Ref. Ares(2022)1248100 - 18/02/2022

ICT-56-2020 "Next Generation Internet of Things"
Grant Agreement number: 957218

IntellioT

Deliverable D5.1
Integrated IntelIloT framework & use case implementations (first version)

Deliverable release date	18/02/2022
Authors	1. SIEMENS: Andreas Zirkler, Andreas Ziller, Christian Bachmann 2. EURECOM: Jérôme Härri 3. AAU: Beatriz Soret, Lam Duc Nguyen, Andreas Casparsen 4. UOULU: Sumudu Samarakoon, Mohamed Abdelaziz 5. TTC: Martijn Rooker 6. TSI: Andreas Brokalakis, Babis Savvakos 7. PHILIPS: Anca Bucur, Nancy Irisarri Méndez 8. SANL: Konstantinos Fysarakis, Ioannis Vezakis 9. HSG: Simon Mayer 10. HOLO: Carina Pamminger, Nour Fendri 11. AVL: Wolfgang Hollerweger, Holger Burkhardt
Editor	Wolfgang Hollerweger (AVL)
Reviewer	Simon Mayer (HSG), Carina Pamminger (HOLO)
Approved by	PTC Members: (Vivek Kulkarni, Konstantinos Fysarakis, Sumudu Samarakoon, Beatriz Soret, Arne Bröring, Maren Lesche) PCC Members: (Vivek Kulkarni, Jérôme Härri, Beatriz Soret, Mehdi Bennis, Martijn Rooker, Sotiris Ioannidis, Anca Bucur, Georgios Spanoudakis, Simon Mayer, Filippo Leddi, Holger Burkhardt, Maren Lesche, Georgios Kochiadakis)
Status of the Document	Final
Version	1.0
Dissemination level	Public

Table of Contents

Figures 4
Acronyms and Definitions 6
1 Introduction 9
1.1 Outline of this Deliverable 9
2 Component integration 10
2.1 Overview of Integration and Implementation Methodology 10
2.2 Collaborative IoT Enablers 11
2.2.1 Hypermedia MAS Infrastructure 11
2.2.2 Web-based IDE for Hypermedia Multi-Agent systems 16
2.2.3 Interoperability Box 16
2.2.4 Global and Local AI 18
2.3 Human in the Loop Enablers 20
2.3.1 HIL Service 20
2.3.2 HIL Application 22
2.3.3 End User Goal Specification Interfaces 25
2.3.4 Remote Operator Devices 25
2.4 Trust Enablers 25
2.4.1 Overall Trust Enablers Integration 26
2.4.2 Security Assurance Platform 29
2.4.3 Trust-based IDS 37
2.4.4 Moving Target Defences 39
2.4.5 Authentication – Authorisation – Accounting (AAA) 42
2.4.6 Distributed Ledger Technologies 45
2.5 Infrastructure Management Components 48
2.5.1 Edge Management 48
2.5.2 Dynamic Network Management 49
3 Use Case specific Implementations 64
3.1 UC1: Agriculture 64
3.1.1 Tractor Controller 64
3.1.2 Integration with the Agriculture AI models 66
3.1.3 Remote Operator Device 67
3.2 UC2: Healthcare 68
3.2.1 Smart Devices 69
3.2.2 Integration with the Healthcare AI Models 69
3.2.3 Integration with the PATIENTS' Data Repository 69
3.3 UC3: Manufacturing 69

3.3.1 Robot controller 69
3.3.2 Engraver App 70
3.3.3 Integration with the Manufacturing AI models 74
3.3.4 Remote Operator Device 74
4 Conclusions and future work 76
References 77

FIGURES

Figure 1: The IntellIoT framework – example artifacts and devices are shown in the upper, deployment part. 9
Figure 2: private GitLab group of IntellIoT 10
Figure 3: UC1 demo setup worked out at Consortium Meeting#5 11
Figure 4: Component architecture of the IAKM (server and client) 13
Figure 5: Process flow of IAKM assisted FML training and FML model exchange 14
Figure 6: IAKM REST process for FML service 15
Figure 7: IAKM subscription/matching process for FML service 15
Figure 8: IAKM FML service demonstration 16
Figure 9: High-level, logical architecture of Interoperability Box 17
Figure 10: HIL support operation sequence chart. 21
Figure 11: HIL service HTTP interface 22
Figure 12: Human in the Loop Interfaces. 23
Figure 13: Unity3D application for HoloLens 2 and MQTT communication. 23
Figure 14: 14 ISAR - Server to Client interface. 24
Figure 15: ISAR Interaction Diagram. 24
Figure 16: HIL Application: Machine and Weather Data. 24
Figure 17. Trust Enablers UML logical diagram. 26
Figure 18. Overview of Trust Enablers' integration via shared Trust Broker. 27
Figure 19. Deployment view of local Trust Enablers' integration testbed. 27
Figure 20: Recording of SAP Assessment Results in the ledger, via interaction with DLT Manager (request & response). 28
Figure 21. Overall view of "Demonstrator" testing project and related KPIs within the SAP front end 30
Figure 22. View of defined assets and executed assessments during SAP testing. 31
Figure 23. SAP testbed Monitoring assessment results. 32
Figure 24. SAP Vulnerability Assessment carried out on testbed setup (Overview, KPIs). 33
Figure 25. SAP Vulnerability Assessment carried out on testbed setup (viewing details on results). 34
Figure 26. SAP Dynamic Testing assessment carried out on testbed setup (Overview, KPIs). 35
Figure 27. SAP Dynamic Testing assessment carried out on testbed setup (viewing details on results). 36
Figure 28. Exported SAP assessment report in PDF format (in this case, of a vulnerability assessment). 37
Figure 29. Architecture of the Trust-based IDS tool. 38
Figure 30: Communication between IDS components and the MTD and SAP modules 39
Figure 31 General communication architecture of Ethereum DLT network. 45
Figure 32 Implementation of DLT managers. 46
Figure 33: Smart contract of recording sensing data. 47
Figure 34 The migration of smart contracts. 47
Figure 35 Signing a transaction. 48
Figure 36: Sending a transaction from clients to distributed ledger. 48
Figure 37 Querying transaction details. 48
Figure 38: Topology view with stream information in Postman. 50
Figure 39: TSN controller interface description in Swagger. 51
Figure 40: Topology view with stream information in Web UI. 52
Figure 41: Communication service request via the Web UI. 53
Figure 42: Communication service details in Web UI. 54
Figure 43: TSN test app, sender side. 54
Figure 44: TSN test app, receiver side. 55
Figure 45. LL-MEC HTTP API available to the communication resource manager. 56
Figure 46. FlexRAN HTTP API available to the communication resource manager. 56
Figure 47: Mosaic5G FlexRAN HTTP API. 58
Figure 48: Mosaic5G FlexRAN-based RNIS service. 59
Figure 49: Mosaic5G FlexRAN-based ElasticMonitoring HTTP API. 59
Figure 50: OpenAirInterface 4G/5G Core [source: OAI] 60
Figure 51: OpenAirInterface 4G/5G Core software repository 61

Figure 52: OpenAirInterface 5G RAN NSA and SA architecture (source: OAI).....62
Figure 53: Illustration of the 5G RAN UE radio unit. 62
Figure 54: Illustration of the 5G RAN gNB Remote Radio Unit (RRU) unit. 63
Figure 55: Schematic overview of the Tractor Controller. 64
Figure 56: Test setup for tractor controller 65
Figure 57: tractor controller Ubuntu bionic console. 65
Figure 58: Tractor Controller setup IP tables. 65
Figure 59: Functionality testing of Docker with hello-world app.....66
Figure 60: tractor controller runDLT manager. 66
Figure 61: Code Sample - Custom Send Function - Server Side. 68
Figure 62: Code Sample - Custom Received Function 68
Figure 63: Code Sample - Custom Send Function - Client Side.....68
Figure 64: Example workflow of production with laser cutter. 71
Figure 65: Example workflow of production with milling machine. 72
Figure 66. /job/text example. 73

ACRONYMS AND DEFINITIONS

Acronym	Definition
5G NR	5th Generation New Radio
AES	Advanced Encryption Standard https://en.wikipedia.org/wiki/Advanced_Encryption_Standard
AI	Artificial Intelligence
AMPQ	Advanced Message Queuing Protocol https://www.amqp.org/
API	Application Programming Interface
AR	Augmented Reality
CAN	Controller Area Network
CBC	Cipher Block Chaining
CIDR	Classless Inter-Domain Routing
CN	Core Network
COVID-19	Corona Virus Disease of 2019
Dapps	Decentralized apps
DLT	Distributed Ledger Technology
ECG	Electrocardiogram
EMS	Edge Management System
ETH	*Eth*er crypto currency
FML	Fast Machine Learning
GPS	Global Positioning System
HMAC	(Keyed) Hash-based Message Authentication
HMU	Head Mounted Unit
HyperMAS	Hypermedia Multi-agent System
IAKM	Infrastructure Assisted Knowledge Management
ID	Identification
IDS	Intrusion Detection System
IMSI	Internal Mobile Subscriber Identity https://en.wikipedia.org/wiki/International_mobile_subscriber_identity
IO	Input Output
IoT	Internet of Things
ISAR	Interactive Streaming for Augmented Reality
IT/OT	Information Technology/Operation Technology

JSON	JavaScript Object Notation
JSON-LD	JSON for Linking data; JSON-LD - JSON for Linking Data
JSON-RPC	https://www.jsonrpc.org/ Remote procedure call
JWT	JSON Web Token; JSON Web Tokens - jwt.io
Keycloak	https://www.keycloak.org Add authentication to applications and secure services with minimum effort
LLDP	https://en.wikipedia.org/wiki/Link_Layer_Discovery_Protocol
MEC	Mobile Edge Computing
MTD	Moving Target Defences
Node-RED	Low-code programming for event-driven applications https://nodered.org/ built on Node.js
NodeJS	Node.js (https://nodejs.org)
OAuth	https://oauth.net/ a way to get access to protected data from an application. It's safer and more secure than asking users to log in with passwords
openID	https://openid.net/connect a simple identity layer on top of the OAuth 2.0 protocol
OpenAPI	https://www.openapis.org
QoS	Quality of Service
RabbitMQ	https://www.rabbitmq.com open-source message broker
RAN	Radio Access Network
ROS	Robot Operating System: https://www.ros.org
RNTI	Radio Network Temporary Identifier
SAP	Security Assurance Platform
Solidity	https://soliditylang.org a statically-typed curly-braces programming language designed for developing smart contracts that run on Ethereum
TD	Things Description of W3C WoT
TLS	Transport Layer Security
TSN	Time Sensitive Networking
TUN	network TUNnel (Internet protocol/ Link Layer)
UC	IntellIoT Use Case (1: agriculture, 2: healthcare, 3: manufacturing)
UDP	User Datagram Protocol
UE	User Equipment
UML	Unified Modelling Language
UR5	Robot arm https://www.universal-robots.com/products/ur5-robot
VLAN	Virtual Local Area Network

VPN	Virtual Private Network
VR	Virtual Reality
W3C	World Wide Web Consortium
WoT	Web of Things
WP	Work Package of IntellIoT project

1 INTRODUCTION

This deliverable D5.1 summarizes the work that has been done in the first cycle of Task 5.1, related to the components and their interfaces in the 3 use cases (UCs) of IntellIoT. The final, updated version is D5.4 due in Month 32 of the project and will include the final definition of the use cases and the definition of the second open call.

The aim of Task 5.1 is to implement the IoT applications, underlying services and planned demonstrators of the use cases, while remaining in the developer environment and utilizing mostly simulated data. The step towards the demonstration environment will be done in Task 5.2. Deliverable D5.1 will therefore provide the first version of the integrated IntellIoT framework of the first release of technology components (from WP3 and WP4) as well as the first implementation of use case applications and services. The components of the IntellIoT framework are based on the 3 pillars (a) human-defined autonomy, (b) distributed, self-aware IoT applications and (c) an efficient, reliable and trustworthy computation & communication infrastructure as indicated Figure 1.

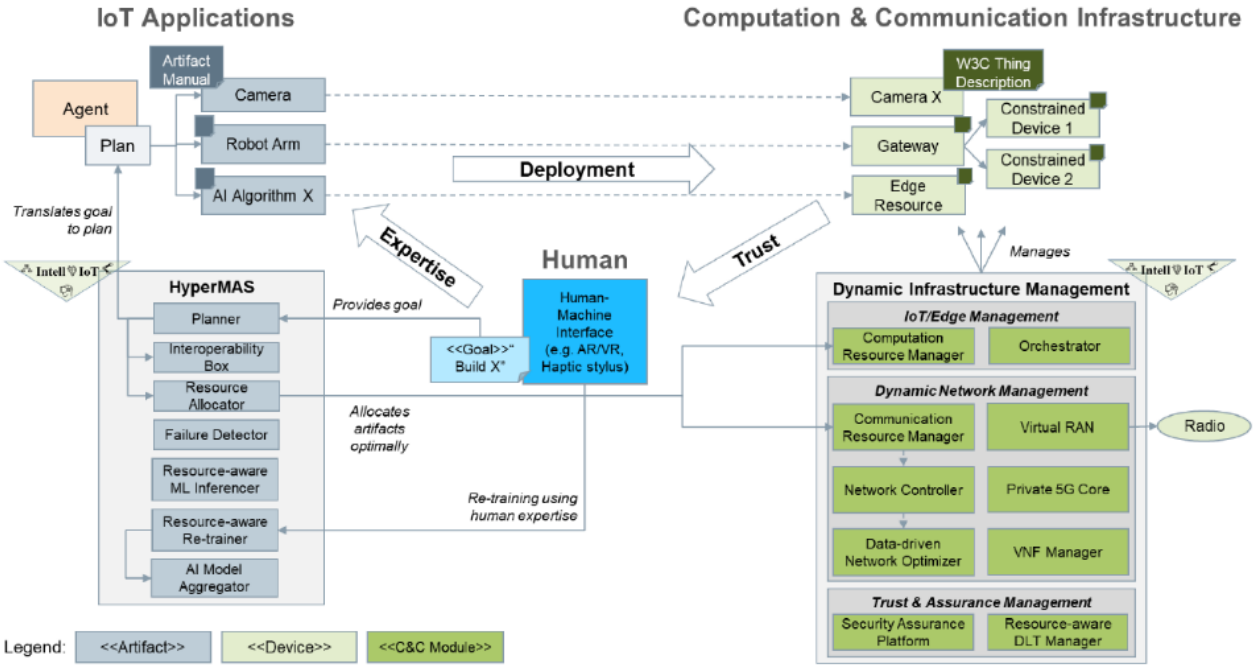


Figure 1: The IntellIoT framework – example artifacts and devices are shown in the upper, deployment part.

The relation to other Tasks and Deliverables of the IntellIoT project is the following:

In T2.3, the architecture specification & interoperability the components were defined and described in D2.3 High Level Architecture. In the first versions of the deliverables of tasks WP3 (distributed self-aware IoT applications) and WP4 (Efficient, reliable and trustworthy computation & communication infrastructure) the technological components making up the IntellIoT framework have been described in detail. T5.2 is following T5.1 in the project's progress. It will deploy, test and demonstrate the IntellIoT framework components and use case applications in the demonstration environment, that means we will conduct the demonstrations for the 3 use cases in real-life settings by incorporating, e.g., an actual tractor, real patients, and actual manufacturing devices and machinery.

1.1 Outline of this Deliverable

To present the work of Task 5.1 on integrating the components, the remainder of this deliverable is organized as follows:

- Chapter 2 details the integration efforts on the different components of the IntellIoT framework.
- Chapter 3 describes the use case specific implementations, particularly, the implemented of applications and services.
- Chapter 4 concludes this deliverable and outlines future work.

2 COMPONENT INTEGRATION

In D2.3 the high-level architecture of IntellIoT was described. Components were identified and clustered into five core component groups (compare D2.3/*Figure 12*). In the deliverables of WP3 and WP4 the functionality and design of those components is described. Before in D5.2 deployment of these components to UC1-3 is done for the first time, they are integrated in the developer environment. Therefore, in the following sections the test environment, inputs and evaluation of the outputs are described as far as currently available. The following sections represent the first four core component groups, and we describe the individual components within these groups in the respective subsections. The fifth core component group is covered separately in Chapter 3.

2.1 Overview of Integration and Implementation Methodology

During this first phase of the project some components are still not deployed (this is content of follow-up task T5.2), but work is done in different developer environments. Nevertheless, as a first step to *Continuous Integration* (CI) a *private GitLab group* was set up with sub-groups for the pillars of IntellIoT containing further sub-groups and projects for the components.

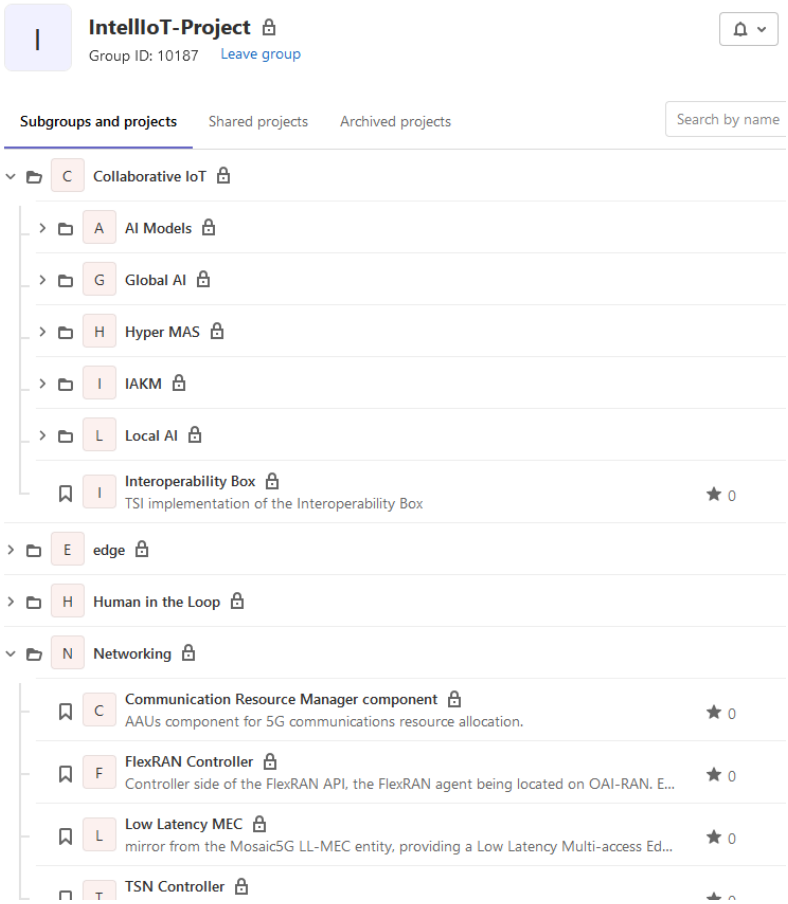


Figure 2: private GitLab group of IntellIoT

From May 2021 to January 2022 fifteen bi-weekly WP5 meetings including T5.1 topics were held. When the consortium members finally could meet face to face in Munich at the consortium meeting #5 (Nov 2nd & 3rd 2021) it was easier to align integration issues of the 3 UCs. A result was e.g., the UC1 demo setup slide in Figure 3.

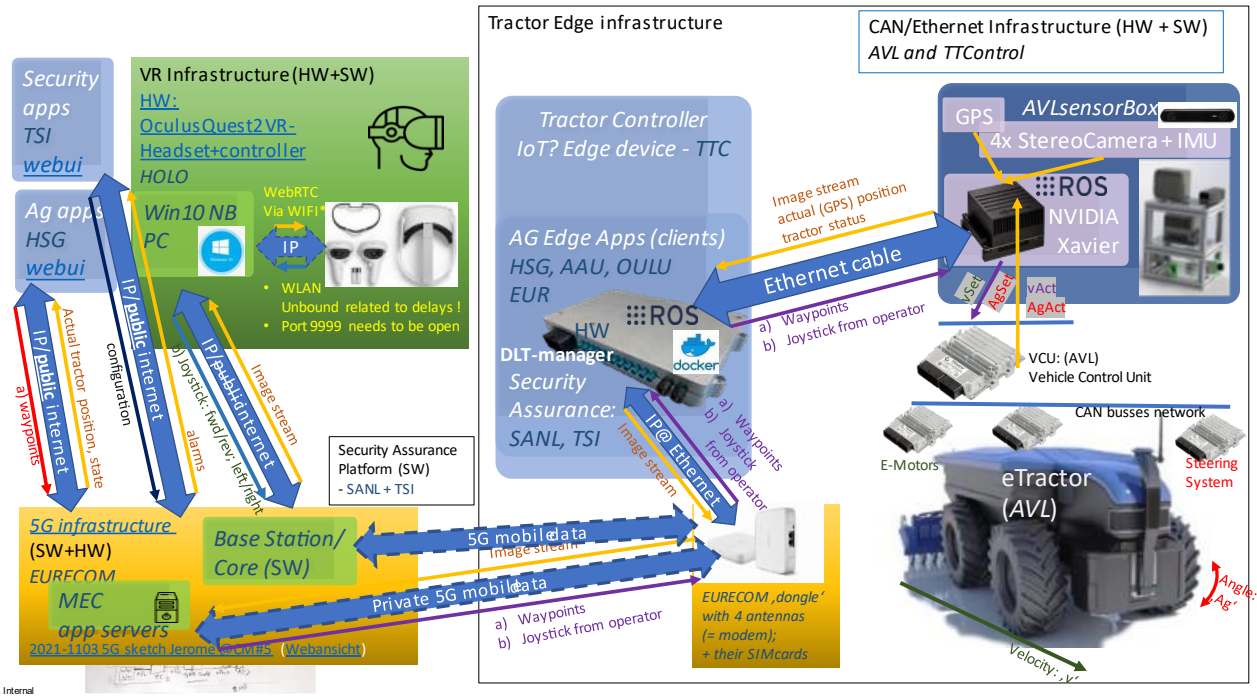


Figure 3: UC1 demo setup worked out at Consortium Meeting#5

2.2 Collaborative IoT Enablers

2.2.1 HYPERMEDIA MAS INFRASTRUCTURE

The Hypermedia MAS Infrastructure[1] is a platform that is used to create and deploy hypermedia environments that are based on the Agents & Artifacts Meta-Model. Its key abstractions are "environments", "workspaces", and "artifacts", all of which are modelled as W3C WoT Things and all of which expose their functional interfaces using W3C WoT Thing Description[2]. The infrastructure furthermore permits Agents to observe resources in the hypermedia environment through a W3C WebSub Hub. Since the Hypermedia MAS Infrastructure is described in detail in Section 3.1 of IntellIoT Deliverable D3.1, we only give a brief account of its capabilities for contextualization in this integration-oriented document.

The current version of the Hypermedia MAS Infrastructure is deployed on HSG's servers[3] and is reachable world-wide. To machine clients, it then provides the functionality to create, populate, and update hypermedia environments, workspaces, and artifacts. Relevant devices that are deployed in HSG's laboratories (e.g., mobile robots[4] and stationary robots[5]) that mock devices from IntellIoT partners (in this case, AVL's autonomous tractors and Siemens' UR5 robots) as well as selected devices that are deployed with IntellIoT partners (e.g., the UC3 Mock Engraver[6]) are registered with the infrastructure using W3C WoT TD. Given this setup, for the registered

[1] https://github.com/Interactions-HSG/yggdrasil
[2] These refer to the World Wide Web Consortium's (W3C) Web of Things (WoT) standardization group that has recently standardized Thing Descriptions (TDs) that describe the metadata and interfaces of WoT Things.
[3] https://yggdrasil.interactions.ics.unisg.ch/
[4] https://yggdrasil.interactions.ics.unisg.ch/environments/intelliot/workspaces/uc-agriculture/artifacts/spockbot
[5] http://yggdrasil.interactions.ics.unisg.ch/environments/intelliot/workspaces/uc-industry/artifacts/leubot1
[6] https://yggdrasil.interactions.ics.unisg.ch/environments/intelliot/workspaces/uc-industry/artifacts/engraver

(hypermedia) artifacts (i.e., the robots, engraver, etc.), the infrastructure exposes the W3C WoT TD-described hypermedia controls to permit Agents to interact with this artifact and to permit Domain Experts to configure Agents to autonomously interact with it (see Section 2.2.2, Web-based IDE for Hypermedia Multi-Agent systems). In addition, artifact TDs can be updated, and clients can subscribe to receive notifications through the infrastructure's W3C WebSub Hub. Furthermore, the infrastructure exposes semantic containment relationships using the EVE vocabulary (see [1] and [2]) for workspaces and environments that permit the crawling of all registered resources and, thus, the efficient searching for artifacts that might be required by a client.

2.2.1.1 INFRASTRUCTURE ASSISTED KNOWLEDGE MANAGEMENT (IAKM)

The IAKM module has a dual task of assisting in the identification of a potential AI models required by an agent according to a specific semantic, as well as assisting in the training of an AI model according to the model's required environment semantics. Accordingly, the IAKM provides two main functions to other IntellIoT components:

- An HTTP API to subscribe to train or use a required model.
- A JSON-based semantic for AI models uniquely describing the AI model or the AI environment required to use, respectively to train.

