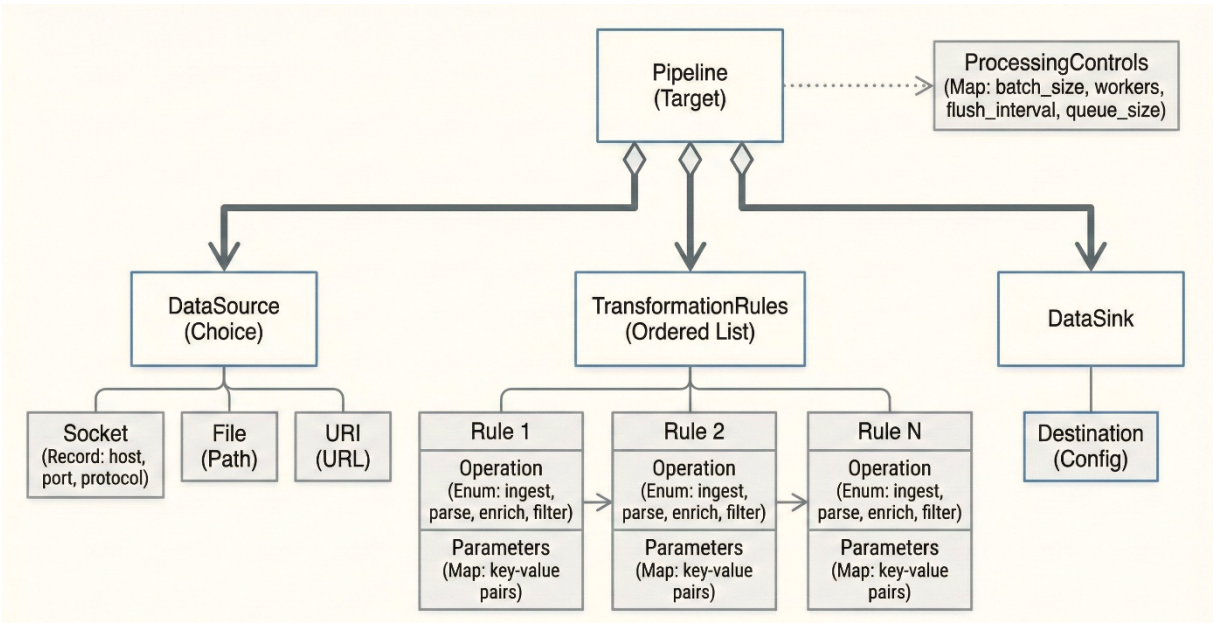

Figure 6: Data Processing Profile (x-dpp) data model.

6.3.1. Core Targets

The profile registers three primary targets that represent the resources managed by the service:

- **`x-dpp:pipeline`:** The central object representing a complete processing job. It is a composite structure aggregating arrays of `DataSource`, `TransformationRule`, and `DataSink` objects. This is the mandatory target for start commands.
- **`x-dpp:pipeline_id`:** A strongly-typed string identifier representing a specific, instantiated pipeline. Unlike a generic string, this type enforces format validation to ensure safe reference handling during stop and query operations.
- **`x-dpp:running_pipelines`:** An abstract target used solely for discovery. It contains no data structure itself but signals the intent to retrieve the collection of all active pipeline IDs.

6.3.2. Modular Data Structures

To normalize the definition of data flows, the profile defines granular data types:

A. Data Source

The `DataSource` object is implemented as a *Choice* structure, allowing pipelines to be instantiated with varying ingress methods.

- **Socket:** This is a structured Record containing a host (validated IPv4 address), port (integer), and protocol (Enumerated TCP/UDP). This structure prevents malformed network configurations (e.g., invalid ports) from ever reaching the actuator.
- **File:** Represents ingestion from static log files or file-system buffers.
- **URI:** Supports ingestion from remote web resources or APIs.

B. Transformation Rules

Enrichment and normalization logic is defined via `TransformationRule` objects. A pipeline contains an ordered list of these rules, each consisting of:

- **Operation:** Defined in operation.py, this is an Enumerated type that maps human-readable actions to stable integer values. Supported operations include ingest, parse, enrich, and filter. By using an enumeration, the system insulates the API client from internal implementation details; for example, the abstract parse operation maps to the grok filter in Logstash, but the integer identifier remains constant even if the backend changes.
- **Parameters:** A dictionary of key-value pairs specific to the operation (e.g., defining the specific regex pattern for a parse operation).