2.2.1.2 IAKM INTERNAL FUNCTIONS

The IAKM has two components; *IAKM backend* and *IAKM client*, securely connected and provided as a multi-microservice docker container.

The *IAKM backend* is a set of microservices provided as a multi-docker container and providing the infrastructure side of knowledge management. It consists of various complementary microservices:

- *Databroker – It is implemented as an MQTT broker, with secured subscription and message passing between the IAKM agents and the IAKM server microservice.*
- *IAKM server* - its main objective is to handle the authentication of the external microservices, such as the IAKM clients pr the global AI component. Its second objective is to act as a Web server and expose an HTTP API to the AI entity.
- *Database* – it is implemented as a MongoDB and is connected to the AIKM server to save and retrieve knowledge models if locally available.

The *IAKM client* is a set of microservices provided on the client side and connected over an HTTP API to local AI workers. It has two main interfaces:

- *Northbound API – it is a MQTT API to send and retrieve data from the databroker. It also subscribes to the AIKM server.*
- *Southbound API – it is a HTTP API to expose subscribe and publish functions.*

The architecture of the IAKM (backend and client) is available in Figure 4.

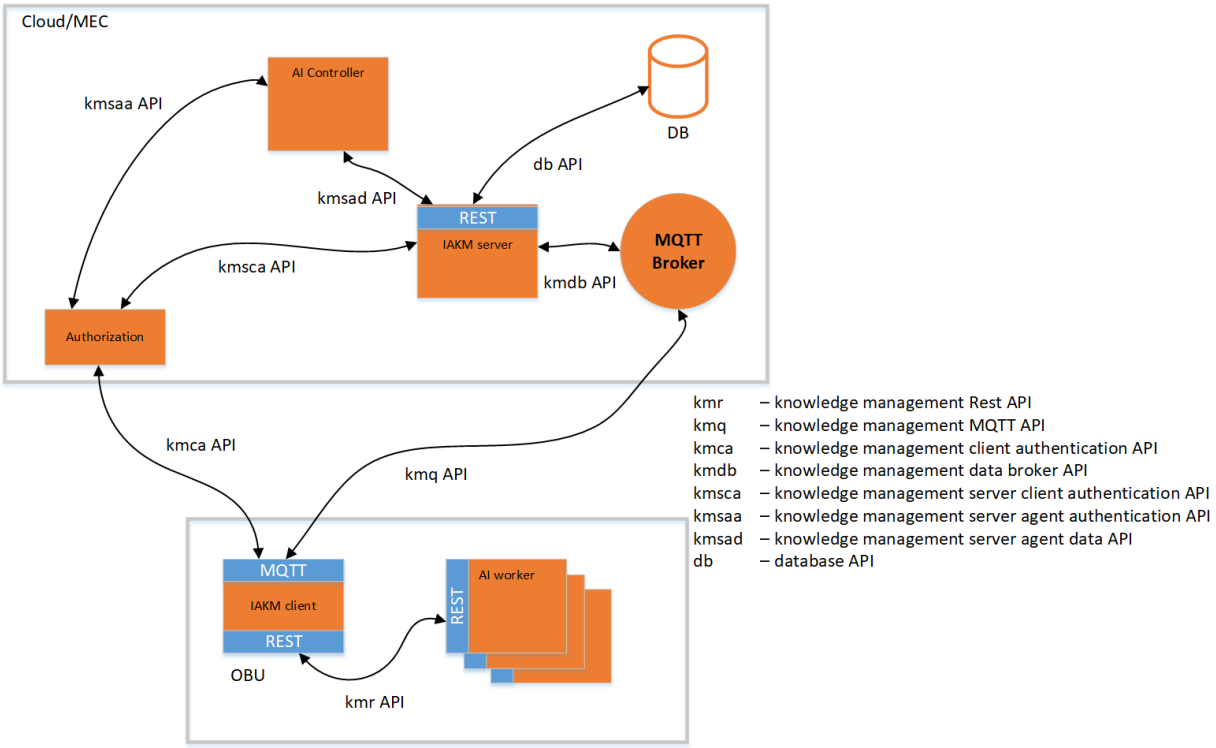


Figure 4: Component architecture of the IAKM (server and client)

2.2.1.3 IAKM MESSAGE FLOW AND INTEGRATION WITH AI COMPONENTS

Figure 5 illustrates the message flow and integration of the IAKM client and backend with the Global AI component. In this example, we considered one AI infrastructure as a FML controller and two AI agents as FML workers for two functions:

- AI knowledge retrieval for usage – one or more AI agents require the assistance of the IAKM to locate a particular AI model. In that function, the AI infrastructure is not required. If the IAKM has the requested model in its local DB it will provide it to the AI agent. If not, it will contact external IAKM backend entities to locate the AI model.
- AI knowledge training assistance – the IAKM does not train AI models but provides assistance in the training in the form of the identification of the most efficient AI agents to train a particular AI model.

Although IAKM internal message exchanges are either based on MQTT or plain socket APIs, the IAKM only exposes HTTP APIs to subscribes to a particular IAKM service, to post or get particular AI models.

On upper part of Figure 5 (for usage) we can first see that both FML controller and both AI agents subscribe to an IAKM service. Note that specific callback functions are implemented on the IAKM agent and server to provide a REST subscription API. In the example, the AI agents subscribe to a particular AI model according to a specific AI semantic. The IAKM retrieves the AI model and pushes it back to the AI agents.

On the lower part of Figure 5 (for training), we can see that the AI agents subscribe for their availability to train according to a particular environment (transmitted using an AI semantic). In turn, the FML controller subscribes to finding AI agents according to a particular context. The IAKM server makes the match and provides the AI model to the matching AI agents. In turn, after training, both agents send the updated models to the IAKM server, which pushes back to the FML controller. Once merged, the updated AI model is saved on the IAKM database.

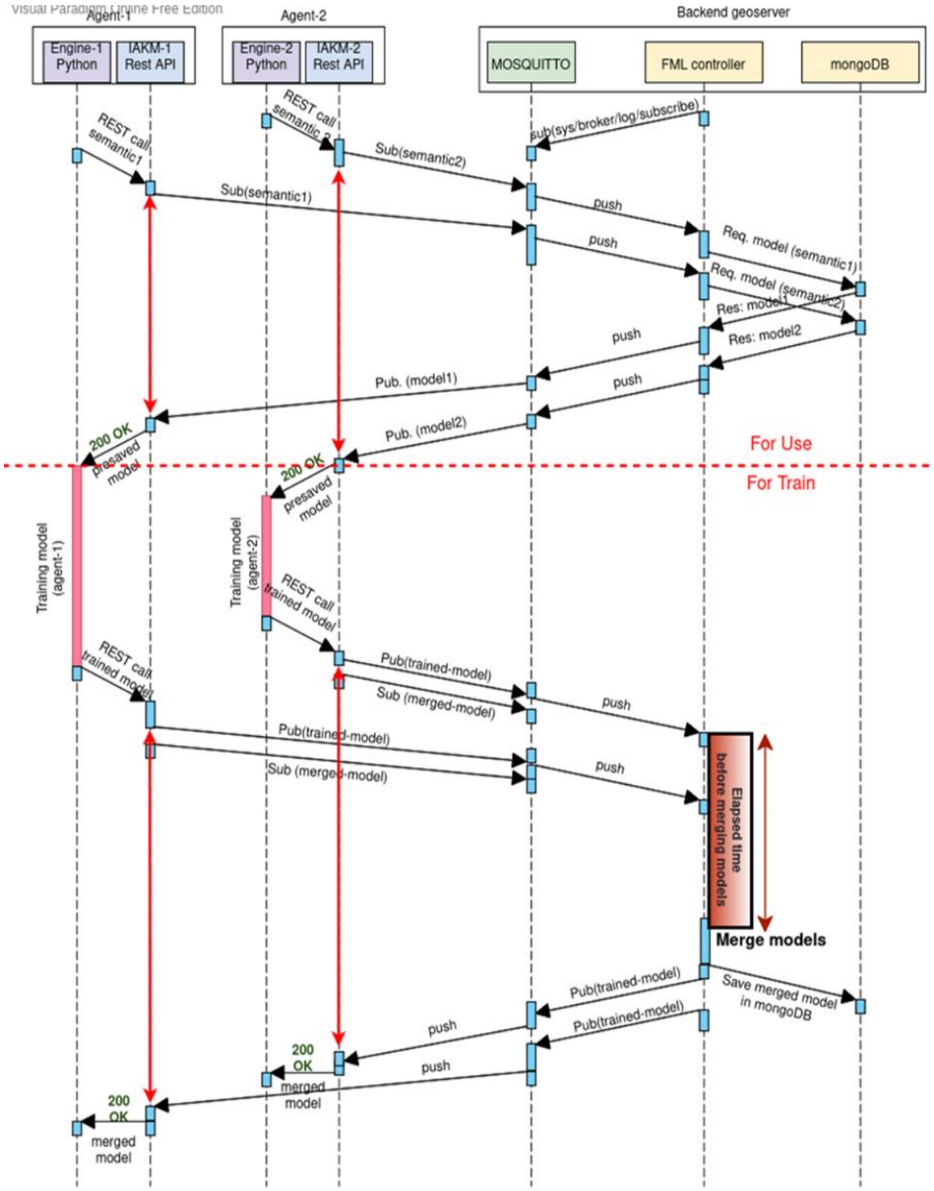


Figure 5: Process flow of IAKM assisted FML training and FML model exchange

2.2.1.4 IAKM HTTP API DESCRIPTION

Figure 6 depicts the subscription flows between the FML controller, the FML agent and the IAKM. The IAKM agent HTTP API provides subscriptions to three AI functions:

- Train – it is a subscription to ask for the assistance of other AI agents to jointly train a required model. In the case of an infrastructure FML, the IAKM will locate AI agents and provides them to the FML controller. In the case of Gossip FML, the IAKM locates AI agents itself and exchange AI models directly between AI agents.
- Use – it is a subscription to ask the IAKM to retrieve an AI model it needs to use.
- Available – it is a subscription to indicate to the FML controller that the AI agent is available for training according to the current AI context.

The IAKM server provides different services:

- Save/load - saves or loads an AI model from the DB and provides it either to the infrastructure AI or to the AI agents.
- Subscription matching – provides the matching between the subscriptions of the infrastructure AI and the AI agents.

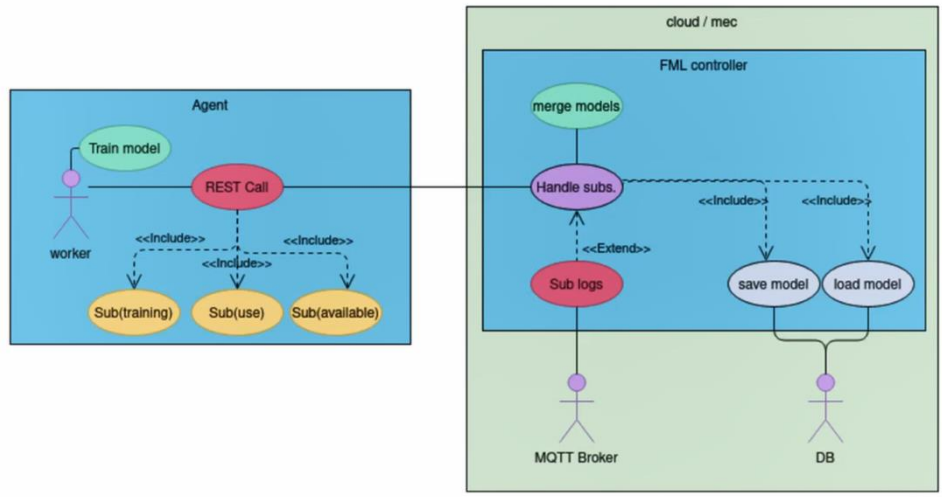


Figure 6: IAKM REST process for FML service

The subscription matching may be better seen on Figure 7. After both FML controller and workers initially subscribe to the IAKM backend (broker) using a REST Sub interface, the IAKM server matches both according to a similarity index on the AI semantic and publish the matching AI model.

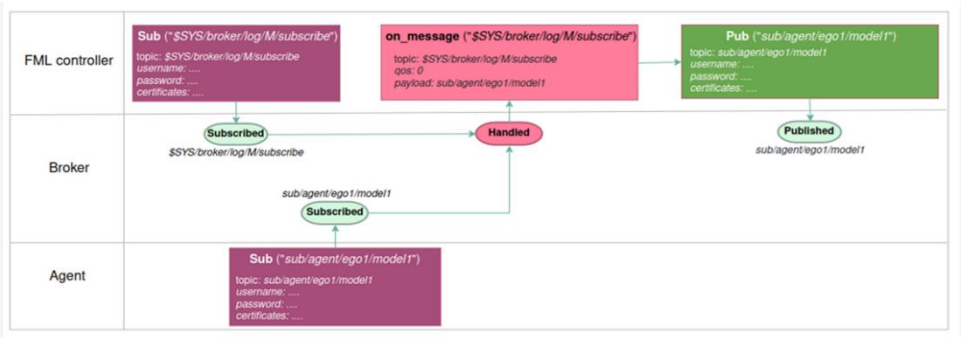


Figure 7: IAKM subscription/matching process for FML service

2.2.1.5 IAKM EXAMPLE – FML TRAINING

Figure 8 depicts the output of the IAKM assisting FML. Both FML and the FML workers are implemented are *pytorch* FML code connected to the IAKM via the exposed HTTP APIs. The selected semantics to subscribe to and post AI models are generic at that stage and for the sole purpose of showing the functionality of the back-end services. The IAKM AI semantics are currently being defined.

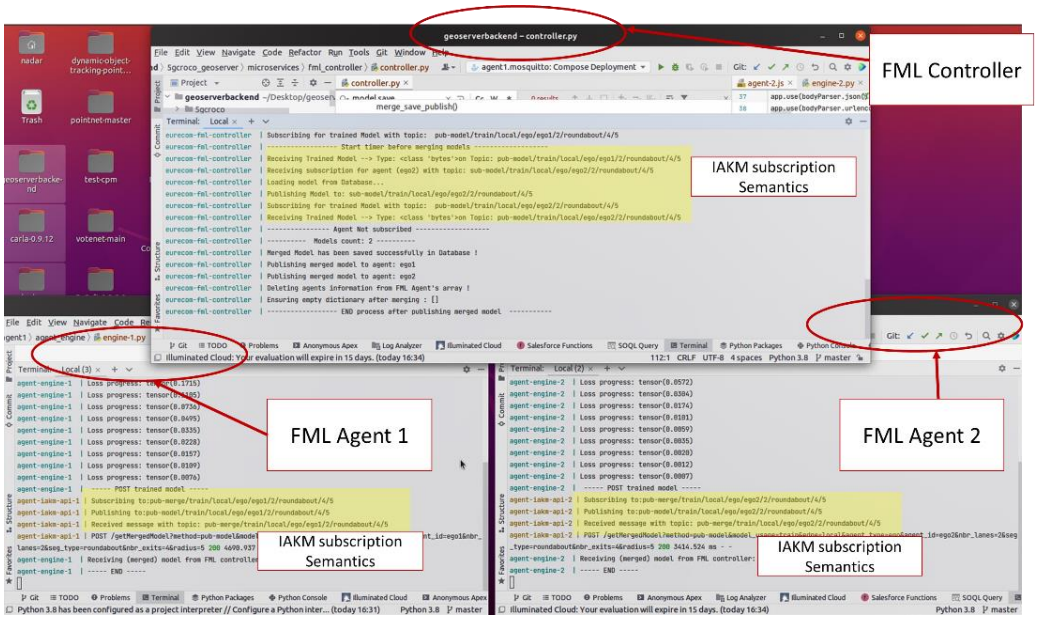


Figure 8: IAKM FML service demonstration

2.2.2 WEB-BASED IDE FOR HYPERMEDIA MULTI-AGENT SYSTEMS

The Web-based IDE for Hypermedia MAS enables Domain Experts to configure Hypermedia MAS by integrating Agent-oriented Programming with a no-code environment that permits programming Agents' procedural knowledge as well as the organizational setup of a Hypermedia MAS. This component is described in detail in Section 3.2 of the IntellIoT Deliverable D3.1 and we therefore only give a brief account of the capabilities of the currently implemented system.

Using the Web-based IDE for Hypermedia MAS, Domain Experts can select an operating workspace from the workspaces provided by the Hypermedia MAS Infrastructure, and thereby gain access to the artifacts in that workspace via W3C WoT TDs. Based on these machine-readable descriptions of the artifact interfaces, the Web-based IDE synthesizes Blockly no-code programming blocks that are tied to programming abstractions in the Jason agent programming language. By combining these blocks, Domain Experts may configure Agents to consume information from artifacts, reason upon it, and issue commands to artifacts based on their reasoning. Agents may also be configured to communicate with each other. In the current version of the Web-based IDE for Hypermedia MAS, Domain Experts may program multiple Agents simultaneously using the blocks abstraction (including authentication workflows when using interaction affordances of devices and services, which is currently limited to key-based authentication methods [Basic, API-Key]), persist and load Agent code, define and store MAS configurations, submit MAS configurations for execution, and add and remove Agents to/from a currently running MAS.

2.2.3 INTEROPERABILITY BOX

The Interoperability Box (IO Box) has been described in Deliverable D3.1. As a component in the IntellIoT framework it aims to resolve incompatibility constraints between heterogeneous devices that have either their own private semantics or are unable (from a computational point of view as these are limited resource devices) to communicate with the Hypermedia MAS and the related communication mechanisms it uses. The Interoperability Box is deployed in an intermediate node (assuming the role of a gateway) bridging IoT devices and the IoT application.

The gateway has the necessary physical interfaces to communicate with those devices and the software stack necessary to bridge these devices with the IoT application. The software stack has two main components. The first one involves the device-specific low-level drivers that represent how the local system views and accesses the device. The second component (called IO Box) provides a translation between this local view and a higher level W3C WoT TD description that the remote IoT application may use to communicate with the device. In this way, devices with protocols that do not yet have a W3C WoT TD binding become available. Once the device description has been

advertised to the HyperMAS, the communication API is defined, and the IO Box implements the specified HTTP REST API to enable the handling of the device from the IoT application. Figure 9 demonstrates this architecture.

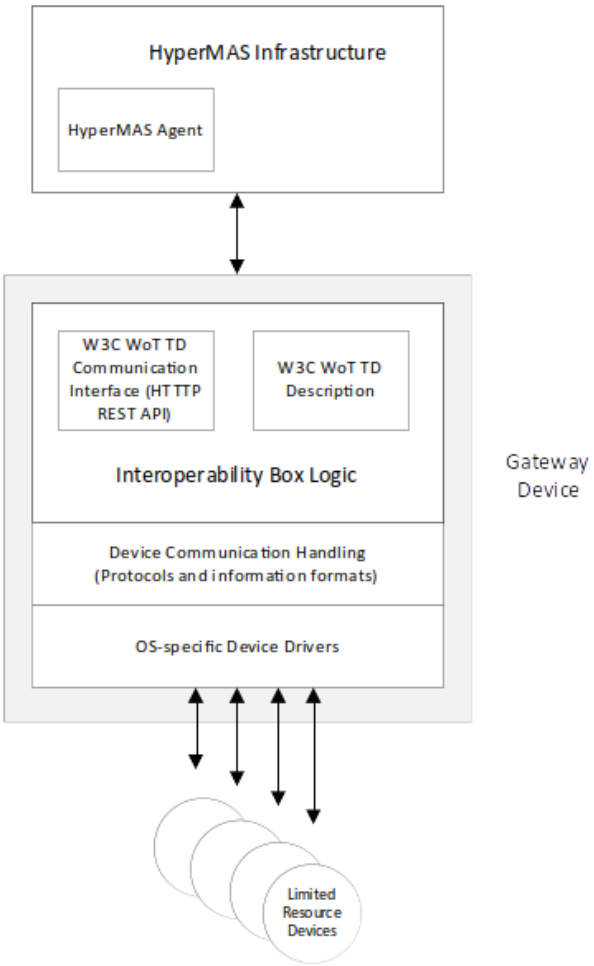


Figure 9: High-level, logical architecture of Interoperability Box

The Interoperability Box as a software component requires a Linux-capable device, with proper interfaces with all devices that have limited resources and need its services in order to communicate with the IoT application. A common example of such a device is the Raspberry Pi, which is the chosen device for demonstration purposes in IntellIoT.

The first version of the IO Box is implemented with several mocked devices for testing purposes. These are virtual devices that emulate actual physical character devices (examples of such devices are all devices that are accessed through serial interfaces, like I2C). The mocked devices that have been implemented emulate a temperature sensor and a switch. For each device, the IO Box defines a user-level driver that abstracts the device details. The user-level driver operations are mapped to the defined HTTP API to complete the communication path. A demo instance is up and running on our premises and it is reachable through the public internet. An example GET request (`curl http://147.27.38.61:3000/temperature`) from an external entity will trigger a response from the virtual temperature sensor like this:

```
{
  "degrees":-7.808579856712106,
  "unit":"celsius"
}
```

Through this simple example, an external entity (an IoT application) that does not have any specific knowledge of the device that provides temperature sensing, is able to communicate with this device and retrieve the information that it requires. The switch mock device, unlike the temperature sensor, is not only read, but can accept and respond to commands. Therefore, a PUT request can be used to alter the switch state. Supposing that the switch is in a disabled state, then a GET request (`curl http://147.27.38.61:3000/switch`) from an external entity will trigger the following response:

```
{
  "enable":false
}
```

Sending the following PUT request:

```
curl -X POST -d '{"enable": true}' http://147.27.38.61:3000/switch
```

will prompt

```
{
  "enable":true
}
```

and a subsequent GET request will verify that the switch is indeed in enabled state by returning the same response:

```
{
  "enable":true
}
```

2.2.4 GLOBAL AND LOCAL AI

In collaboration, global and local AI establish a distributed AI system, implementing Federated Learning. The main tasks of the Global AI Component are model aggregation and model evaluation. Also, in UC1 and UC3, it performs the task of resource-aware scheduling. These tasks are described below in terms of the necessary Input, resulting Output, and the Test Environment used in the development of the component. The main two roles of the Local AI are (I) to train the local AI model using training over local data and communicating with the Global AI, and (II) to make local inference utilizing the knowledge acquired throughout the network. The tasks defined under inference is use case-specific, in which, described separately in chapter 3. In this section, we also briefly describe the integration of the Global AI with the Local AI Component and the IAKM Components.

2.2.4.1 RESOURCE AWARE SCHEDULER

Depending on the available computation and communication resources, the Global AI needs to determine which Local AI components need to be scheduled to carry out local model training for a given training iteration. Here, the objective is to ensure that the accuracy of the model is improved over scheduling a subset of Local AI components.

INPUT: For a given training iteration, the size of the local datasets, the estimates of local computing power available at the devices running Local AI components, the estimated channels between the local devices and Global AI component, and available communication resources (resource blocks) are used as the input for the Resource Aware Scheduler.

OUTPUT: A Boolean vector corresponding to a subset of Local AI Components that needs to carryout local model training and model sharing during the selected training iteration.

TEST ENVIRONMENT: This is tested over a simulated environment prior to integration in Agriculture and/or Manufacturing use cases. The simulation environment is a Python development environment running on a Windows workstation. Here, the local computing power availability is modelled as a Poisson arrival process [3] and the dynamic wireless channel are generated according to Rayleigh distribution [4].

2.2.4.2 MODEL AGGREGATOR

The role of Model Aggregator is to generate a Global AI Model using the Local AI Models received from multiple Local AI components and then, share with all the Local AI Components during each training iteration.

INPUT: For each training iteration, the size of the local datasets along the Local AI Models are used as the input.

OUTPUT: The Local AI Models parameters are averaged in terms of dataset size as their weights to generate the parameters of the Global AI Model, which is the output of the Model Aggregator.

TEST ENVIRONMENT: This is implemented and tested over simulated environments prior to integration. The simulated environment depends on the use case, I.e., a Python development environment running on a Windows workstation is used for Agriculture and Manufacturing use cases while a dedicated specialized framework (which extends PySyft with relevant components) is adopted in the Healthcare use case that supports several aggregation algorithms. The UC2 approach supports as well flexible aggregation, which allows to configure the timeout and minimum number of workers required for aggregation, which is essential when learning across a large number of end-user mobile devices which can behave autonomously.

2.2.4.3 MODEL EVALUATION

As part of the federated learning process, new AI models with the aggregated updates from the Local AI need to be evaluated against the base or current Global AI model. This evaluation is performed by computing an appropriate model performance metric (accuracy, mean squared error, amongst others) on a curated dataset available at the Global AI. The evaluation will determine if the new AI model should be sent to the Local AI and replace the current Global AI model, also considering the personalization of Local AI models in certain use cases.

INPUT: A base or current Global AI model and a new, updated AI model as well as a curated dataset that serves as a reference dataset to ensure that model performance is preserved.

OUTPUT: The model performance metric from both models described in INPUT, computed on the curated dataset. A researcher will determine if there is an improvement, in which case there is a new Global AI model. The researcher will also determine if the new model should be sent to the Local AI. We will as well explore local customizations of the model and evaluate approaches towards out of distribution detection in order to automatically reject updates that are based on erroneous or low-quality data and ground truth. We will explore if this method can reject updates based on data collected for instance with wrongly positioned devices or other type of technical errors.

OR An indication of the model with the best performance metric on the curated dataset so that it automatically becomes the current Global AI model used in the system. If the indication is the new AI model, then it is also sent to the Local AI.

In UC2, considering that in production the models will be subject to strict regulatory requirements, we separate the training and validation processes from the deployment of new models in production and their use for inference. Similarly, the exploration of model customization to local data, without updating the global model, will be explored as a research concept but not automatically rolled out from the training and validation workflow into the production/model inference workflow.

TEST ENVIRONMENT: As part of the Global AI, deployed on Amazon Web Services, namely a virtual machine or a Kubernetes cluster.

2.2.4.4 INTEGRATION WITH THE LOCAL AI

As described in Deliverable D3.2, the federated learning framework used in IntellIoT distinguishes between a Global AI and a Local AI. The Global AI will receive model updates sent by the Local AI through a protected and secure API and a HTTP over TLS connection on a 5G network. As was described in Deliverable D2.3, pending model aggregation and evaluation, updated models with improved performance will be sent to the Local AI through the same API and connections. Once the Global AI determines that an improved model is available, it will push it to the Local AI as either an update to the model itself or as an update to the whole Local AI.

It is envisioned that the training as part of the federated learning process at the Local AI will be scheduled through constraints of the Edge device. However, since meaningful model aggregation and evaluation depends on the

amount and quality of updates received from the Local AI and since the constraints may be hindered by how the users go about with the Edge device, it is plausible that the training will need to be scheduled by the Global AI instead.

2.2.4.5 INTEGRATION WITH THE IAKM

The Global AI will communicate with the IAKM, as described in Deliverables 2.2 and 2.3 and in this deliverable Section 4.2.2. The details of the communication channel and related APIs will be described in Deliverable 5.4.

2.3 Human in the Loop Enablers

2.3.1 HIL SERVICE

The HIL Service is an IntellIoT specific service which mediates between HyperMAS and HIL application. Figure 10 gives an example, how HIL Service, HyperMAS, Edge Orchestrator, Robot Controller and multiple instances of the HIL application could interact on a help request raised by an AI in the manufacturing use case. Please note that these interactions are based on the actual interface definitions in the TDs that are resolved at run time. Figure 10 provides examples of what requests might be sent by the components, which are defined by the component vendors and described in Swagger specifications. The HIL service itself solely exposes the "help request" interface, which is used by HyperMAS in the example in Figure 10 and depicted in Figure 11. See https://gitlab.eurecom.fr/intelliot-project/human-in-the-loop/hil-service/-/blob/master/hil-service.yaml for details. All other interfaces referred to in Figure 10 are exposed by the components surrounding the HIL Service and defined in the corresponding component descriptions.

To test the HIL Service, we will use an HTTP client, e.g., *Postman*, to trigger the help request interface. For the interaction with other services, we will use these services where already available, or simple stubs, which can be generated from the swagger descriptions and filled with simple code to provide the expected responses.

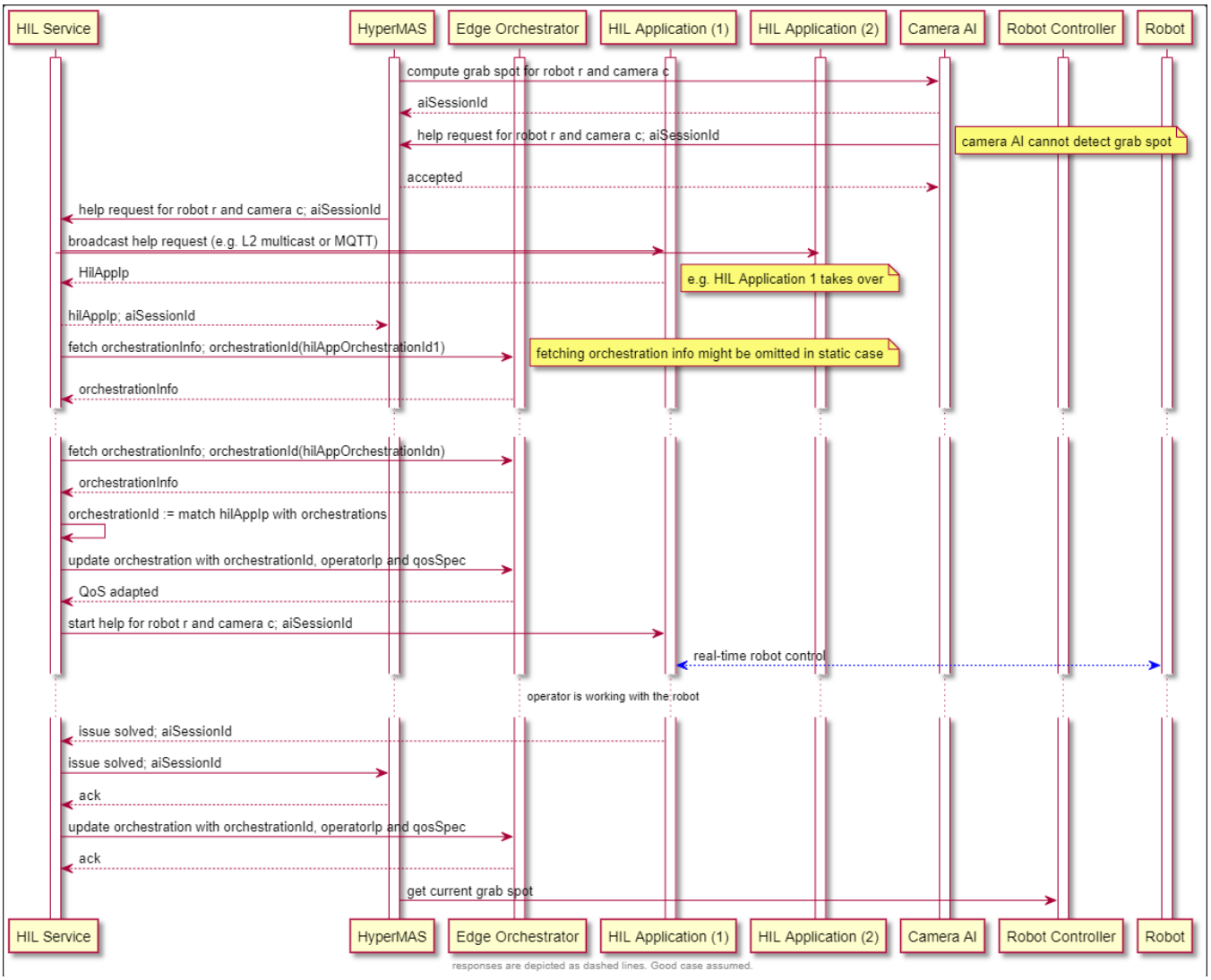


Figure 10: HIL support operation sequence chart.

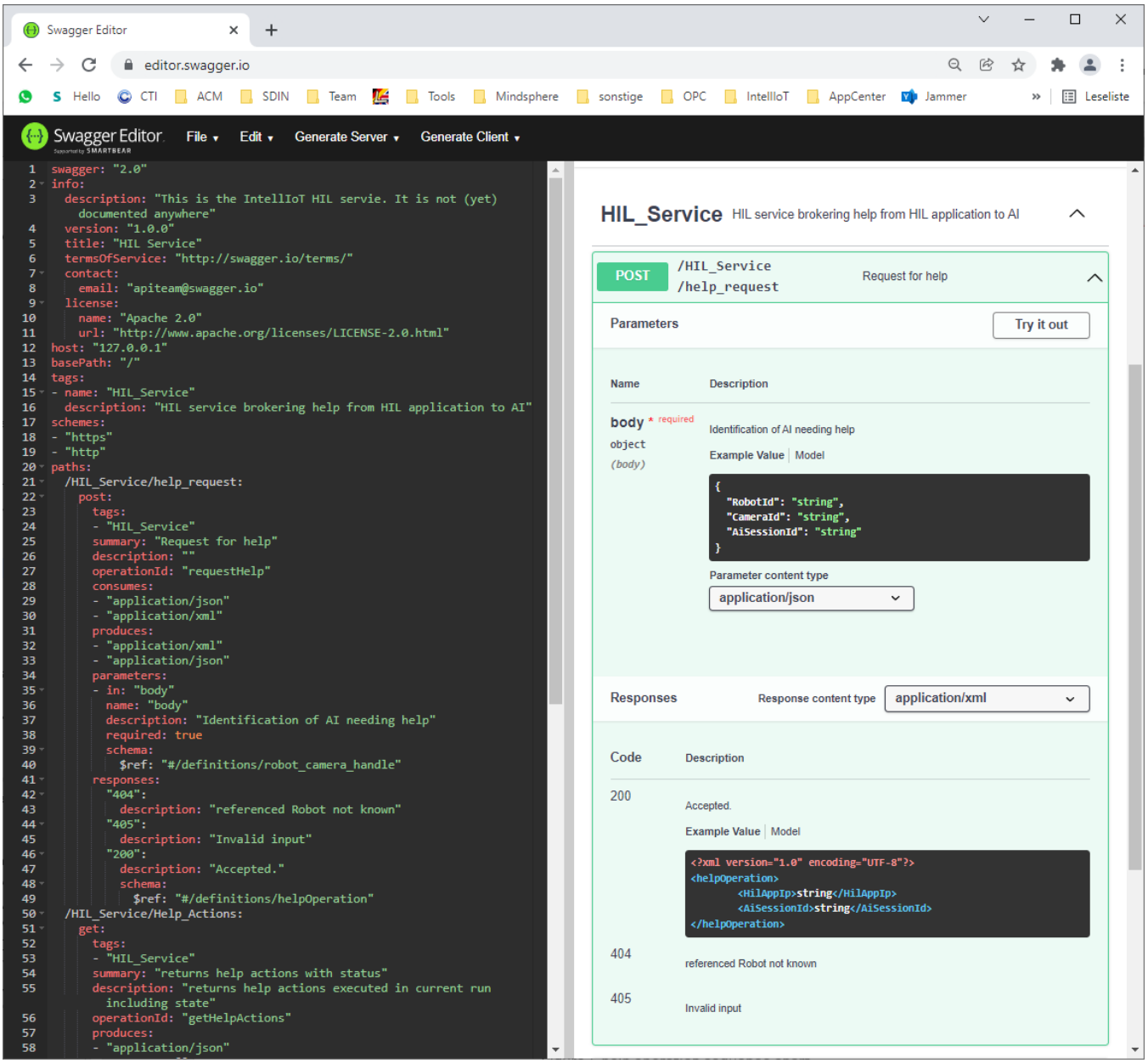


Figure 11: HIL service HTTP interface

2.3.2 HIL APPLICATION

The HIL Application will support UC 1 and UC 3 and is developed and built with the *Unity3D engine* and equipped with HOLO's *ISAR* technology. ISAR itself is a two-component solution, the Windows application running on an edge device (here referred to as "HIL Application") and the client application, installed on a smart glass (i.e., *HoloLens 2* or *Oculus Quest 2*). The solution will serve multiple purposes: the visual interface that the Operator can directly interact with through controls passed in via the *Oculus Quest 2* controllers or the *Holo-Stylus*; recipient of video and machine control data; data point generator for AI training. The core interfaces, which are not use case specific are the ISAR technology, as well as the interface to the HIL Service. To support both Use Cases a custom Windows application is being developed per use case, as well as an altered ISAR Client which connects to the HIL Service.

The HIL Application will be deployed on the edge (Virtualized Environment) controlled by the Edge Orchestrator. The concept is multiple instances of either application will run on multiple VMs. The amount can be scaled up and down, depending on the need.

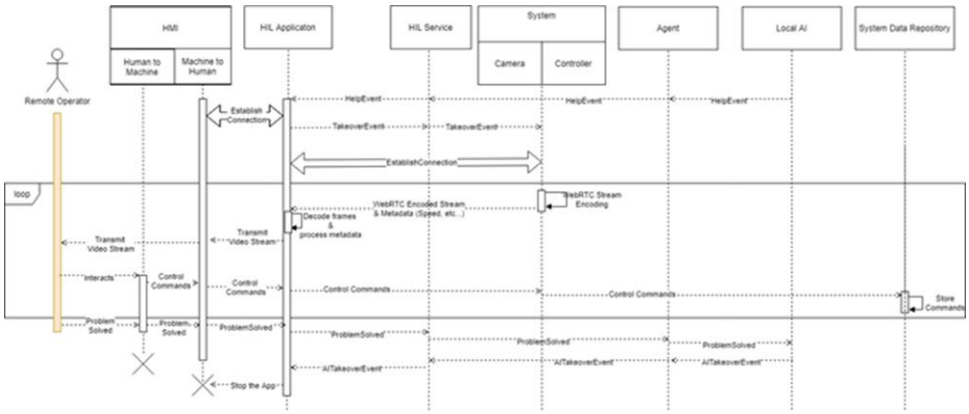


Figure 12: Human in the Loop Interfaces.

The HIL Service, which links up to the ISAR Client via MQTT and is installed on the Operator's smart glass, triggers a notification through a "help request" event. This request is initiated by the AI when it meets with an obstacle hindering the completion of its task. The available Operator will be informed of the event and will then launch the Window application and by doing so connects to the machine's interfaces. Both use cases will display a video stream from one or more cameras and allow the takeover of the machine controls. After the Operator solves the problem faced by the AI, the HIL Application triggers an "issue solved" event and sends it to the HIL Service, releasing its control over the AI. A more detailed explanation of the use case specific interfaces between the different devices will be covered in Section 6.5 of this deliverable.

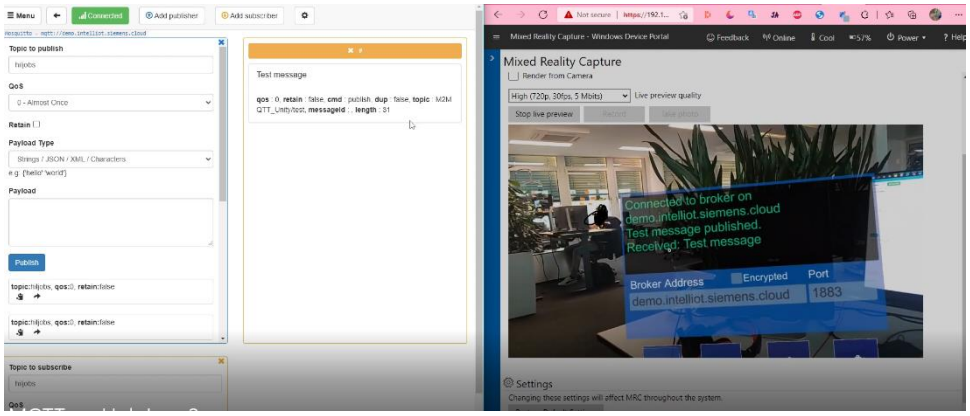


Figure 13: Unity3D application for HoloLens 2 and MQTT communication.

The used ISAR technology – which uses the WebRTC protocol – connects the HIL Application via the Signalling Server with the Client application through a peer-to-peer connection. The data channel encodes and decodes media streams, such as Audio data and Video data between both the HIL Application and the Client Application. The Client Application, however, currently is not able to transmit the data from the smart glass' cameras, or sensors, it can only display it. Amongst other functions, it supports a custom sent function, allowing the transmission of the smart glass' SLAM data, head movement as well as controller input.

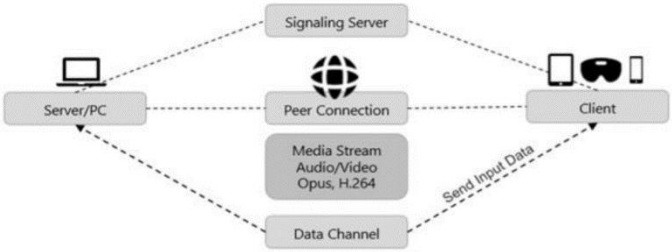


Figure 14: 14 ISAR - Server to Client interface.

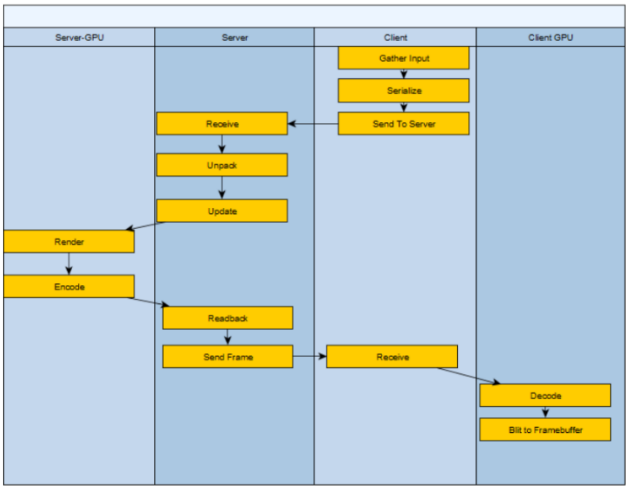


Figure 15: ISAR Interaction Diagram.

Use case specific interfaces will most likely be based on the WebRTC protocol, but function differently than ISAR. Siemens and TTC will create the interface allowing the protocol to send the video feed and send and receive machine controls. If possible, machine data such as temperature, acceleration, velocity etc. will be sent to the HIL Application as well, so it can be integrated into the UI.



Figure 16: HIL Application: Machine and Weather Data.

2.3.3 END USER GOAL SPECIFICATION INTERFACES

As part of IntellIoT's T3.1 and specific to UC1 (Agriculture) and UC3 (Manufacturing), we are developing and integrating graphical interfaces for end users. These components permit end users (i.e., farmers and customers) to specify the goal to be achieved by the IntellIoT system (e.g., "Fertilize Field 5"; "Engrave the text 'IntellIoT' in a wooden work piece") and transmit specified goals for processing to an Agent in the Hypermedia MAS Infrastructure that has been configured using the no-code configuration environment.

The end user goal specification interfaces are simple Web applications that exchange JSON-LD payloads with the Hypermedia MAS Infrastructure according to agreed-upon schemas. For UC1, the payload contains the ID and entry coordinates of the field as well as a description of the desired process to be executed, i.e., the implement to use (e.g., a "Sprayer"), the specification of the lines to move along the field (e.g., 30 lines that are spaced 5m apart) and, optionally, further parameters for the implement (e.g., flow rate). For UC3, the payload contains the desired work piece material and process (e.g., "Laser") along with a list of drawings (each with properties "Text", "position", and "size").

2.3.4 REMOTE OPERATOR DEVICES

To realize AR/VR features in UC 1 and UC3, we utilize specific devices that enable the remote operation. Thereby, the key components needed to support either use case are an edge device and a 5G connection for the almost real-time data display. To utilize the power of ISAR (i.e., the edge device), in this case a Virtual Machine, needs to meet the following specifications:

- OS: Windows 10 - min. version: 10.0.17763 Build 17763
- RAM: min. 16Gbyte
- CPU: min. Intel i7 (7th Gen)
- GPU: Nvidia Tesla with min. 5GByte Memory incl. Nvidia Grid

If these criteria are not met, GPU encoding, as well as rendering will not function. Deliverable D4.2 focuses on the 5G connection devices, so it will not be addressed here. However, what is important to mention, is the requirement of open ports. Port 9999 for the ISAR connection and any other port defined by Siemens and TTC for the WebRTC interface between the Camera / Machine controls and HIL Application.

The details about use case specific remote operating devices are given in Section 3.1.3 and Section 3.3.4 for UC1 and UC3, respectively.

2.4 Trust Enablers

The Trust Enablers pillar of IntellIoT, as defined in Deliverable D2.3 - "High level architecture (first version)", comprises innovative building blocks that individually, but also through their tight integration, realise the project's vision to provide security, privacy, and trust by design (see also IntellIoT's Objective 4: *"Enable security, privacy and trust by design with continuous assurance monitoring, assessment and certification as an integral part of the system, providing trustworthy integration of third party IoT devices and services"*). The high-level, logical view of the components within said pillar is provided in Figure 17.

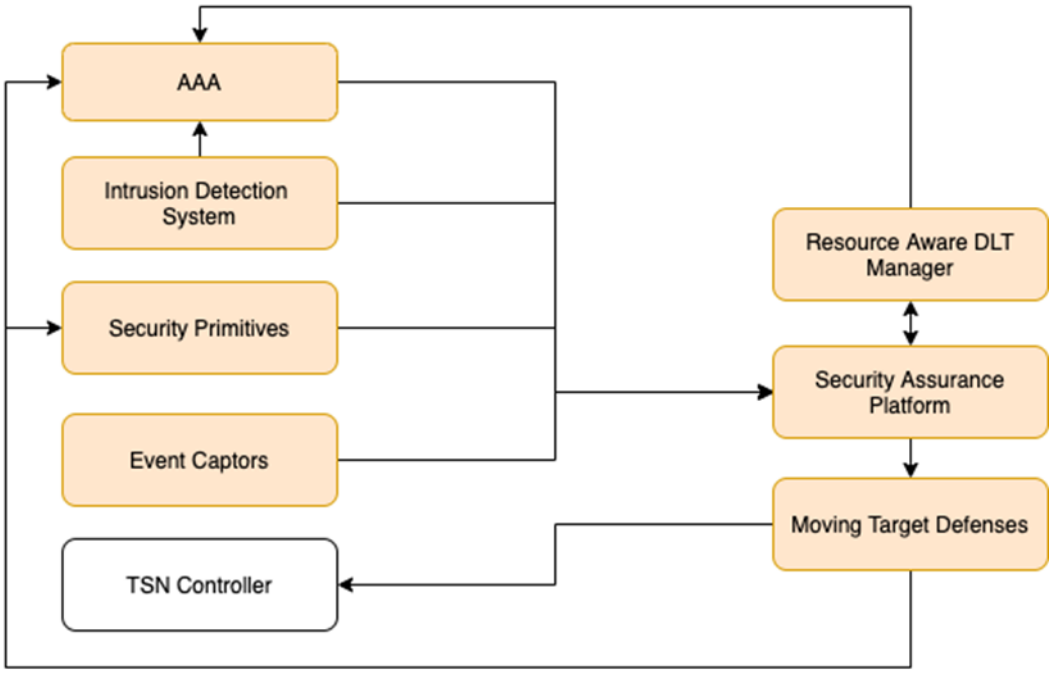


Figure 17. Trust Enablers UML logical diagram.

The Trust IDS continuously records events such as network traffic and analyses it for anomalies. They are responsible for aggregating pertinent evidence from multiple sources related to the operation of individual components, IDS computes a local trust value for each other node that it communicates with. When the trust value for a node drops below a threshold, it generates a warning. These warnings are left to be processed by the rest of the system in order to determine if an action is needed to be taken against offending nodes.

Similar to the IDS, the Event Captors (ECs) are software modules responsible for aggregating pertinent evidence from multiple sources related to the operation of individual components, as well as the overarching processes where these components are involved in, thus enabling the real-time, continuous assessment of the security posture of the IntellIoT system. ECs can be deployed across all layers of the IntellIoT architecture, from the robot arm (e.g., to monitor telemetry that may indicate abnormal activities) to device operating systems (e.g., to monitor running processes) and backend storage databases (e.g., to parse access logs or calculate uptime), transmitting their monitoring evidence to the Security Assurance Platform (SAP).

The SAP is central to the trustworthiness components and has its own queue in the message broker pipeline. SAP digest messages respectively from the IDS queue and EC queue while at the same time through appropriate REST APIs it can communicate with the Distributed Ledger Trustworthiness (DLT) manager.

Finally, the MTDs are responsible for mitigation actions once the SAP receives an Intrusion Warning from the IDS. MTDs are divided into two distinct components (the server MTD, and the client MTD) with the first having its own queue (MTD server queue) while MTD clients consumes messages from the latter queue.

While details on the design and implementation of the individual components is provided in Deliverable D4.4 – "Trust mechanisms (first version)", the subsections below will provide additional details on the testing and testbed integration of the components, but also their overall integration within the Trust pillar. The latter follows the overall integration as sketched (from a logical, process, development, and deployment perspective) within D2.3, and re-iterates and expands upon the initial testing and integration aspects presented in D4.4.

2.4.1 OVERALL TRUST ENABLERS INTEGRATION

As detailed in Section 6 of D4.4, a key characteristic that maximises the efficacy of IntellIoT's Trust Enablers is their tight integration and continuous communication through the dedicated Trust Broker. An overview of this integration is provided in Figure 18, while for details on the integration (e.g., technologies, message structure and sample messages) we refer the reader to D4.4 (Section 6).