C. Data Sink

Defines the destination and format of the processed output (e.g., JSON storage).

D. Processing Controls

To support the optimization of pipelines across different tiers of the computing continuum (Edge vs. Cloud), the profile includes `ProcessingControls`. This Map structure allows operators to pass performance-tuning parameters such as `batch_size`, `workers`, `queue_size`, and `flush_interval`. This separates the *logical* definition of the pipeline from its *operational* tuning, allowing the same pipeline logic to be deployed with different performance profiles depending on available resources.

6.4 Implementation Logic and Workflows

The operational capability of the DHP is driven by rigorous logic flows implemented within the `LogstashActuator`.

6.4.1. Pipeline Initialization (Start)

When a start command is received, the Actuator executes the following sequence:

1. **Validation:** The payload is validated against the x-dpp schema.
2. **Identification:** The utility `random_name_generator.py` assigns a unique, human-readable ID to the pipeline (e.g., `pipeline_Alpha_1234`).
3. **Compilation:** The Actuator invokes the `LogstashConfigGenerator`. This critical utility acts as a compiler, iterating through the `DataSource` and `TransformationRule` objects and generating the specific Logstash DSL syntax (e.g., converting a Socket object into a tcp { ... } input block).
4. **Persistence:** The full pipeline configuration is serialized to JSON and stored in the SQLite database. This ensures a permanent record of the requested state.
5. **Execution:** The compiled configuration string is written to a `.conf` file in the shared volume. The Logstash service, monitoring this directory, automatically detects the new file and spawns the pipeline worker.
6. **Response:** A standard OpenC2 response containing the new `pipeline_id` is returned.

6.4.2. Pipeline Termination (Stop)

The termination workflow ensures clean resource cleanup:

1. **Resolution:** The actuator queries the database to verify the existence of the provided `pipeline_id`.
2. **Detonation:** It constructs the specific `filepath` for the configuration and deletes it from the shared volume. This action signals `Logstash` to gracefully terminate the processing thread.
3. **Cleanup:** The record is removed from the persistent database to reflect the system's new state.

6.4.3. Introspection and Discovery (Query)

The DHP supports deep introspection to facilitate orchestration:

- **Feature Discovery:** Upon receiving a query features command, the actuator uses the ProcessorConfigLoader to read local definition files (`capabilities.json`, `operations.json`). It returns a list of supported profiles, versions, and operations, allowing the Orchestrator to dynamically discover which operations (e.g., "does this node support GeoIP enrichment?") are available on a specific node before issuing commands.
- **State Reconstruction:** When querying a `pipeline_id`, the actuator retrieves the stored JSON blob from the database and deserializes it back into a `Pipeline` object. This confirms that the DHP can report on the exact logical configuration of active jobs without needing to parse the underlying engine's config files.

The actuator is fully compliant with the OpenC2 standard and supports the aforementioned commands, as summarised in Table 11.

Table 11. List of supported commands in the Data Handling Pipeline.

Action	Target	Description
Start	x-dpp:pipeline	Creates and launches a new data processing pipeline. The actuator generates the specific Logstash configuration, persists the state, and triggers execution.
Stop	x-dpp:pipeline_id	Terminates a running pipeline. The actuator removes the configuration file and updates the system state.
Query	features	Returns the capabilities of the actuator, including supported profiles, OpenC2 versions, and supported operations.
Query	x-dpp:running_pipelines	Returns a list of all currently active PipelineIDs.
Query	x-dpp:pipeline_id	Retrieves the detailed configuration (Sources, Transformations, Sinks) of a specific active pipeline.

The actuator also implements an **Actuator Specifier** mechanism (e.g., `asset_id`). This allows the MIRANDA platform to target specific DHP instances (e.g., "`Edge-Processor-01`" vs "`Core-Processor`"), enabling precise control in distributed or multi-tenant environments.

6.5 Integration and Security

To function securely within the multi-ownership environment of MIRANDA, the DHP integrates specific security handlers.