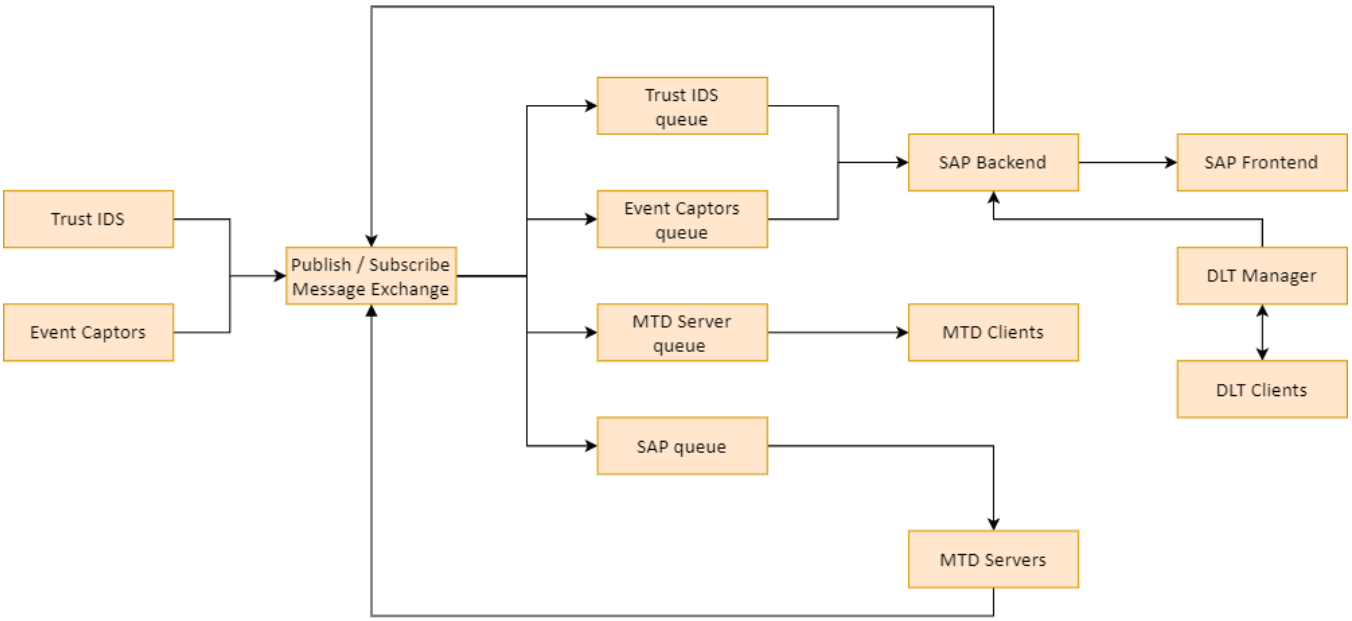


Figure 18. Overview of Trust Enablers' integration via shared Trust Broker.

From a physical (deployment) perspective, the local integration testbed used in SANL's premises is shown in Figure 19.

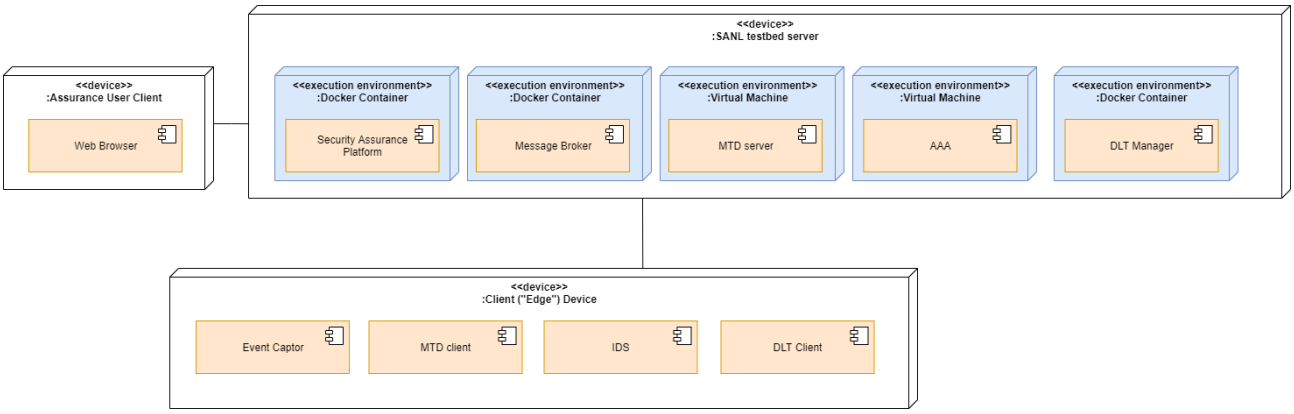


Figure 19. Deployment view of local Trust Enablers' integration testbed.

2.4.1.1 SAP-DLT INTEGRATION TESTING

Through interaction between the SAP and the DLT Manager, evidence is recorded in the Ledger in an automated manner, and the entries (transaction and block IDs) are returned to the SAP, to be provided to the SAP operator for verification (e.g., in the case of any audit). This provides an additional layer of trust for said evidence aggregated at the SAP.

Our testing of these components encompassed all different types of evidence that these interactions may involve, including:

(i) evidence generated internally at the SAP (e.g., vulnerability or dynamic testing assessment results);
(ii) evidence the SAP collects from monitoring and interacting with other Trust Enablers (e.g., Trust IDS alerts, or triggered MTD strategies), and;
(iii) Monitoring Evidence, triggered (reasoned) from monitoring the Event Captors deployed across the various layers of IntellIoT and the protected deployment.

A screenshot of a successful message exchange between the SAP and the DLT Manager, whereby SAP Assessment Results (in this case Dynamic Testing results) are recorded in the Ledger, can be seen in Figure 20.

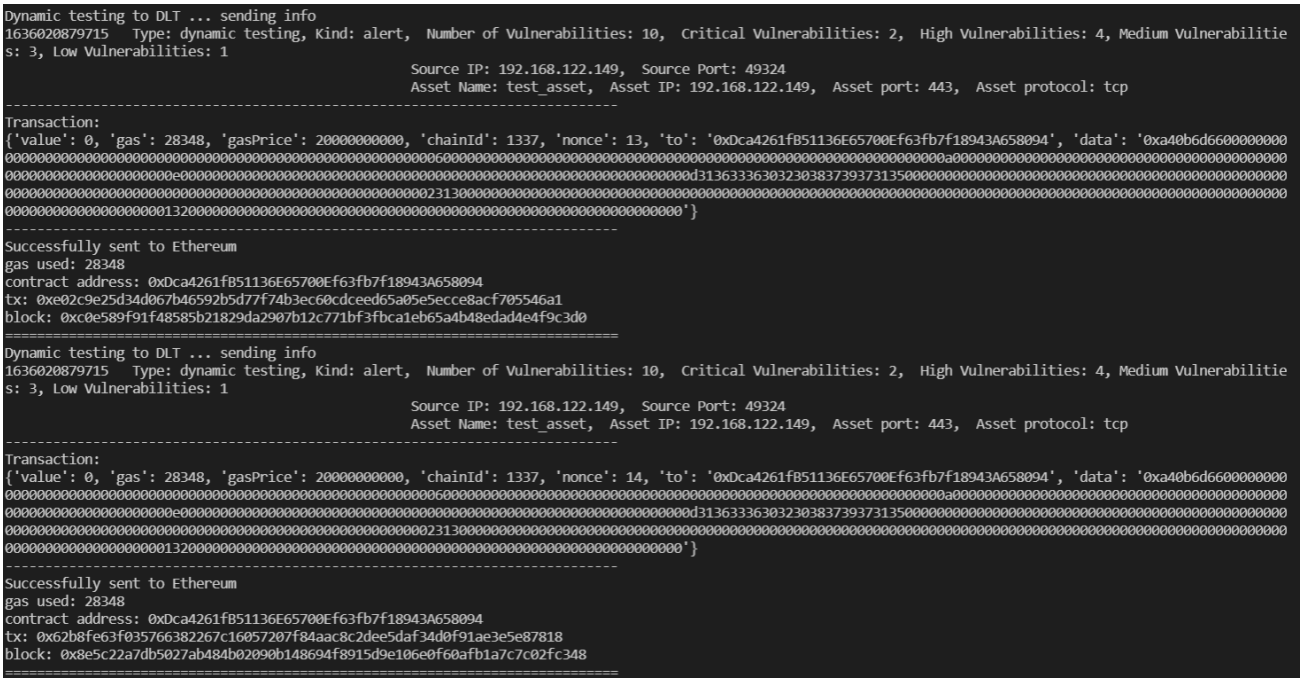


Figure 20: Recording of SAP Assessment Results in the ledger, via interaction with DLT Manager (request & response).

Similar to the above, the integration was successfully tested for all envisioned exchanges between the two components (e.g., recording to the Ledge evidence such as Trust IDS events, Monitoring events, vulnerability assessment results).

2.4.1.2 IDS, MTD & SAP INTEGRATION VIA BROKER

As mentioned, the Trust-based IDS and MTD components communicate through a dedicated Trust broker based on RabbitMQ. In the following, a description of a demo will be presented to showcase the integration of the IDS, MTD server and several MTD clients with a RabbitMQ broker. In this example, a RabbitMQ broker is initiated along with an MTD server and two MTD clients. Another node carrying an IDS component is also in the network. Upon completing the initialization and node registration phases, the node with the IDS detects that a node's trust falls below a certain threshold, and this triggers a warning message to the broker. This warning is relayed to the MTD server that isolates the offending node by sending a new network configuration message to the healthy node. This new configuration is not relayed to the offending node and therefore this node is unable to communicate with other nodes (effectively it is isolated from everyone else).

2.4.1.3 TESTING RESULTS

To realize this demo, since all software components are dockerized, we employ docker-compose and bring up four nodes:

1. Node 1 with IP address 192.168.176.2, where a RabbitMQ broker is hosted
2. Node 2 with IP address 192.168.176.3, where the MTD server runs
3. Node 3 with IP address 192.168.176.4, where a Trust IDS instance is running along with an MTD client
4. Node 4 with IP address 192.168.176.5 has the same configuration as Node 3 plus a network traffic generator

IDS monitors network traffic. To simulate a misbehaving node, a traffic generator is executed on Node 4. IDS is configured with a suitable packet rate threshold. Every time this threshold is exceeded, trust is decreased for the misbehaving node. When trust goes below 25/100, a warning is triggered.

During the demo, Node 4 starts generating traffic towards Node 3. IDS instance in Node 3 will trigger a warning message to the broker concerning Node 4 as soon as its trust value falls below the 25/100 threshold. Then the

broker relays a new network configuration to Node 3 in order to drop Node 4. Part of the logs during the demo follows.

IDS in Node 3 lowers its trust toward Node 4:

```
PacketRateFilter window timeout
192.168.176.5, trust: 23
Send warning to broker
```

The broker delivers the warning to MTD server. The MTD server drops Node 4:

```
Received warning:
  {
    "Type":"error",
    "Action":"drop",
    "Body":"client2_mtd",
    "Time":{
      "seconds":1639752428,
      "nanos":612924426
    }
  }
dropping mtd.config.client2_mt
```

This triggers a network configuration change that is only delivered to Node 3:

```
mtd.config.client_mtd -> &{
  CIDR:10.0.1.1/24
  Port:30000
  Protoc:udp4
  CipherKey:38336737335274704...
  CipherType:4
  ExtIP:192.168.176.4
  LocIP:10.0.1.1
  RoutesToAdd:map[10.0.1.1:192.168.176.4]
  RoutesToRemove:[10.0.0.1]
}
```

And now Node 4 is successfully isolated.

2.4.2 SECURITY ASSURANCE PLATFORM

As detailed in D4.4, the Security Assurance Platform (SAP) is an integrated framework of models, processes, and tools to enable the continuous assurance and certification of the security properties of the devices across the IntellIoT infrastructure.

The initial (Cycle 1) version of SAP has been deployed locally in SANL's testbed, along with a set of credentials, an "IntellIoT" organization and associated project needed to enable testing the correct operation of the different capabilities that said platform will have to support (as a standalone component) for the first version of the IntellIoT integrated platform. It should be noted that the same testbed was used for the initial integration (and testing of said integration) with the other Trust Enablers, as detailed in 2.4.1 above.

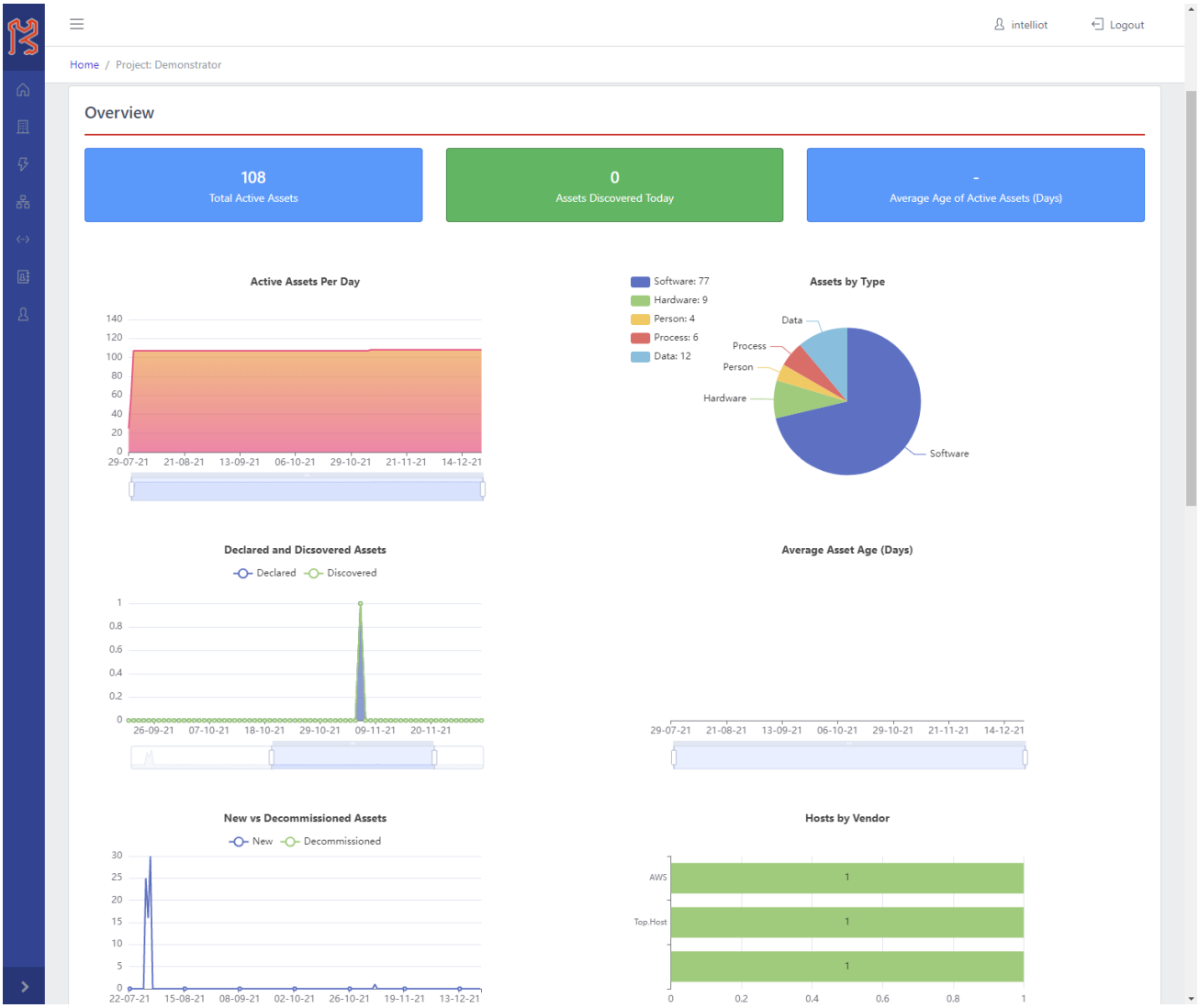


Figure 21. Overall view of "Demonstrator" testing project and related KPIs within the SAP front end

Within the "Demonstrator" project an overview of the assets, assessment results and other key performance indicators are correctly shown (see various KPIs in Figure 21 and the list of assets and assessments carried out during testing in Figure 22).

A first step in the use of SAP is the definition of the IntellIoT assurance model (see Section 2.3 of D4.4). For testing purposes, a corresponding testing model was created, including assets of all types that reflected the actual testbed environment.

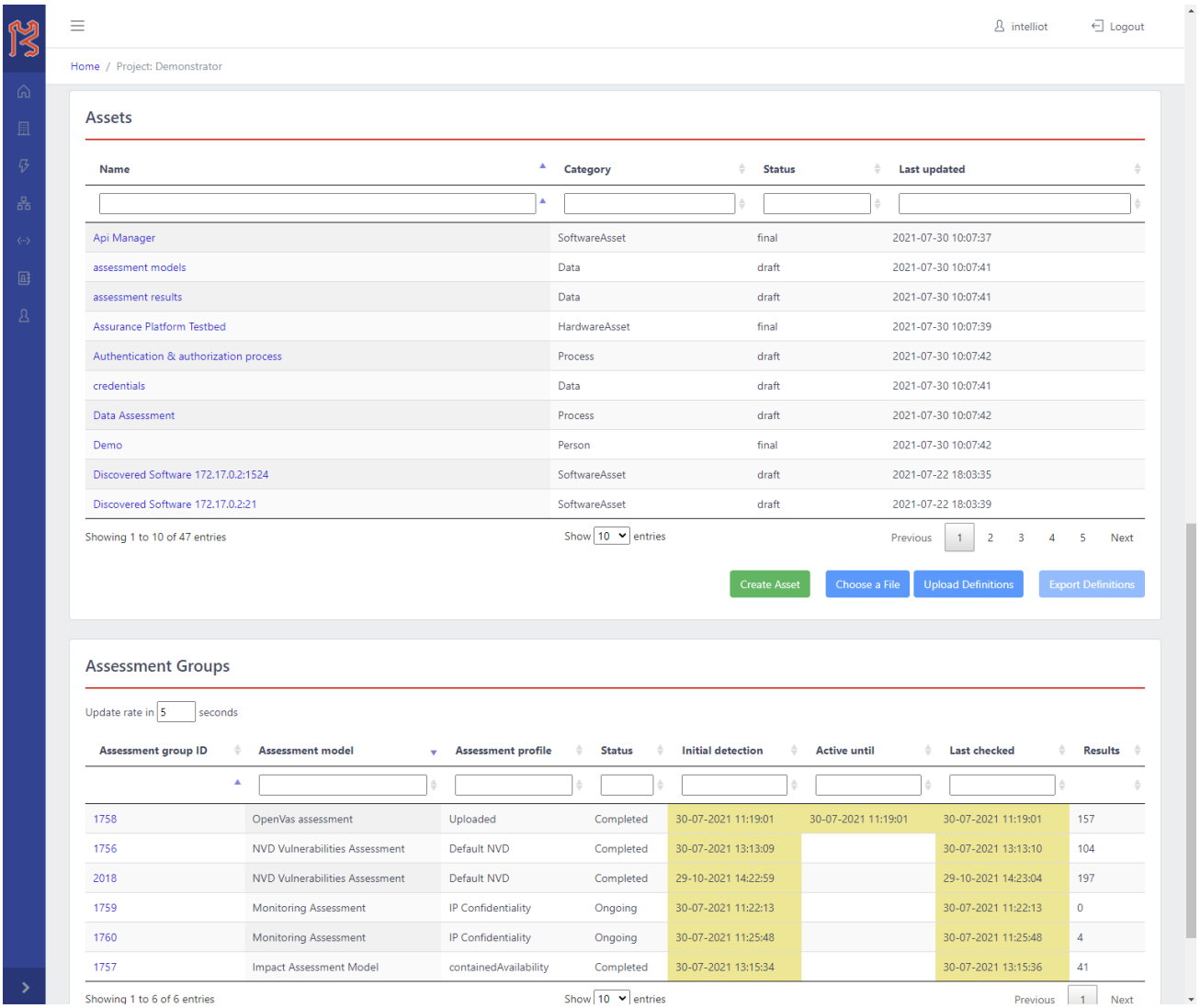


Figure 22. View of defined assets and executed assessments during SAP testing.

Furthermore, all types of assessments that will be delivered for Cycle 1 where successfully tested within the testbed environment. These, as detailed in Section 2.4 of D4.4, included:

(i) Monitoring Assessment with satisfaction and violation of pre-defined testing rules (see results in Figure 23)
(ii) Vulnerability Analysis Assessment on the testbed assets (see Figure 24 and Figure 25), and
(iii) Dynamic Testing Assessment of the testbed assets (see Figure 26 and Figure 27).

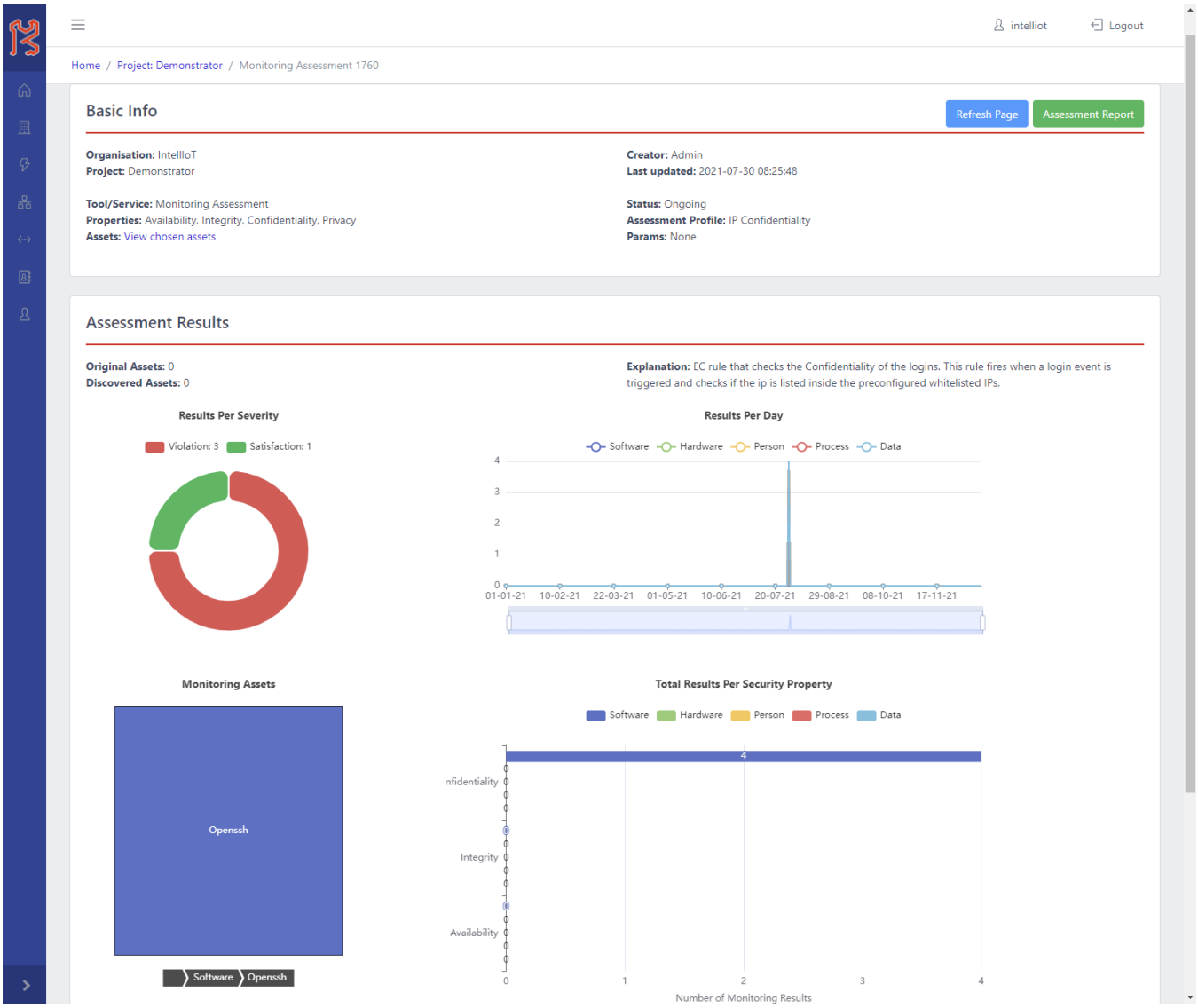


Figure 23. SAP testbed Monitoring assessment results.

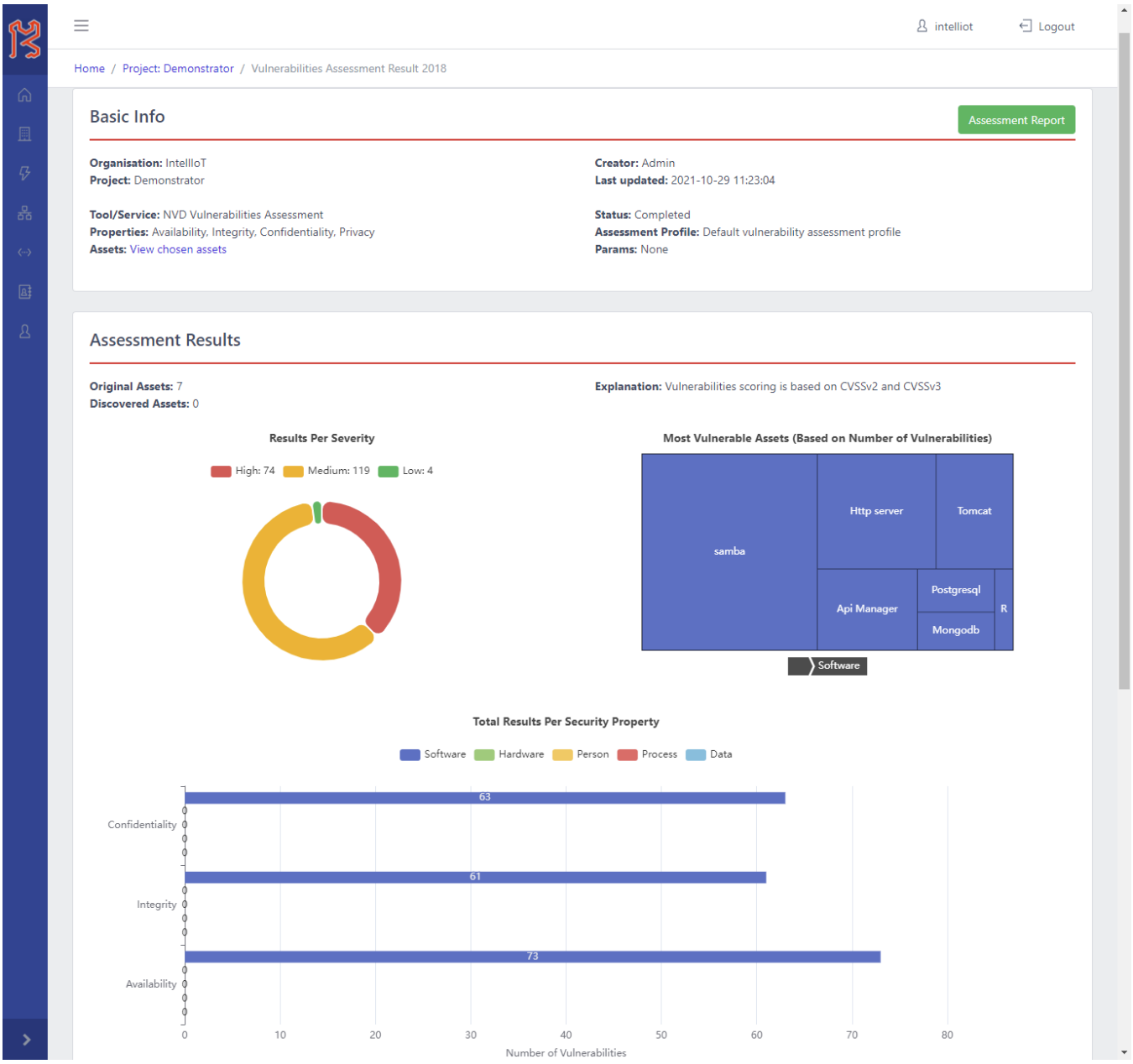


Figure 24. SAP Vulnerability Assessment carried out on testbed setup (Overview, KPIs).

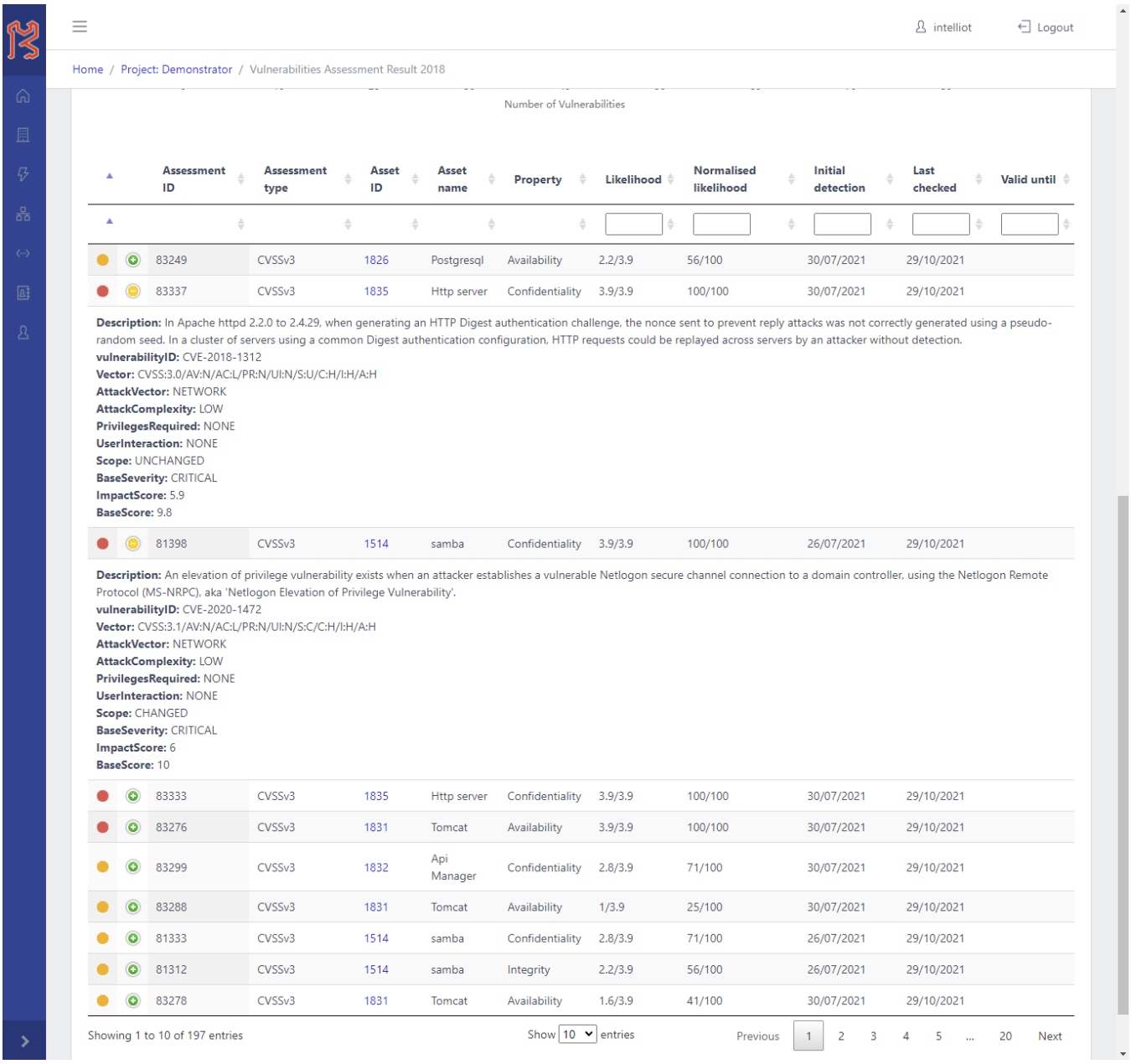


Figure 25. SAP Vulnerability Assessment carried out on testbed setup (viewing details on results).

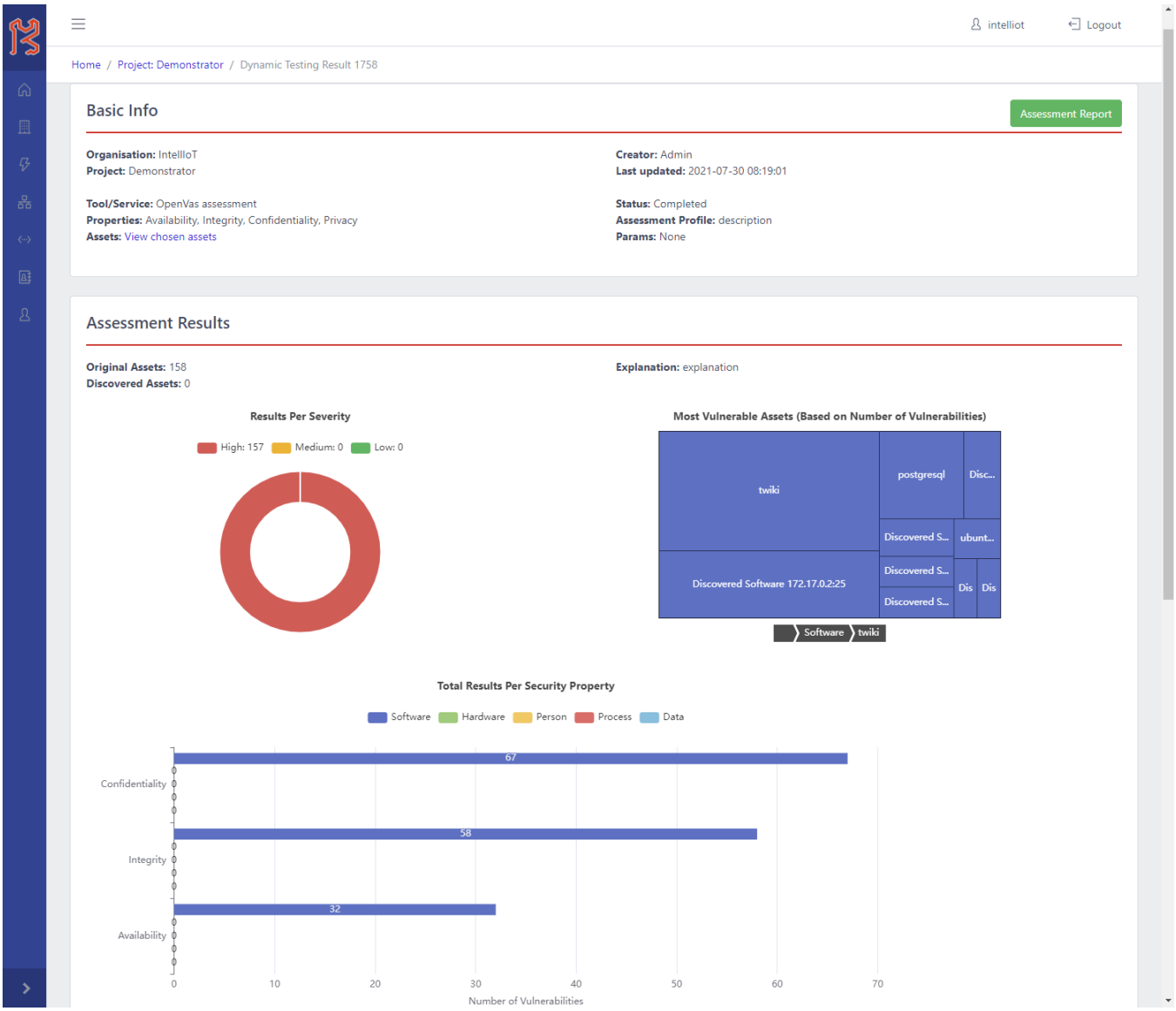


Figure 26. SAP Dynamic Testing assessment carried out on testbed setup (Overview, KPIs).

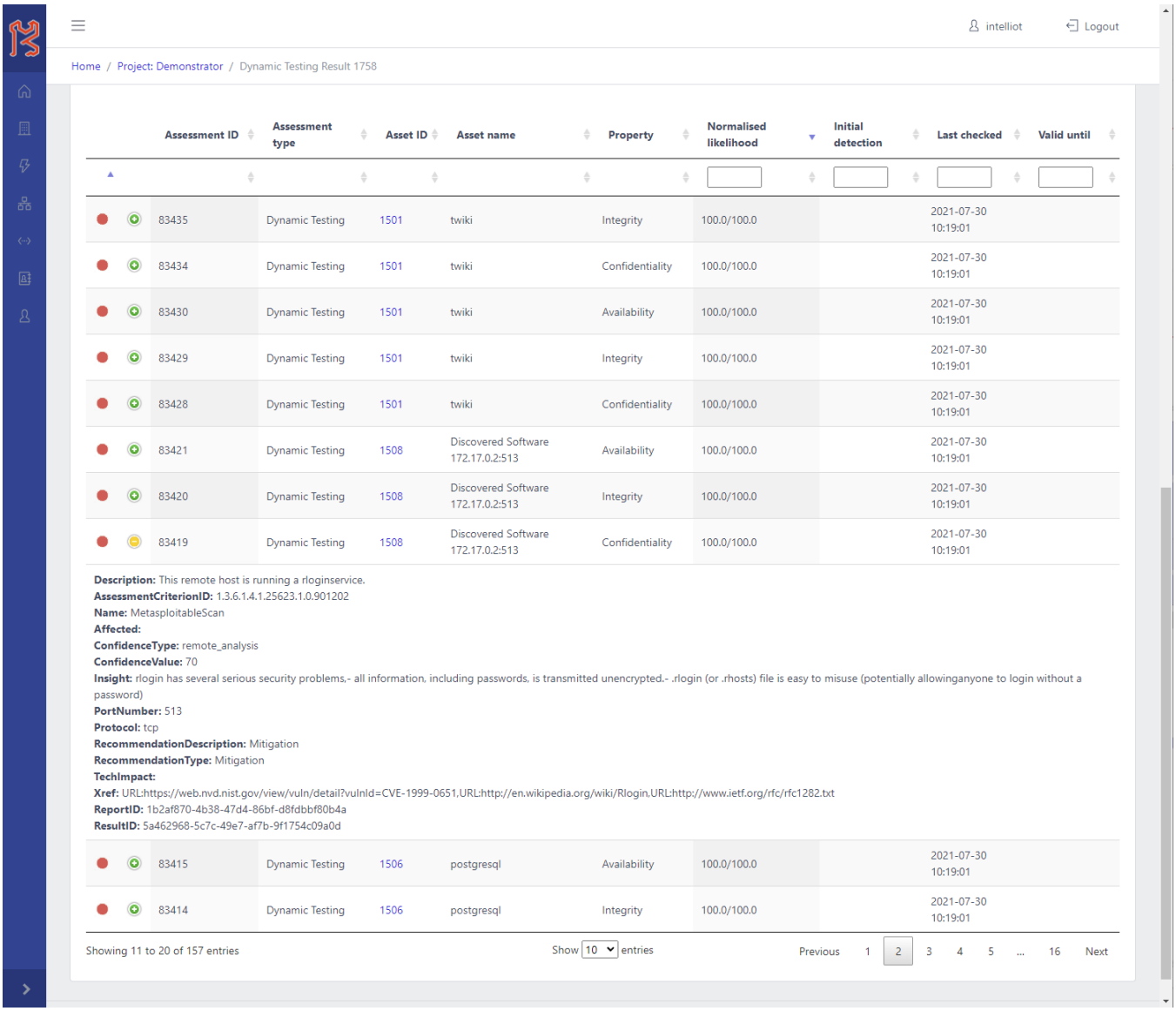


Figure 27. SAP Dynamic Testing assessment carried out on testbed setup (viewing details on results).

Finally, the horizontal capability of SAP to export reports of its assessment results in PDF format (e.g., to provide in written form the certification evidence required) was also successfully tested, as shown in Figure 28.

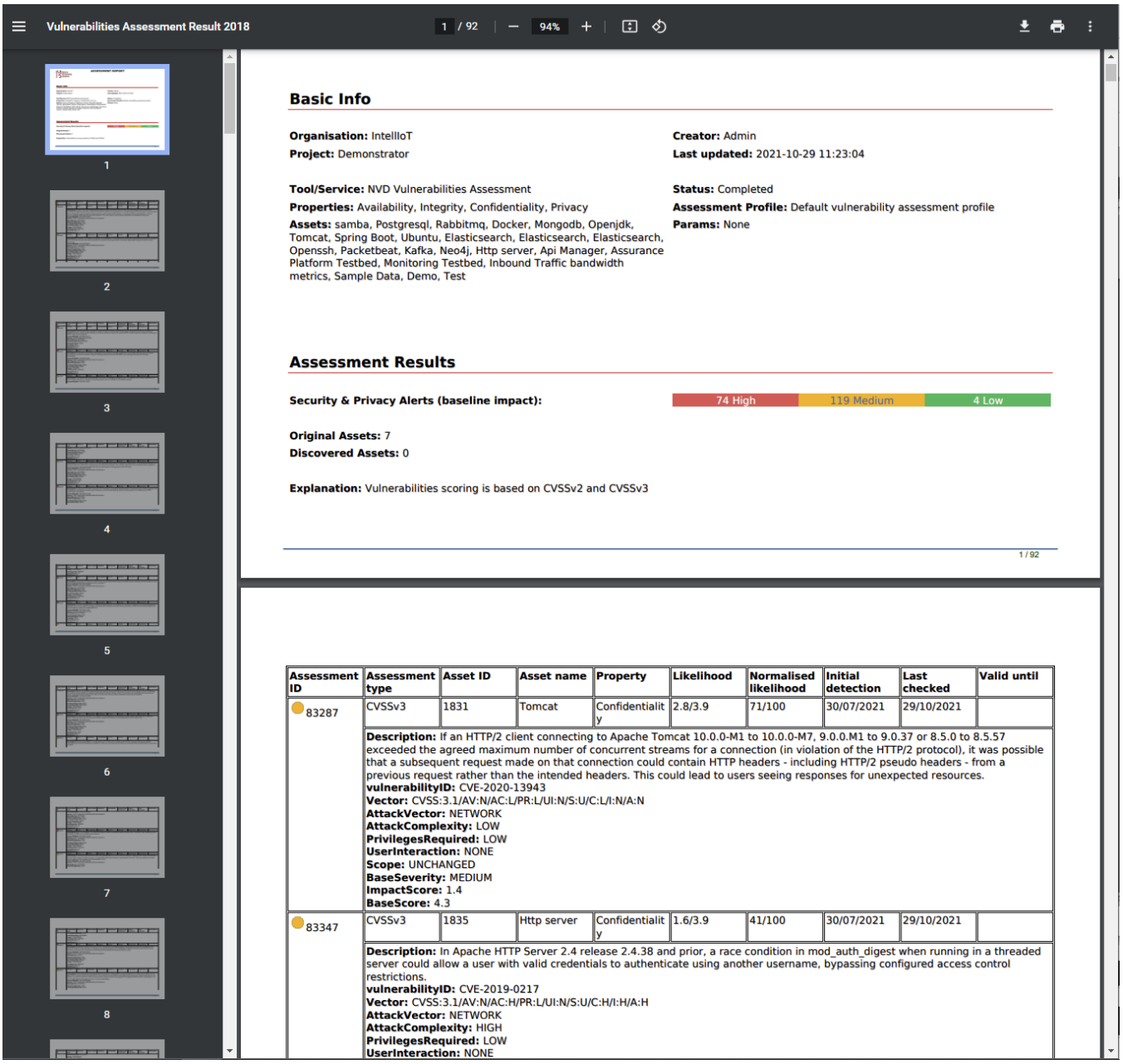


Figure 28. Exported SAP assessment report in PDF format (in this case, of a vulnerability assessment).

Further testing of the SAP that took place focused on its integration with the rest of the Trust components (as mentioned in D2.3 and D4.4, SAP is a core component that all other Trust enablers interact with), as was presented in Section 2.4.1.

2.4.3 TRUST-BASED IDS

The Trust-based Intrusion Detection System (IDS) has been described in Deliverable D4.4. It is a software component that resides on each network node of the IoT application (in the tractor, wearable device, robotic arm in IntellIoT's use cases for example) and monitors network traffic. For every node of the network that there is communication with, it builds a trust value based on certain criteria. Depending on the computed trust value, nodes are characterised as trustworthy or not and this information (warnings) is relayed to the security components of the network so that action may be triggered. More details about the software component, the rules upon which it operates, and computes trust values and its implementation architecture may be found in Deliverable D4.4, however for clarity and readability purposes Figure 29 reiterates the architecture of the tool.

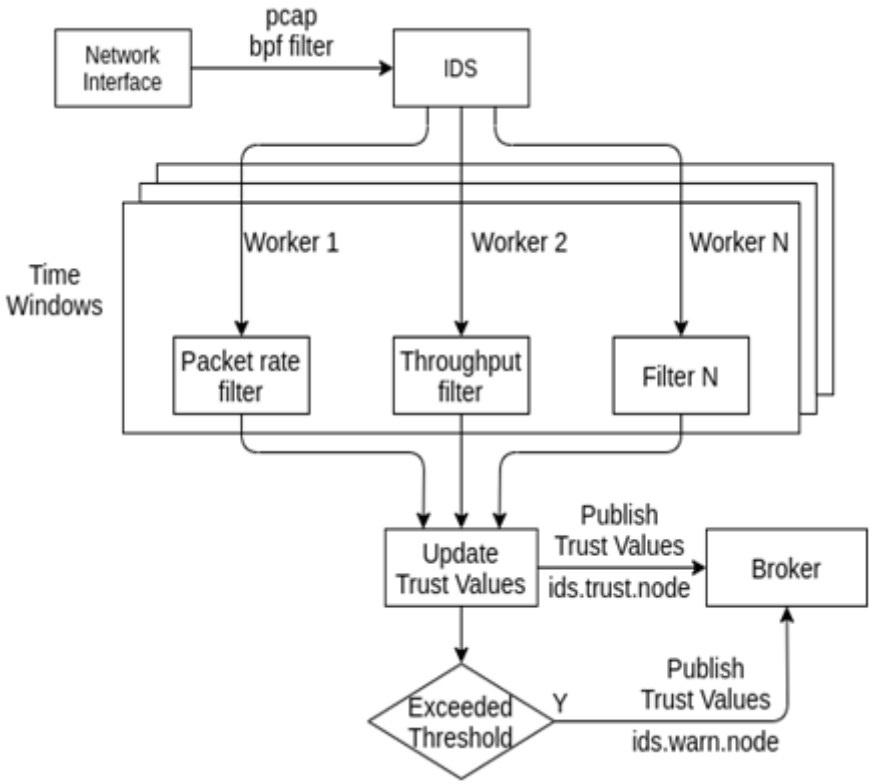


Figure 29. Architecture of the Trust-based IDS tool.

Concerning the integration of the tool with the other software components that is the focus of the current deliverable, the Trust-based IDS, as can be seen from Figure 29, uses a secure communication channel. This channel is an Advanced Message Queueing Protocol (AMQP) broker, that is used to send the trust values and warning messages to the other security components (the MTD module and the SAP). AMQP is an application layer messaging protocol that allows for various routing topologies, such as point-to-point and publish-subscribe, with TLS support to provide secure communication. This communication scheme is depicted in Figure 30.

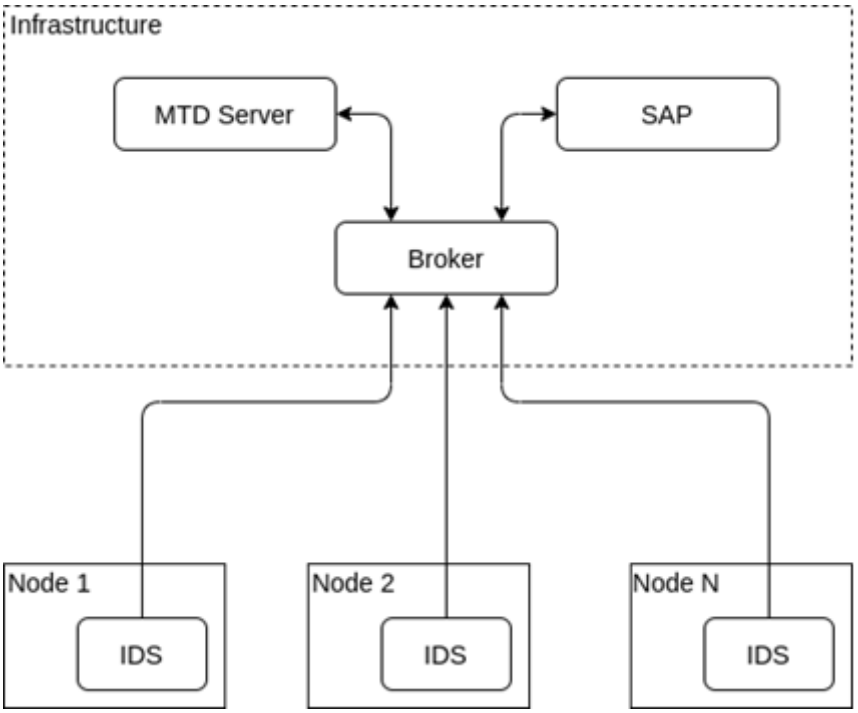


Figure 30: Communication between IDS components and the MTD and SAP modules

2.4.3.1 INTEGRATION OF IDS WITH OTHER TRUST ENABLERS

IntellIoT employs a dedicated Trust broker based on RabbitMQ and using the AMQP binary protocol as mentioned above. Through the broker, the IDS component communicates with the SAP and the MTD components.

IDS exposes two topics, in JSON format, to the broker for the rest of the security components to consume and take appropriate actions. The first one is the trust topic `'ids.trust.[node id]'`, where each IDS instance publishes its trust values for the nodes it has communicated with. An example payload is:

```
{
  "10.0.0.1": 0.0,
  "10.0.0.2": 1.0,
  ...
}
```

The second topic is the `'ids.warn.[node id]'`, where each IDS instance publishes a list of nodes that are considered untrustworthy. An example payload is:

```
{
  "nodes": [
    "10.0.0.1"
  ]
}
```

2.4.4 MOVING TARGET DEFENCES

The Moving Target Defences (MTD) component has been presented in Deliverable D4.4. The MTD comprises of a number of components that aim to provide a dynamic and constantly shifting configuration to the network

infrastructure that they try to defend. This is done either proactively (to prevent an attack) or reactively (to mitigate an ongoing attack).

While MTD can be applied in several ways, in IntellIoT, we focus on managing the network configuration and dynamically changing it to fit the needs of the infrastructure. Examples of our strategies include the periodic shuffle of IP addresses and communication ports, as well as the encryption of network traffic while changing in different time intervals (or as a response to certain events) of the encryption keys and the encryption algorithms used to counter sniffing and Man-In-The-Middle attacks.