A. User Context and Multi-Tenancy

The service implements producer attribution logic. The `user/config.py` module resolves a `PRODUCER_ID` from the execution context. This ID is used to scope all database interactions. In shared environments, this ensures that pipelines are strictly owned by the entity that created them, preventing unauthorized modification of shared resources.

B. Actuator Specifiers

To support distributed scalability, the DHP leverages the OpenC2 Actuator specifier. Commands may include an `asset_id` (e.g., `asset_id="edge_node_04"`). The `DataProcessor` logic inspects this field and ignores commands not addressed to its specific instance. This allows the MIRANDA Orchestrator to broadcast directives to a fleet of DHP nodes while targeting execution to specific processors.

C. Response Standardization

All outcomes are managed by a centralized `response_handler` module. This utility maps internal states to standard OpenC2 response codes (e.g., `200 OK`, `400 Bad Request`, `500 Internal Error`). It populates a custom `Results` object (defined in `results.py`) that extends the standard schema to include fields like `pipeline_config` and `running_pipelines`, ensuring predictable, strictly typed responses for all API consumers.

6.5.1. Producer-consumer interaction

The DHP exposes its capabilities via an **API Server** that functions as an OpenC2 Consumer.

1. **Transport:** The interaction uses HTTP as the underlying transport protocol.
2. **Command Flow:** A Producer (such as the MIRANDA GUI or Orchestrator) issues a command (e.g., `start pipeline`) serialized in JSON.
3. **Validation:** The API Server passes the request to a generic `DataProcessor` which validates the command against the strict `x-dpp` schema.
4. **Execution:** The `LogstashActuator` translates the abstract command into a concrete configuration file, writes it to the shared volume, and updates the database.
5. **Response:** The actuator returns a standard OpenC2 response (e.g., `200 OK` containing the new `pipeline_id`) back to the `Producer`.

This interaction model ensures that the complexity of the underlying log processing engine is completely hidden from the consumer, facilitating the automation of data flows across the MIRANDA ecosystem.

7. CONCLUSIONS

This deliverable presents the work carried out in Work Package 3 “Cybersecurity Digital Twin” of the MIRANDA project during the first half of the project. It has reported on the activities performed in Tasks 3.1, 3.2, and 3.3, which address context discovery, monitoring, enforcement, attack and threat modelling, and visibility, enrichment, and contextualization.

The document has described the design principles, architectures, and implementations developed, with a focus on the abstractions and functional capabilities of the WP3 components. The reported results illustrate the adoption of a standards-based approach, primarily relying on OpenC2, to enable interoperable control and coordination of heterogeneous security services across different environments.

Task 3.1 has delivered context discovery, monitoring, and enforcement services that support visibility into infrastructure behavior and controlled management of security functions. Task 3.2 has developed a threat feed and an attack and threat modelling framework that combines cyber threat intelligence with contextual information to support structured security analysis. Finally, Task 3.3 has defined a data handling pipeline for the ingestion, processing, and contextualization of heterogeneous security data.

From a scalability perspective, the WP3 components described in this document are designed to grow by replication rather than reinvention. The context discovery service offers *lighter* discovery modes to cope with large topologies by enabling caching and trading off speed for information freshness. Common services (e.g., firewalls) expose uniform OpenC2 control surfaces, so capacity can be increased by adding more actuator-backed instances behind the same abstract commands, thus distributing and alleviating the overhead of controlling multiple micro-services. The threat feed tool improves scalability through caching and temporal filtering, reducing redundant work for recurring queries and enabling faster retrieval under load. The entire workflow is also reinforced by the Data Handling Pipeline's stateless service design, which externalizes operational state to persistent stores, enabling rapid restarts and horizontal scale-out. It is also worth noting that the attack modelling component, one of MIRANDA’s most computationally demanding components, is primarily intended for offline and occasional use; as a result, it is not a primary driver in the scalability considerations discussed here.

Overall, this deliverable provides a consolidated view of the WP3 results achieved during the first half of the project and serves as a basis for subsequent integration and validation work.

This project has received funding from the European Union's Horizon Europe Research and Innovation programme under grant agreement No. 101168144. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Cybersecurity Competence Centre. Neither the European Union nor the granting authority can be held responsible for them.