The MTD components that have been developed are a client-server set of software components that manage the network configuration of the edge network. The MTD server is responsible for managing all the clients, handling events like warnings from the IDS or actions from the SAP and generating new configurations. The MTD clients are responsible for applying the configuration sent by the server, and based on that configuration, encrypt the traffic, and transmit it through tunnels to avoid packet sniffing and Man-In-The-Middle attacks.

The server and clients communicate by exchanging the network configuration. Parts of the network configuration are the same for all clients, and parts are specific to each client.

The MTD server proactively generates a new configuration at fixed intervals to shift the attack surface, and when a warning is received, it reactively generates a mitigation configuration based on predefined strategies to mitigate the possible attack. It is also possible for the HIL service to send actions (e.g., *isolate node X*) through the Security Assurance Platform. Configuration changes focus on network layer 3 and above, and do not interfere with the TSN controller and the resource reservations it maintains. The MTD clients are responsible for managing the network configuration, maintaining an encrypted connection between each other using the routing table sent by the MTD server and applying any changes the server sends.

At start up, each client has to register with the server by sending a registration request containing the client name and IP address through the broker. The client name is a combination of the user name of the user that owns this device and the devices client ID in the format *<username>_<clientID>* (e.g. ,TSI_sensor1, TSI_sensor2, SANL_tractor etc).

2.4.4.1 INTEGRATION OF MTD WITH OTHER TRUST ENABLERS

MTD exposes the following topics:

`mtd.registration`

`mtd.config.<client name>`

`mtd.keepaliveReq.<client name>`

`mtd.keepaliveResp.<client name>`

`mtd.alert`

`mtd.trigger`

Details on each of the four topics are provided in the subsections that follow.

2.4.4.1.1 MTD REGISTRATION TOPIC

MTD clients have write access to this topic and the server has read access. This is the topic where clients publish registration/deregistration requests. Example payloads are shown below:

```
{
  "Action": "register",
  "NodeName": "TSI_sensor1",
  "NodeIp": "192.168.1.10"
}
```

```
{
  "Action": "deregister",
  "NodeName": "TSI_sensor1",
  "NodeIp": ""
}
```

2.4.4.1.2 MTD CONFIG TOPIC

A new subtopic is created for each registered client and each client has read access only to its own subtopic. The server has write access to all subtopics and sends personalised configurations to each client, possibly skipping clients that are deemed compromised.An example payload is provided below:

```
{
  "CIDR": "10.0.0.2/24",
  "Port": 30000,
  "Protoc": "udp4",
  "CipherKey": "3863766a4c4f553166502d716a324b69",
  "CipherType": 3,
  "ExtIP": "192.168.176.5",
  "LocIP": "10.0.0.2",
  "RoutesToAdd": {
    "10.0.0.1": "192.168.176.4",
    "10.0.0.2": "192.168.176.5",
  ,
  "RoutesToRemove": [
    "10.0.0.3"
  ]
}
```

2.4.4.1.3 MTD KEEP-ALIVE REQUEST AND KEEP-ALIVE RESPONSE TOPICS

Similarly, to the configuration topic, a new subtopic is created for each registered client and each client has read access only to its own request subtopic and write access to its own response subtopic. The server has write access to all request subtopics and read access to all response subtopics. The server sends requests to which the clients must answer within a specific amount of time before being treated as disconnected. The payload is a randomly generated string that the client has to send back along with its status using its own response topic.

2.4.4.1.4 MTD AND SAP

The MTD client and server applications communicate over a secure channel provided by the Trust Message Broker. More specifically, the MTD Server consumes messages generated from IDS and takes mitigation actions when this is needed. When such an action is taken, the server needs to notify the Security Assurance Platform, as it is the component providing a holistic view of the current security and privacy posture of the system to the operators. This happens through the '`mtd.alert`' topic.

The reverse case is also needed. The Security Assurance Platform, through the event captors, can also identify attacks and trigger a mitigation action using the '`mtd.trigger`' topic.

Both of these follow the same payload logic as the '`ids.warn`' *topic:*

```
{
  "nodes": [
    "10.0.0.1"
  ]
}
```

2.4.5 AUTHENTICATION – AUTHORISATION – ACCOUNTING (AAA)

Authentication, Authorization, and Accounting (AAA) in IntellIoT is provided through a framework that allows legitimate users or applications to gain access to protected resources, thus ensuring network security. It can enforce policies (e.g., define multiple permission levels) and audit usage.

As described in Deliverable D4.4, the solution selected for IntellIoT is *Keycloak*. *Keycloak* comprises a dedicated server that centrally controls user access. This decouples local security configuration considerations and allows for better scalability and less administrative tasks. Furthermore, the servers available on the network are not tasked with storing user credentials locally, which is a preferable approach from both the security and administrative perspectives.

Authorization is the process of evaluating to what extent a user can access a resource. With *OAuth 2.0*, the following four roles are defined:

(1) **Resource Owner** (i.e., any entity that owns a number of protected resources, such as files)

(2) **Resource Server** (e.g., the actual server that keeps the files)

(3) **Client** (e.g., an application that requests access to a resource from the server is a client)

(4) **Authorization Server** (the Keycloak server itself in this case)

When a client wants to access a resource on behalf of an Owner in the Server, it first contacts the Authorization Server. The Authorization Server issues a JSON Web Token (JWT) that gives limited access to the resources, which can then be used to access the Resource Server. JWT tokens are also a standard format based on JSON, supported by libraries for various programming languages. **Authentication** is the process of identifying a user. OpenID Connect defines three roles:

(1) **End User** (i.e., entity that needs to be authenticated)

(2) **Relying Party** (the application/server that asks for a user to authenticate himself/herself before accessing it)

(3) **OpenID Provider** (the server providing the actual authentication, i.e., Keycloak in this case)

Similarly, to the authorization case, a JWT token is used. In this scenario it contains additional information in order to signify the authentication process. **Accounting** is the last functionality provided which is about monitoring authentication and authorization events for auditing. These are clustered in two categories: the login events and the admin events. Login events are generated during normal operation and cover all authentication and authorization aspects. Admin events are generated during Keycloak administration and cover any action that can be performed from within the administration web UI. Every action can be recorded and stored in the Keycloak database as well as the system log.

In the following two subsections, two approaches for the deployment of Keycloak in IntellIoT are presented. The first one is a direct deployment of Keycloak where a service can access it, as long as it talks OAuth 2.0. The second one uses an intermediate reverse proxy node in front of a service that requires authentication and authorization but does not support OAuth 2.0. In this way users can authenticate and access a service from their browser. Additional work is in progress to allow automated access without a need for a browser.

2.4.5.1 DIRECT DEPLOYMENT

To facilitate integration with the IntellIoT deployments in the different use cases, Keycloak will be used as a Docker container, allowing to be easily spawned in different use case environments. A newly created instance of Keycloak contains only an administrator account and a default Master realm. A realm is a self-contained configuration, isolated from other realms. It aggregates all the relevant information such as the users, groups, roles that comprises a secure environment.

A new realm can be created to support the IntellIoT use cases (i.e., one realm per use case deployment). We can afterwards start adding all the details that make up the realm. For example, add groups, users, roles etc. Users can be mapped to a group or a role. This mapping defines what each user can do.

A sample application is created to test and showcase Keycloak's capabilities. When the user tries to login a redirect happens to the Keycloak Authentication Server. The application must know where the Keycloak Server is and the realm it needs to get access to. By submitting the credentials, Keycloak verifies that a user exists and is authorized to use the application. The user is then redirected back to the application.

2.4.5.2 DEPLOYMENT THROUGH REVERSE PROXY

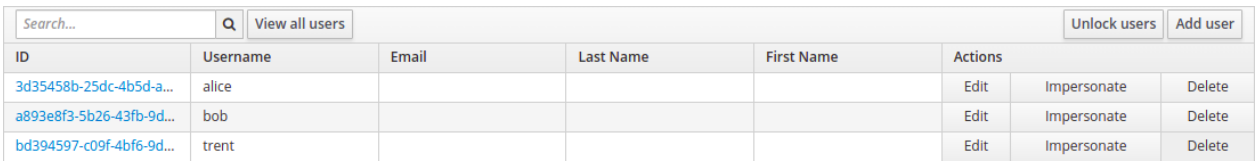
In the example showcased in the previous subsection, the application uses Node.js libraries to properly integrate with Keycloak. This is not always needed or feasible. An alternative scenario is to use a reverse proxy (Nginx in our case), to be integrated with Keycloak. Having a reverse proxy as an intermediate for services that do not have OAuth 2.0/OpenID Connect support can provide authentication and authorization capabilities for them and even simplify the whole architecture of a system.

The example that follows demonstrates this deployment scenario. For the example, we created a docker-compose environment where five containers are brought up:

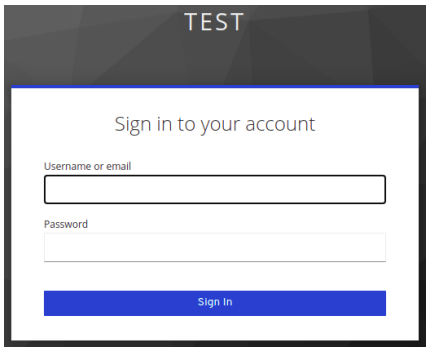
(1) Keycloak
(2) MySQL
(3) OpenResty
(4) Simple Web application (App1)
(5) Application with HTTP API (App2)

MySQL is used as the Keycloak database. OpenResty is Nginx with additional lua modules. We are interested in lua-resty-openidc which implements the OpenID Connect Relying Party (RP) and the OAuth 2.0 Resource Server (RS) functionality. App1 is just a static web page, it is used in order to showcase the different access rules between users. App2 is a basic application with an HTTP API and is needed to show that such applications can work properly in this kind of setup. Both applications have no concept of authentication/authorization.

In Keycloak we have created two users namely 'alice' and 'bob'. 'alice' is set up to have access rights only for app1. 'bob' has access rights only for app2.



These access rights are expressed as roles in Keycloak and are just associations that affect the information added to the user access token. We have also created a client named 'test' that represents the proxy. It is also associated with app1 and app2 roles. At this point Keycloak is configured. When the proxy starts it makes an initial connection to Keycloak and stores data in its cache that will be used by subsequent requests. The data includes a number of well-known endpoints based on OAuth 2.0 and the authorization method between them. If we now try to access app1, the proxy will redirect us to Keycloak for login as in the first example:




```
2021/12/17 11:51:46: openidc_call_token_endpoint(): token endpoint response:
{"error":"invalid_grant","error_description":"Session not active"}

2021/12/17 11:51:46: authenticate(): Authentication is required - Redirecting to OP
Authorization endpoint
```

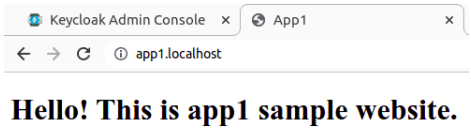
When a user logs in, e.g. *alice*, a LOGIN event is created on Keycloak describing the action needed. The first step is to authenticate the proxy (test client) so it knows a legitimate request is made and sends the appropriate user token as response. Nginx gets Keycloak response and begins processing the initial user request in order to allow or decline access to the service behind it:

```
2021/12/17 11:51:50: openidc_call_token_endpoint(): token endpoint response:
{"access_token":"...","expires_in":300,"refresh_expires_in":1800,"refresh_token":"...","
token_type":"Bearer","id_token":"...","not-before-
policy":1640173296,"session_state":"43a0828c-b23d-4328-92ed-
cd89c7d0317a","scope":"openid app1"}
```

Since the user is legitimate and the request scope includes app1, our proxy passes the initial request to the appropriate service:

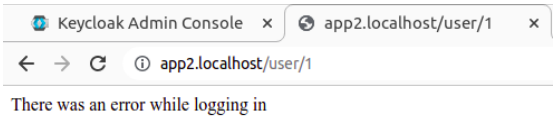
```
2021/12/17 11:51:50: authenticate(): OIDC Authorization Code Flow completed ->
Redirecting to original URL (/)
```

App1 receives the request and responds:



When alice tries to access app2, we validate the scopes available for alice and the proxy denies access:

```
2021/12/17 11:51:56: Scope is not allowed, client: 192.168.176.1, server:
app2.localhost, request: "GET /user/1 HTTP/1.1", host: "app2.localhost"
```



The same happens for bob when accessing app1:

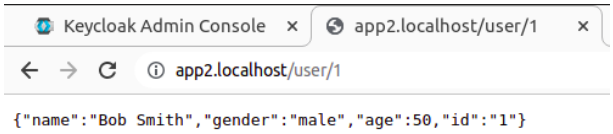
```
2021/12/17 11:53:23: Scope is not allowed, client: 192.168.176.1, server:
app1.localhost, request: "GET / HTTP/1.1", host: "app1.localhost"
```

But when bob tries to access app2, proxy allows the redirect:

```
2021/12/17 11:53:51: authenticate(): OIDC Authorization Code Flow completed ->
Redirecting to original URL (/user/1)
```

And app2 responds:

```
2021/12/17 11:53:51 Received a GET request
2021/12/17 11:53:51 Response body: {"name":"Bob
Smith","gender":"male","age":50,"id":"1"}
```



2.4.6 DISTRIBUTED LEDGER TECHNOLOGIES

There are various types of Distributed Ledger platforms including public DLTs such as *Ethereum*, *Bitcoin*, *IOTA*, *Solana* and private DLTs such as *Quorum* and *Hyperledger Fabric*. A detailed comparison and evaluation are given in Section 2 of Deliverable D3.4.

In a cross-domain project like IntellIoT, we exploit the Ethereum Blockchain platform for the use cases due to the flexibility and convenience to integrate with different components in heterogeneous environments. Ethereum is a distributed public blockchain network that focuses on running the programming code of any decentralized application. Specifically, *Ethereum* is a platform for sharing information across the globe that cannot be manipulated or changed. Ethereum has its own cryptocurrency, called Ether (ETH), and its own programming language, called Solidity. The decentralized applications on the network are called *Đapps*. Practically, Ethereum provides a convenient platform for development and smart contracts system to integrate with FL. We run the Ethereum network via *Ganache* which is a personal blockchain for rapid Ethereum distributed application development for first phase integration.

The general communication workflow of DLT network is shown in Figure 31:

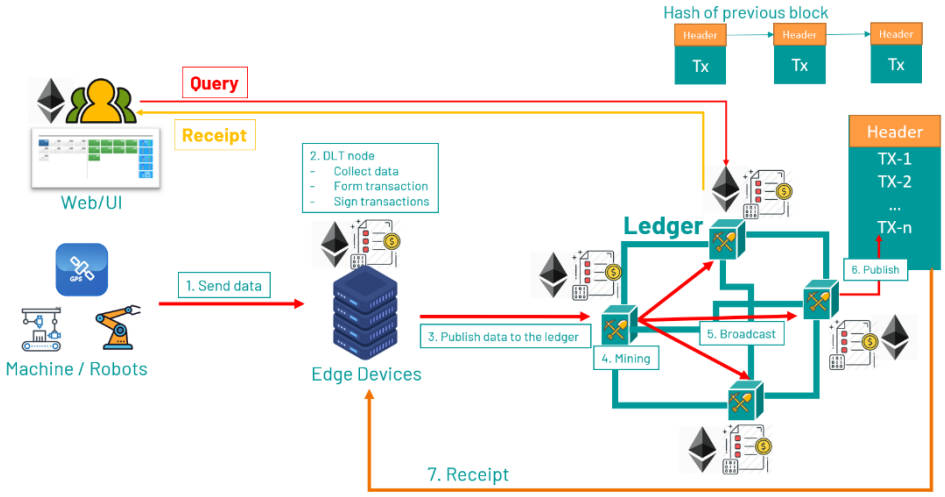


Figure 31 General communication architecture of Ethereum DLT network.

The implementation includes two parts called *DLT-manager* and *DLT-clients*. The DLT-managers require more resources (e.g., storage and computing capacity) to do verification and validation than *DLT-clients*, so that DLT-mangers usually are implemented in edge devices or powerful servers. Meanwhile, the *DLT-clients* are implemented in resource-constrained devices such as IoT devices, and sensors.

The minimum implementation of DLT-managers and clients are described as below:

	DLT Managers	**DLT clients**
Storage	4 GB RAM minimum with an SSD	1 GB RAM minimum with an SSD

Computing	CPU with 2+ cores	N/A
Connectivity	Wire/Wireless	Wire/Wireless
Interfaces	JSON RPC	JSON RPC
Operating System	Linux, Windows	Linux, Windows
Smart Contracts	Solidity	N/A

Table 1: Minimum requirement for installation of DLT components.

In the first phase of integration, we intend to build a simple framework to integrate with components from different partners, for example, the security and information of vulnerable data from TSI, GPS data from sensor of tractors and record these data to tractor controllers, and activities of UR5 Robot arms. We implement an Ethereum private Blockchain by using *Ganache* as *DLT-managers* to simulate a distribute ledger.

The successful implementation of Ethereum DLT private network is demonstrated as Figure 32 below. Each component in the DLT network is assigned an account including a private key and public key. Public key stands for digital identity and broadcast for other nodes, while the private key is used to sign transactions and kept secret. The "gas fee" is requirements for executing transactions in the network.

```
ganache          | Available Accounts
ganache          | ==================
ganache          | (0) 0x2E6d29B17a526334843141F8BEf0f0424760199e (~100000000 ETH)
ganache          | (1) 0xC4D62b4DC0749127b63FB070966c25Ac381c1944 (~100000000 ETH)
ganache          | (2) 0xDF1Db5C73747dB8deD48A3a25C17Af71FBdF1E89 (~100000000 ETH)
ganache          | (3) 0x4988518828cA3aE1E2a3E4dbcce623A40B249102 (~100000000 ETH)
ganache          | (4) 0x936cc17ac0766aE0b5f02Fa1F6BacAE30591313b (~100000000 ETH)
ganache          | (5) 0x5a6571a19e0e444411afb29E8566ba93417C7Ee3 (~100000000 ETH)
ganache          | (6) 0xaA445Bbe0970D210615A37109C734dffBc15f867 (~100000000 ETH)
ganache          | (7) 0xEFAA828E2d2Ca6317fE56a31E44C72390F854006 (~100000000 ETH)
ganache          | (8) 0x53866B55bDc66Ce7a30c20C39F105290C0559aCF (~100000000 ETH)
ganache          | (9) 0x2e94d4AA531df3943Bc09e3eEAa80358a02ad663 (~100000000 ETH)
ganache          |
ganache          | Private Keys
ganache          | ==================
ganache          | (0) 0xf90c16e1628661bc15769b76c8266343ec35db482239904b0ca1bd90c8fbca01
ganache          | (1) 0x7968992c32f33408f0abc9ab42aa95d77fb52524c2e0c62d1adef42840d3d502
ganache          | (2) 0xb492c13c93edc695916522966262ad4dc1c7f64a22b138504c5b250a1c484203
ganache          | (3) 0xb22df97c876895cec01b8ad70b00482e2201dbecc61482a9c1e44544436b1b04
ganache          | (4) 0x69ce909787a29af568092a6d1cb693a3b4510c6d4eebd7d01762aa5f4d824105
ganache          | (5) 0x867f938edd9a5598b40bcf2c68a3106c0fe7486c1a6d50ba9a5171dc865b5306
ganache          | (6) 0xe1921a270bd9751ccc2d04896c7f62cbb5d9cd5dc3195de3fc2d29c483f38507
ganache          | (7) 0x03e88e467120938d05ca0a2677a9a25080e1a16093655bdd3c8fd8c43bd4ba08
ganache          | (8) 0x3b270a195f592a505ce94186793ababb27a33a65a572bc9e7b06766fb6626b09
ganache          | (9) 0x6418670e00dbdb0a6e9bd99f1521f7294f58af0995c7a0534c2021fcd486e710
ganache          |
ganache          | Gas Price
ganache          | ==================
ganache          | 20000000000
ganache          |
ganache          | Gas Limit
ganache          | ==================
ganache          | 6721975
ganache          |
ganache          | Call Gas Limit
ganache          | ==================
ganache          | 9007199254740991
ganache          |
ganache          | Listening on 0.0.0.0:8545
```

Figure 32 Implementation of DLT managers.

After building the DLT infrastructure for recording data and making transactions, we need to build distributed applications which execute autonomously in the network to satisfy the requirements and agreement of involved participants. The smart contract is written in the Solidity language and running in the DLT network. An example of a smart contract is shown in Figure 33 below:

```
// SPDX-L// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.22 <0.6.0;

contract owned {
    address owner;

    constructor() internal {
        owner = msg.sender;
    }

    modifier onlyOwner {
        require(msg.sender == owner);
        _;
    }
}

contract mortal is owned {
    function destory() public onlyOwner {
        selfdestruct(msg.sender);
    }
}

contract IoTSmartContract is mortal {
    event records(address indexed _from, bytes time, bytes temp, bytes hum);
    event led(address indexed _from, address indexed _to, uint8 color);

    function add_records(
        bytes memory time,
        bytes memory temp,
        bytes memory hum
    ) public {
        emit records(msg.sender, time, temp, hum);
    }

    function control_led(address _to, uint8 color) public {
        emit led(msg.sender, _to, color);
    }
}

```

Figure 33: Smart contract of recording sensing data.

A typical smart contract consists of various information, for example, the owner of the devices, data, and predefined rules by system administrators, agreements between involved components in the system. Based on the scenario and specific requirements, the content of a smart contract will be written and run autonomously in the network. Figure 34 shows an example of recording sensing data from IoT sensor devices to the distributed ledger. The code of smart contracts is migrated to the distributed ledger

```
ruffle_suite   |    > contract address:    0xDca4261fB51136E65700Ef63fb7f18943A658094
truffle_suite   |    > block number:        7
truffle_suite   |    > block timestamp:     1633947517
truffle_suite   |    > account:             0x2E6d29B17a526334843141F8BEf0f0424760199e
truffle_suite   |    > balance:             99999999.975605839999999999
truffle_suite   |    > gas used:            361904 (0x585b0)
truffle_suite   |    > gas price:           20 gwei
truffle_suite   |    > value sent:          0 ETH
truffle_suite   |    > total cost:          0.00723808 ETH
```

Figure 34 The migration of smart contracts.

In order to communicate with smart contracts, Ethereum provides an API which allows DLT clients to connect and communicate with smart contracts based on the predefined rules. DLT-nodes which want to publish data to the distributed ledger or query smart contracts for doing a task, first need to generate a DLT transaction format and sign it with JSON-RPC form as:

```
curl --location --request POST 'localhost:8545/' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc":"2.0",
    "method":"eth_sign",
    "params":[
        "0x9b2055d370f73ec7d8a03e965129118dc8f5bf83",
        "0xdeadbeaf"
    ],
    "id":1
}'
```

Figure 35 Signing a transaction.

After signing transaction, the transaction can be sent to the distributed ledger via *sendTransaction* as below:

```
curl --location --request POST 'localhost:8545/' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc":"2.0",
    "method":"eth_sendTransaction",
    "params":[{
        "from": "0xb60e8dd61c5d32be8058bb8eb970870f07233155",
        "to": "0xd46e8dd67c5d32be8058bb8eb970870f07244567",
        "gas": "0x76c0",
        "gasPrice": "0x9184e72a000",
        "value": "0x9184e72a",
        "data": "0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"
    }],
    "id":1
}'
```

Figure 36: Sending a transaction from clients to distributed ledger.

The metadata of transaction shows the address of sender and destination, the gas fee to make the transaction, and the (hashed) data. Clients or customers can query the data recorded in the distributed ledger by using the transaction hash or transaction ID as well as the whole block ID if needed.

```
curl --location --request POST 'localhost:8545/' \
--header 'Content-Type: application/json' \
--data-raw '{
    "jsonrpc":"2.0",
    "method":"eth_getTransactionByHash",
    "params":[
        "0xb2fea9c4b24775af6990237aa90228e5e092c56bdaee74496992a53c208da1ee"
    ],
    "id":1
}'
```

Figure 37 Querying transaction details.

Integration with the SAP is shown in 2.4.1.

2.5 Infrastructure Management Components

2.5.1 EDGE MANAGEMENT

2.5.1.1 EDGE ORCHESTRATOR

The Edge Orchestrator is described in D4.1. It is a software component, which uses multiple other components. In the following, we describe the test and integration concepts for those components.

The Edge Orchestrator itself follows a microservice architecture. The single services provide specific system functionality. For example, we have services for hosting the API, the database, the input processing and many other system services. We are currently not following a strict test process to guarantee the functionality, however,

we support different test cases, which include relevant components tests, so the integrity of the system can be checked. How, these are integrated into a CI/CD pipeline is not yet decided.

The API, which is relevant to integrate the Edge Orchestrator with other systems within IntellIoT is available as an OpenAPI specification. Client applications can use this document to generate client code for various programing platforms. The specification can also be used to generate test servers to facilitate the development of client applications. The software suite of swagger.io provides different tools to enable integrations. For example, with swagger codegen code can be generated for client as well as server applications. If required, a mock server of the Edge Orchestrator can be deployed as a test platform. It has to be kept in mind that a productive system, accessing physical devices, cannot be provided for general tests. Accordingly, we will schedule live tests during IntellIoT's integration workshops, e.g., during a plenary meeting.

Other components, as the Edge Manager, will be exclusively used by the Edge Orchestrator in a remote-controlled fashion. As this component is not used by external systems, no distinct integration or test plans are foreseen. The testing will be covered by the test concepts of the Edge Orchestrator.

Edge Devices are managed by the Edge Manager and won't need specific integration and testing concepts either.

Edge Apps are specific components of IntellIoT. Test concepts of Edge Apps will be in the responsibility by the Edge App owners. However, we provide an Edge Deployment Tool, which takes specifications as input and outputs a configuration, which can be directly deployed within Edge Infrastructure. This means, if Edge Apps are on-boarded or updated, the process of deployment on a live system is widely automatized.

2.5.1.2 EDGE MANAGER

The Edge Manager is an off-the-shelf component used by the Edge Orchestrator. Its correct functionality will be tested during testing of the Edge Orchestrator.

2.5.2 DYNAMIC NETWORK MANAGEMENT

2.5.2.1 TSN CONTROLLER

The TSN controller is an edge application realized in Python and JavaScript. It receives so-called communication service requests, also known as stream requests, via its northbound interface. In IntellIoT's manufacturing use case, this HTTP-based interface will mainly be used by the edge orchestrator. For test purposes, any HTTP client, e.g. Postman[7], can be used, see e.g. Figure 38. Additionally, a Web UI is available for convenient manual testing, see e.g. Figure 41 and Figure 42. A description of the interface is given in in https://gitlab.eurecom.fr/intelliot-project/networking/tsn-controller and valid requests are available as a Postman collection (Figure 38) and in Swagger (Figure 39).

To test the TSN controller, a testbed is available at the Siemens site in Munich Perlach. The TSN controller gathers topology information via LLDP and netconf and presents topology information in the Web UI (Figure 40) and via its HTTP interface (Figure 41, Figure 42). In both cases, information about available resources and resources already reserved for embedded streams are given. As a first step to test the TSN controller, we manually check the correctness of this information in the Web UI and via the HTTP interface.

[7] https://www.postman.com/

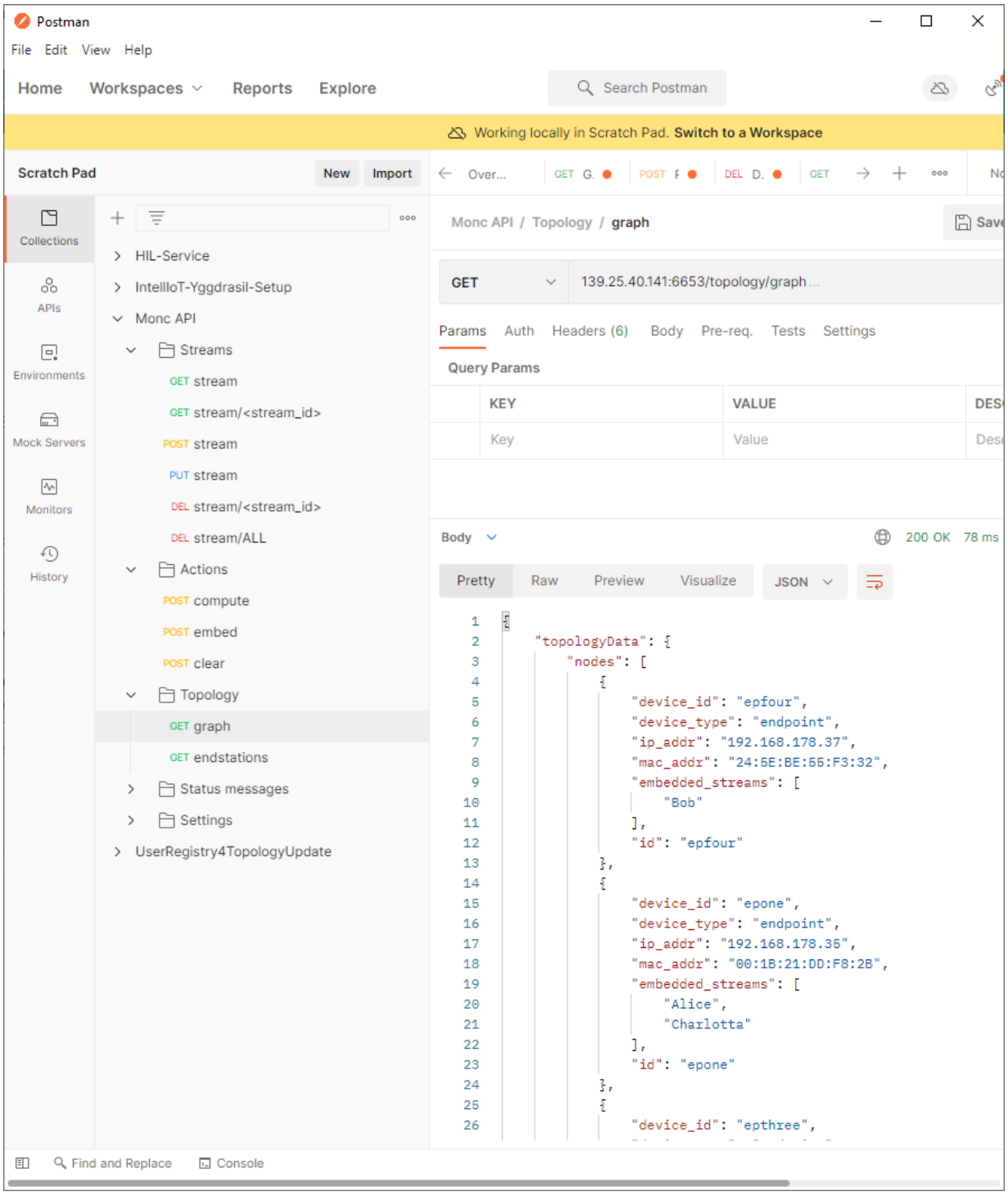


Figure 38: Topology view with stream information in Postman.

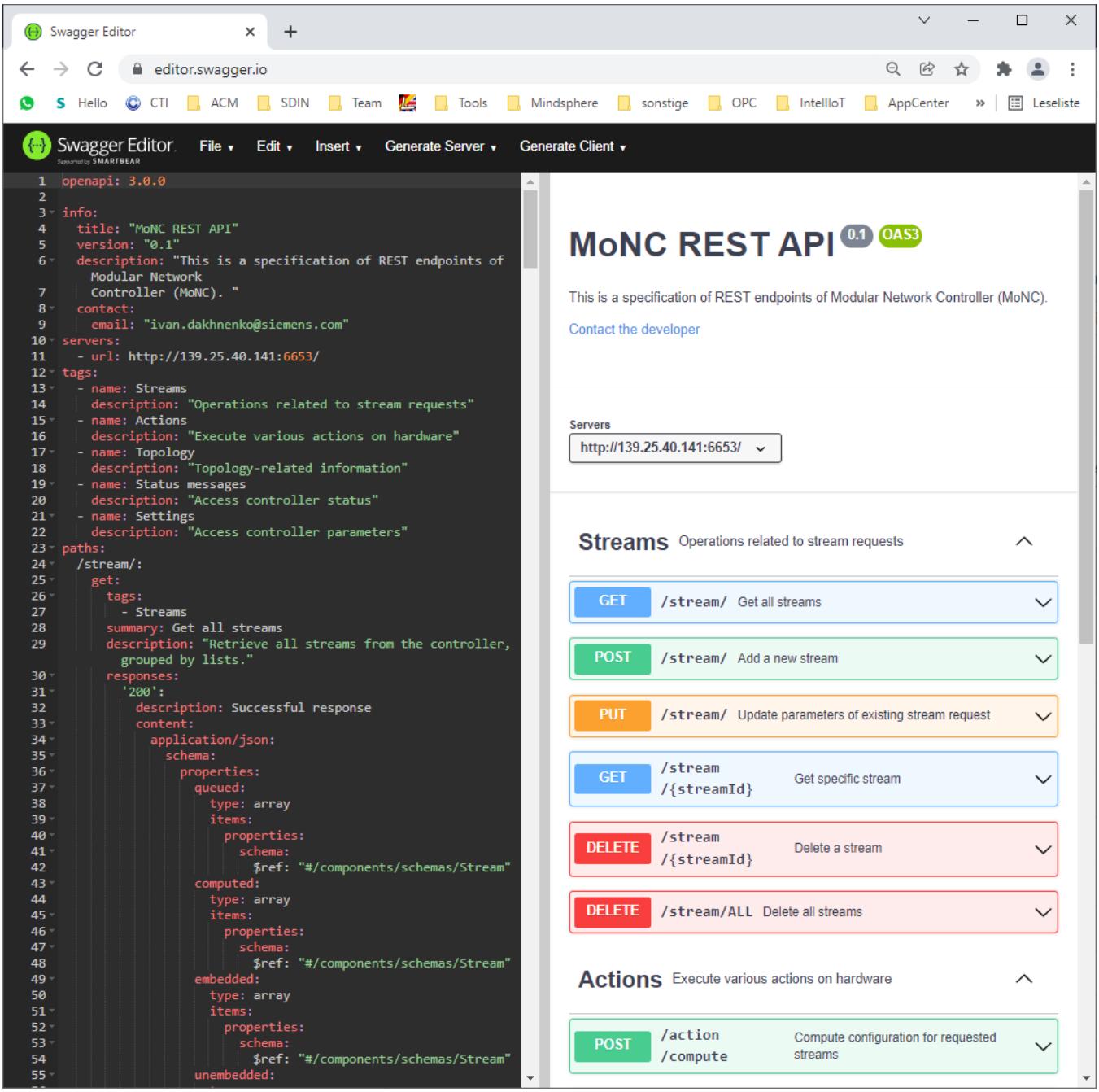


Figure 39: TSN controller interface description in Swagger.

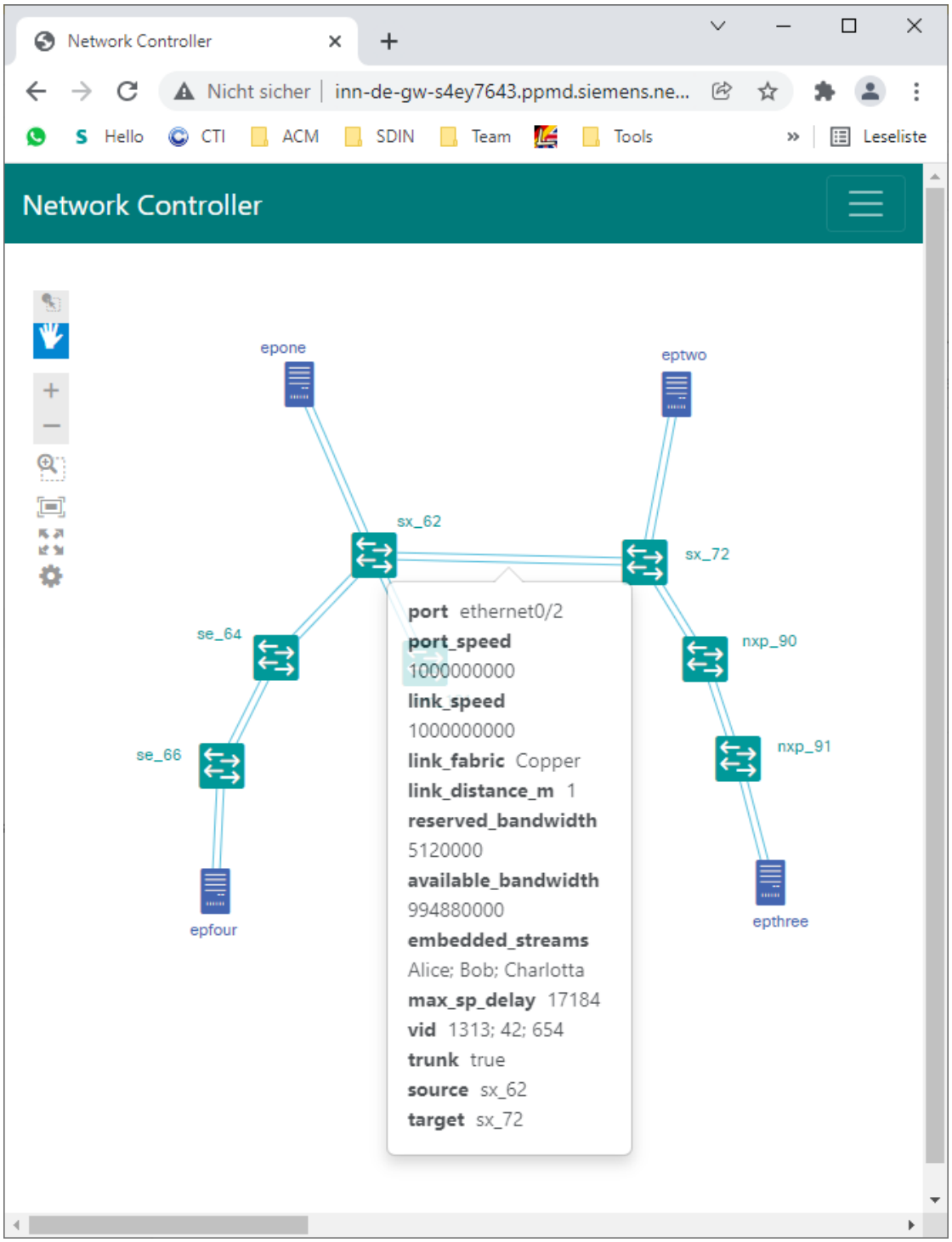


Figure 40: Topology view with stream information in Web UI.

For a standalone test, we send communication service requests, i.e., requests to embed streams, to the TSN controller via its HTTP API and Web UI, see Figure 41. We then request the TSN controller to compute the schedule and embed it to the devices. We will send feasible and infeasible requests, like e.g., the request for a stream called "Dana" in Figure 41, where we requested an infeasible end-to-end delay of only 20 nanoseconds.

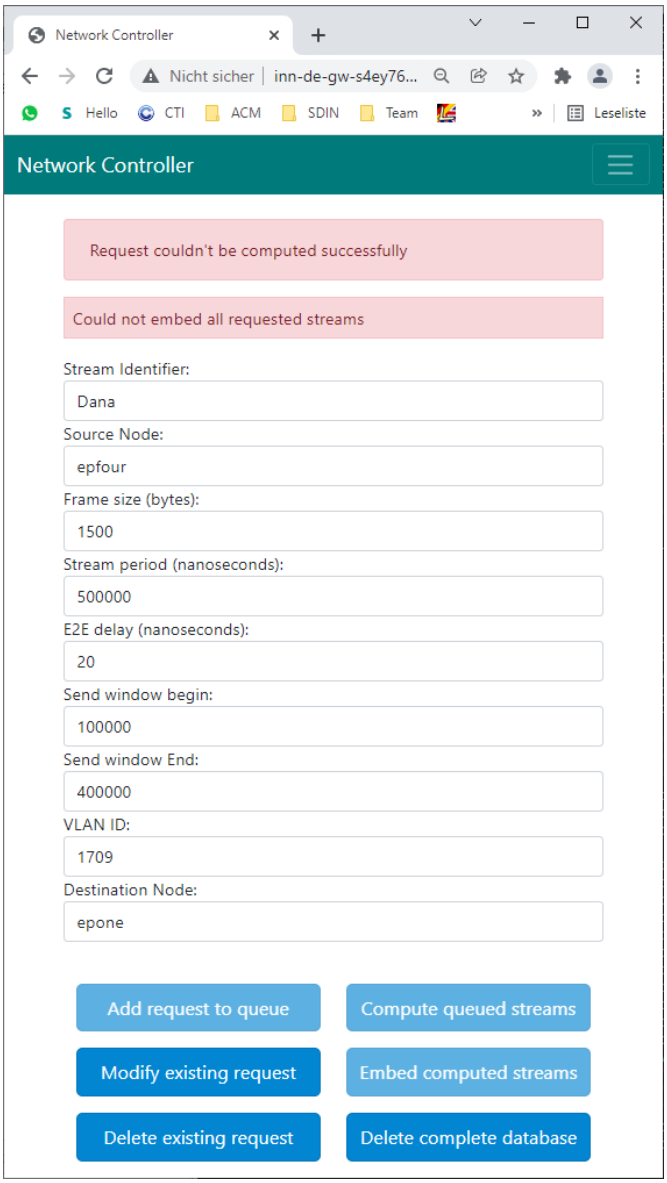
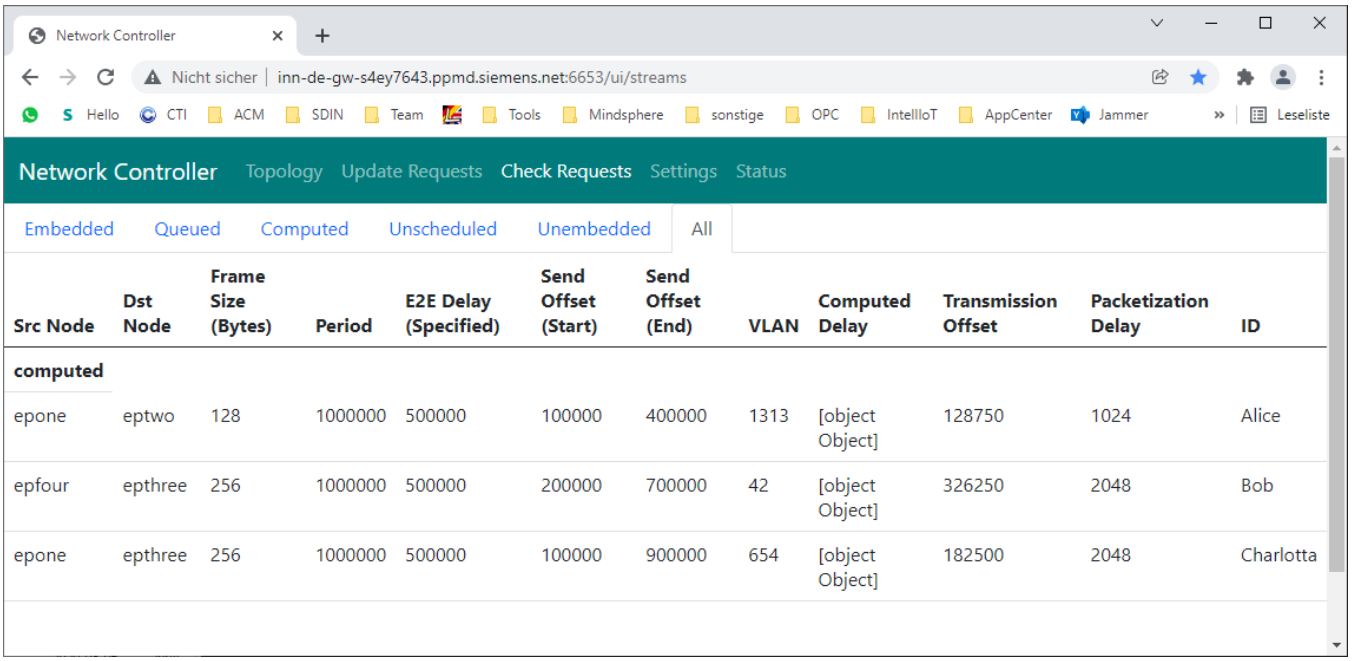


Figure 41: Communication service request via the Web UI.

For feasible requests, we first check if the computed QoS parameters are within the requested ranges, e.g., using the Web UI as shown in Figure 42. This screenshot shows a (meanwhile solved) bug, as the computed end-to-end delay is not displayed properly.



Src Node	Dst Node	Frame Size (Bytes)	Period	E2E Delay (Specified)	Send Offset (Start)	Send Offset (End)	VLAN	Computed Delay	Transmission Offset	Packetization Delay	ID
computed											
epone	eptwo	128	1000000	500000	100000	400000	1313	[object Object]	128750	1024	Alice
epfour	epthree	256	1000000	500000	200000	700000	42	[object Object]	326250	2048	Bob
epone	epthree	256	1000000	500000	100000	900000	654	[object Object]	182500	2048	Charlotta

Figure 42: Communication service details in Web UI.

To check whether the computed schedules were configured in the devices properly, we use the Siemens-proprietary "TSN test app" to measure the real end-to-end delay for computed and embedded streams. For this manual test, we start the TSN test app with the parameters given in the stream request, e.g. cycle time 1ms and frame size 128 Bytes for the stream called "Alice" in Figure 42, and the additional parameters computed by the TSN controller, e.g. the transmission offset of 128750 ns on the end station sending the stream, see Figure 43

```
→  microbox git:(master) sudo ./tsn_test_app -t -s ETF -L 128 -c 1000000 -o 128750
Send mode set to ETF offloading
Packet length (brutto) set to 110
Cycle time set to 1000000
transmission offset set to 128750
using CLOCK_TAI
sending packets ...
packet          0 send time: 15:32:58,    1.128.750
packet       1000 send time: 15:32:59,    1.128.750
packet       2000 send time: 15:33:00,    1.128.750
```

Figure 43: TSN test app, sender side.

The TSN test app on the receiver side Figure 43 provides statistics about delay, jitter and packet loss of the test packets sent according to the configured stream parameters. With this we can easily check if the actual QoS parameters of the requested streams match the requests. Please note that the displayed frame length seems incorrect at the receiver, because the Linux network stack has removed the VLAN tag before the test app receives the packet and computes the packet length.

```
→  microbox git:(master) sudo ./tsn_test_app -l -r HW -F
Receive mode set to HW-based
file output enabled
opening logfile logs/logfile_2021_12_21__15:32:08.csv
receiving from interface with MAC addr: 00:1B:21:DD:F2:74
Listener started, waiting for packets since 15:32:08 ...

Received reset packet. Resetting all counters, DA set to: 00:1B:21:DD:F8:2B
Packet          0 received in cycle          0; ToF forward :       8.064 nsec  (min       8.064, max       8.064)
Packet      1.000 received in cycle      1.000; ToF forward :       7.935 nsec  (min       7.926, max       8.081)
Packet      2.000 received in cycle      2.000; ToF forward :       7.791 nsec  (min       7.784, max       8.081)
```

...

```
Packet     97.000 received in cycle     97.000; ToF forward :       8.021 nsec  (min       7.757, max       8.239)
Packet     98.000 received in cycle     98.000; ToF forward :       8.046 nsec  (min       7.757, max       8.239)
Packet     99.000 received in cycle     99.000; ToF forward :       8.058 nsec  (min       7.757, max       8.239)


received 100000 of 100000 packets, cycle time 1000000 ns, payload size 106 Bytes (124 on wire)
ToF forward (min/avg/max):       7.757 /     8.018,1 /       8.239 nsec, jitter         482 nsec

receive mode: HW-based. test ended at 15:34:39,     375.273 on 2021/12/21

resetting statistic counters
test results have been written to logfile logs/logfile_2021_12_21__15:32:08.csv

→  microbox git:(master)
```

Figure 44: TSN test app, receiver side.

An additional feature to exclude malicious nodes from any communication is currently under development and will be available for cycle 2.

2.5.2.2 COMMUNICATION RESOURCE MANAGER

The 5G Communication Resource Manager is an IntellIoT component aiming at controlling the communication resources allocated by the 5G network. Following the architecture description of the Mosaic5G FlexRIC and FlexCN, the 5G Communication Resource Manager is an xApp and will control the FlexRAN/FlexRIC and LL-MEC/FlexCN components to enable dynamic 5G slice creation, resource optimization as well as dynamic adaptation. It will be deployed by the Edge Orchestrator component. Specifically, it will be responsible of opening, managing and closing 5G RAN. The FlexRAN/FlexRIC/LL-MEC/FlexCN as well as xAPPs are software codes operating on any off-the-shelf laptop or PC supporting containerization (docker, kubernetes) available from https://gitlab.eurecom.fr/mosaic5g.

The Communication Resource Manager communicates with the LL-MEC/FlexCN over an HTTP API as illustrated on Figure 45 (for LL-MEC). Similarly, the Communication Resource Manager communicates with the FlexRAN/FlexRIC over an HTTP API as illustrated on Figure 46 (for FlexRAN).

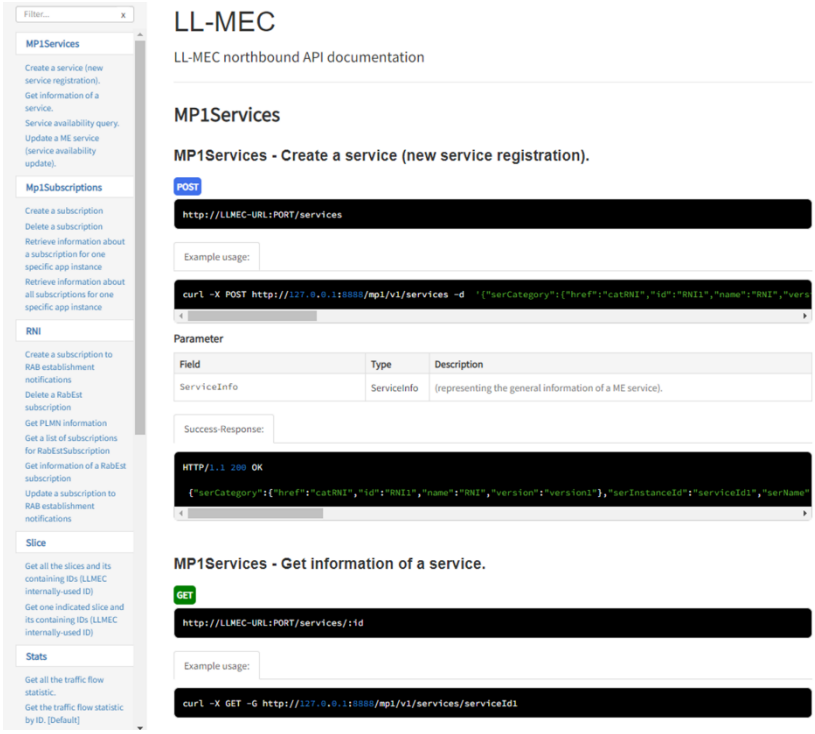


Figure 45. LL-MEC HTTP API available to the communication resource manager.

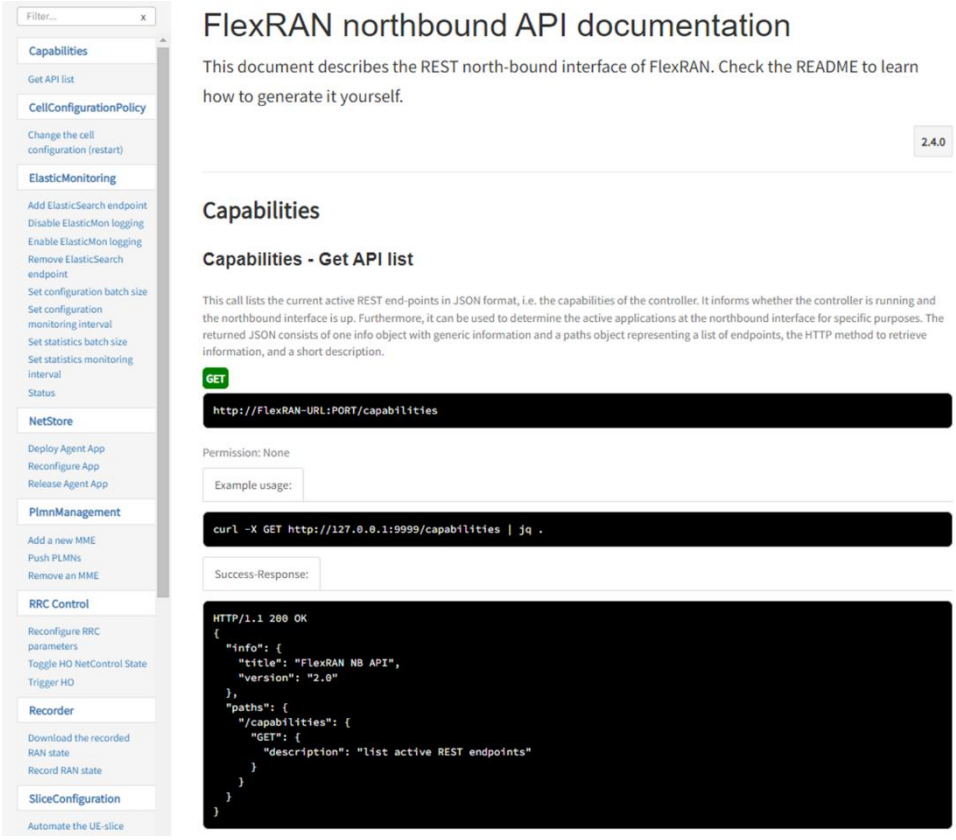


Figure 46. FlexRAN HTTP API available to the communication resource manager.

As a preparation for the integration with the other IntellIoT components, a local setup with a simulated radio-level environment has been used for testing the initial implementation. Specifically, the slicing management for the three UCs is supported currently with the FlexRAN API, and this will be next extended to the FlexRIC API.

There is a default slice that any new UE is subscribed to, and all UEs share these resources equally, in a best-effort manner. Beyond the default slice there will also exist other slices meant to be tailored towards the needs of the UEs. These slices are meant to ensure that UEs with specific needs can have a guaranteed set of resources like bandwidth allocated to them, independent of the number of UEs connected to the network. These slices are referred to as custom slices.

The following functionality has been locally tested:

1. To **define a new slice given the number of RBs and the UE**. This involves choosing a UE, either through its Internal Mobile Subscriber Identity (IMSI) (i.e., a unique identifier associated with the SIM card of a phone) or through Radio Network Temporary Identifier (RNTI) (i.e., a pseudo-random ID that is generated and granted to a UE). After the choice of a UE the number of RBs that are needed will have to be defined, this involves for both UL and DL how much bandwidth is to be allocated for this slice. Upon using this function, a new slice is defined with the requested UL/DL RBs, and then the UE is subscribed to this new slice.
2. To **remove a slice**, it works as a prototype for the future, as currently the input of this function is the specific slice ID. Removing the slice requires the slice ID, and this is not exposed to outside actors currently, which means deleting a specific slice is not possible without guessing the ID.
3. To **associate new UEs to an existing slice** by using the slice ID. The obstacle currently in place is that the slice ID is not exposed to others except for the resource manager. Adding a UE to another slice to e.g., a custom slice will also mean that the resources allocated for a slice are to be shared, and guarantees might not be upheld (at least not without additional allocated RBs). In the final implementation, the aim is to have the Communications Resource Manager identify that the required requirements (UE id, service requirements) are supported by an existing slice, and allocate the new request to an existing slice.

2.5.2.3 5G RAN CONTROLLER

The 5G RAN controller is implemented in OAI as FlexRAN for 4G+ or FlexRIC for 5G RAN architectures. FlexRAN connects to OAI eNB to retrieve RAN-related data. The FlexRAN agent has a northbound and a southbound API.

The Northbound API is implemented as an HTTP API and provides the following HTTP endpoints. The SouthAPIs is implemented as :

- GetConfig – retrieves 4G/5G RAN configuration, such as UL/DL RAN, DRB (data radio bearers) or measurement configurations.
- GetStat – retrieves 4G/5G RAN statistics, such as Channel Quality Indicator (CQI), SINR/RSSI measurements / Localization or UL/DL performance at specific layer
- Commands – the most important API, which actuates the 4G/5G RAN according to specific instructions. It is particularly used for slice creation and per-slice internal configuration.
- EventTrigger – the 4G/5G RAN provides to the FlexRAN controller specific information, such as UE attachment, Scheduling Request.
- Control Delegation – a particular function, which delegate control to the 4G/5G RAN to reduce the latency. Typical delegations are new 4G/5G or handover scheduler, or updated parameters of the existing schedulers or handover algorithm.

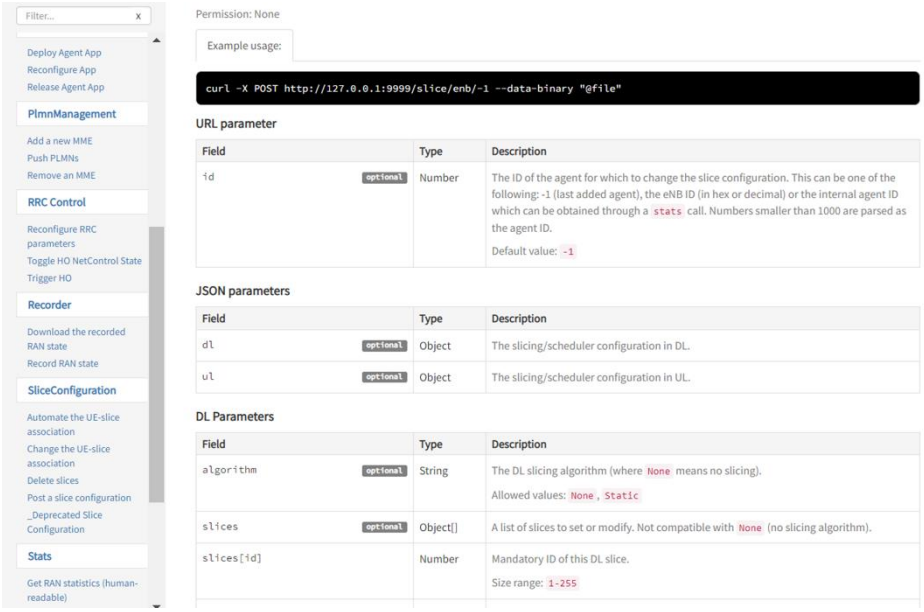


Figure 47: Mosaic5G FlexRAN HTTP API.

Figure 47 depicts the REST API of the FlexRAN controller for a Control API for Slice creation. A full description of the FlexRAN API is available https://mosaic5g.io/apidocs/flexran/

The southbound interface of the FlexRAN controller is implemented according to a Google ProtoBuffer. It however is an internal API which is not exposed outside the FlexRAN component. The FlexRAN southbound API is available https://mosaic5g.io/apidocs/flexran/flexran_spec_v2.2.3.html.

The OAI 5G RAN evolved to support O-RAN E2 agent and the respective APIs. Accordingly, Mosaic5G replaced the FlexRAN agent by a FlexRIC agent, which provides mostly the same HTTP APIs on the northbound interface but replacing the FlexRAN by the E2 API on the southbound interface.

2.5.2.4 5G LOW LATENCY MEC

The main MEC service provided by the 5G low latency MEC is the Radio Network Information Service (RNIS). It is a multi-microservice docker service consisting of three functions:

- RAN data collector – it retrieves data from the 5G RAN over the FlexRAN API. As function of what is requested, it can also actively request specific RAN data.
- Database – the database stores RAN data for faster retrieval.
- Broker – A HTTP server which exposes HTTP APIs to POST or GET specific RNIS data.

Figure 48 illustrates the RNIS service between the 5G RAN and a consuming application, which can be either the Communication Manager or the Edge Controller. FlexRAN is specific to OAI 4G+ architecture and evolved toward a O-RAN compatible E2 agent. However, the RNIS API will not change and will hide the underlying RAN (5G or 4G+).

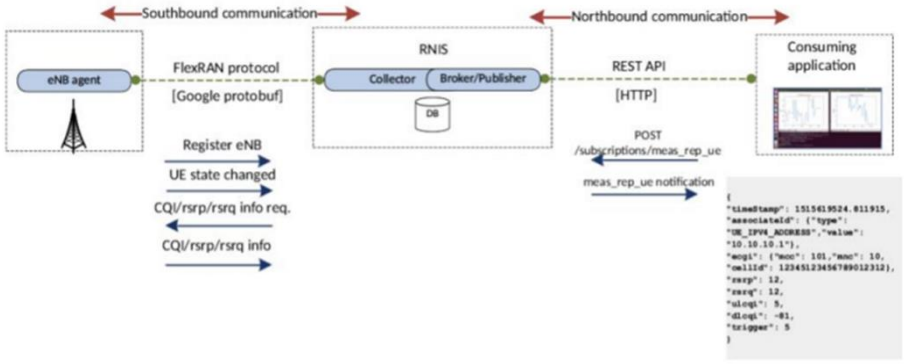


Figure 48: Mosaic5G FlexRAN-based RNIS service.

2.5.2.5 5G NETWORKING MONITORING

The 5G network monitoring can be done either by the RNIS service if monitoring is required as actuation. However, if monitoring is required also for statistics and visualization, elastic search technologies is provided by this component. It corresponds to an Elastic Monitoring component, which exposes Elastic Search APIs to enable or disable an Elastic Search endpoint and connects to Elastic technologies webservices, such as Kibana or Grafana.

The ElasticMonitoring application uses FlexRAN or Flex RIC E2 APIs on a southbound interface to obtain RAN statistics and provide an HTTP API on the northbound API to configure the required statistics and the endpoint to a running ElasticSearch instance.

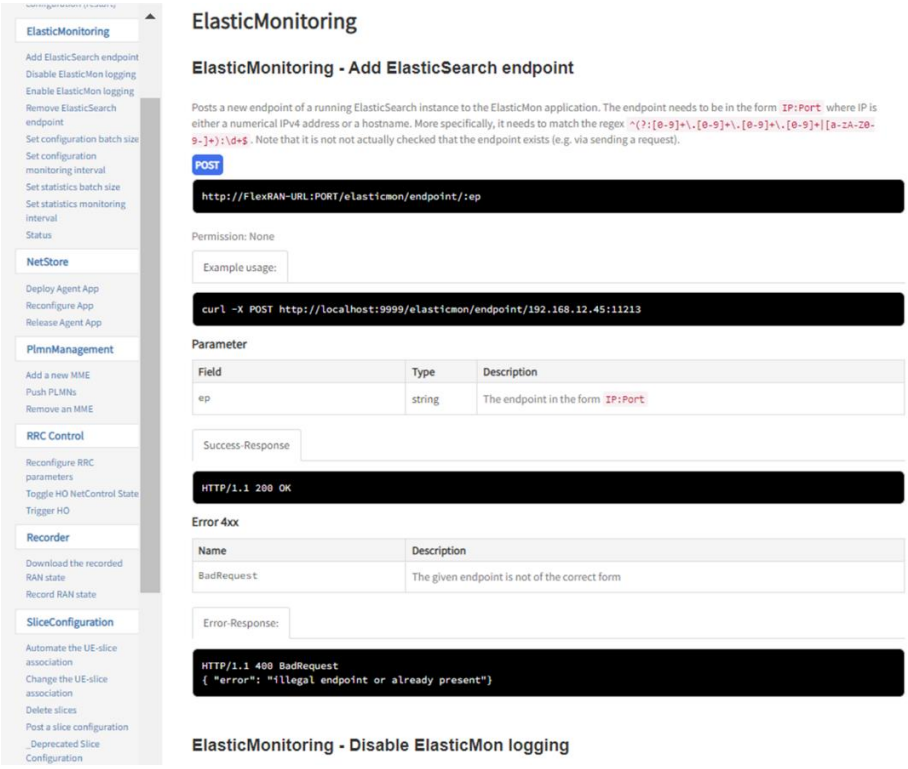


Figure 49: Mosaic5G FlexRAN-based ElasticMonitoring HTTP API.

The HTTP API corresponding to enabling ElasticSearch endpoints as shown in Figure 49 is available at https://mosaic5g.io/apidocs/flexran/#api-ElasticMonitoring. A Mosaic5G FlexRIC-based monitoring component, supporting 5G RAN and compatible with O-RAN is under implementation. The North API will be extended with new E2 features from O-RAN.

2.5.2.6 PRIVATE 5G CORE

The private 5G core is a set of microservices representing 3GPP 5G Core functions and provided by OpenAirInterface CN (https://openairinterface.org/oai-5g-core-network-project/). The private 5G core supports both 4G and 5G core functions as depicted on Figure 50. It operates as a standalone entity which connects to the 4G or 5G RAN via the SGW-U/UFP and AMF/SMF/MME entities for user and control plane respectively. It is considered 'Private' first as it can operate without commercial cellular operators and second it can connect to a private data network (DN/PDN) on the premise it is installed, keeping any communications between the UEs and the data network private.

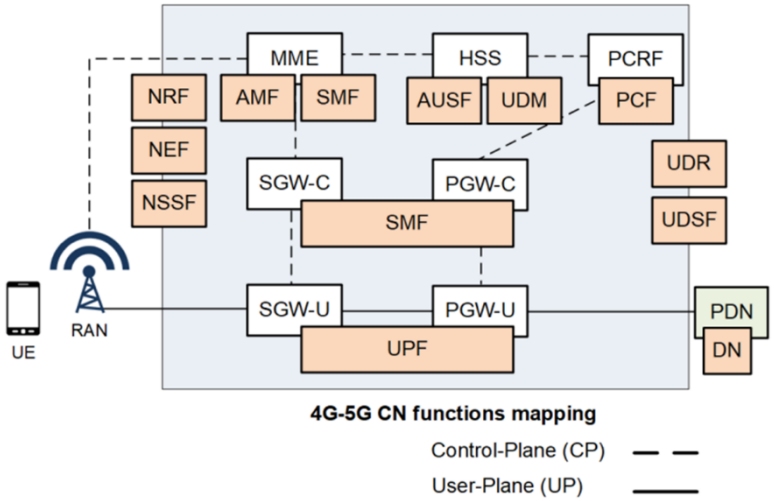


Figure 50: OpenAirInterface 4G/5G Core [source: OAI]

The OAI Private 5G Core can operate on any off-the-shelf PC supporting containerization (docker, kubernetes). The software code can be downloaded from EURECOM OAI GIT repository: https://gitlab.eurecom.fr/oai/cn5g as depicted in Figure 51.

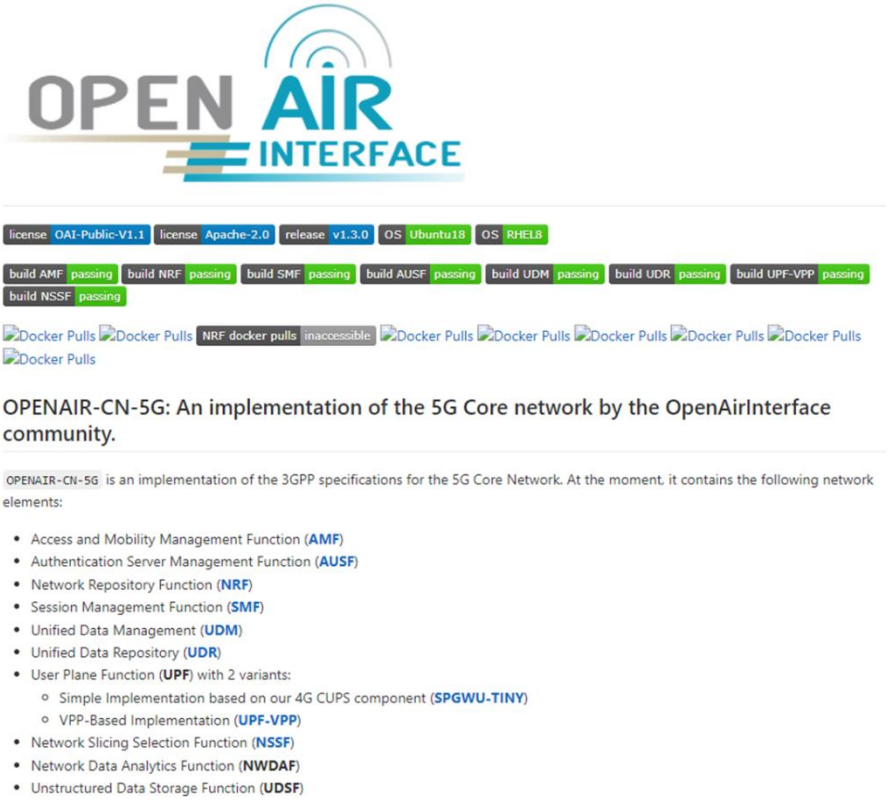


Figure 51: OpenAirInterface 4G/5G Core software repository

The Mosaic5G FlexCN/LL-MEC components interface with the Private 5G Core by reaching the dedicated 5G CN microservices (e.g. UPF, SMF, AMF, RNF) to dynamically adjust 5G CN functions to the requirements of higher services and applications.

2.5.2.7 5G RAN

The 5G RAN represents the wireless domain between a 5G User Equipment (UE) and a 5G base station (eNB/gNB). The 5G RAN is provided by OAI 5G RAN (https://openairinterface.org/oai-5g-ran-project/). The 5G RAN can operate either in non-standalone (NSA) or standalone modes, where the 5G RAN (eNB/gNB) would connect to a 4G Core or a 5G Core (respectively), as illustrated on Figure 52.

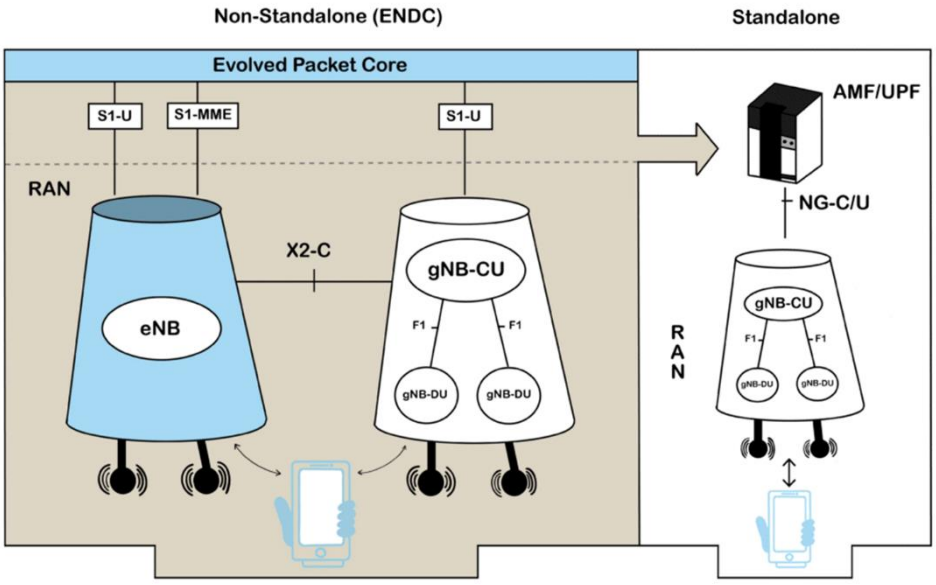


Figure 52: OpenAirInterface 5G RAN NSA and SA architecture (source: OAI).

The 5G RAN is a Software-Defined-Radio (SDR) architecture where RAN functions (RRC, PDCP, RLC, MAC, PHY) are implemented in software and a radio head (RRH) is used to handle the digital/analog and complex signal processing operations, including power amplification. The 5G RAN software contains a UE and a eNb/gNB side, which can be downloaded from EURECOM 5G RAN git repository (https://gitlab.eurecom.fr/oai/openairinterface5g/).

A 5G RAN UE is composed of an off-the-shelf 5G sub-6Ghz radio module connected to an off-the-shelf laptop containing the OAI 5G RAN UE software code, as illustrated on Figure 53.

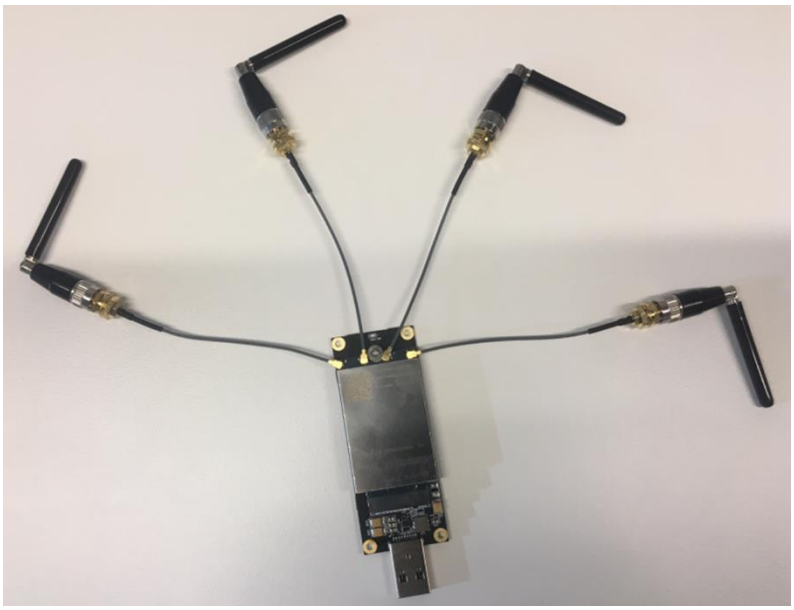


Figure 53:Illustration of the 5G RAN UE radio unit.

The 5G RAN eNb/gNB software code includes in addition to the 3GPP RAN functions (RRC, PDCP, RLC, MAC, PHY) a FlexRAN/E2 agent connected to the FlexRAN/FlexRIC module over a protobuf (FlexRAN) or O-RAN E2 API (FlexRIC). Accordingly, the 5G RAN can be dynamically re-configured or RAN-related information can be reported to the FlexRAN/FlexRIC component for dynamic RAN management.

A 5G RAN gNB is composed of a commercial sub-6GHz radio module (e.g. AW2S RRH - https://www.aw2s.com/RRU.html), connected to a PC containing the OAI 5G RAN gNB software code. 5G omnidirectional antennas are connected to the RRU. Figure 54 illustrates a typical RRU used by OAI 5G RAN.



Figure 54: Illustration of the 5G RAN gNB Remote Radio Unit (RRU) unit.

3 USE CASE SPECIFIC IMPLEMENTATIONS

This chapter corresponds to the 5th core group identified in IntellIoT's architecture (cf. D2.3/*Figure 12*). We describe here the use case specific implementations of components. The specifications of the three use cases are described in D2.1.

3.1 UC1: Agriculture

3.1.1 TRACTOR CONTROLLER

The tractor controller deployed in the agriculture use case is the main control component on the tractor and will host multiple edge apps, introduced in the sections above. The tractor controller is a hardware prototype, that is a ruggedized component explicitly designed for the off-highway domain, so it can stand the different environmental conditions these off-highway vehicles (e.g., tractors, harvesters, but also potentially excavators, etc.) are dealing with. The prototype offers two hosts for development, namely a performance host and a safety host. The performance host is a high-performance computing solution, like e.g., an NVIDIA chip. The safety host will mainly host safety applications ensuring the functional safety of the whole system. An example of a safety host can be an Aurix host. The different hosts are combined using a switch, to enable interaction and communication between the two different hosts. A schematic overview of the solution is provided in Figure 55.

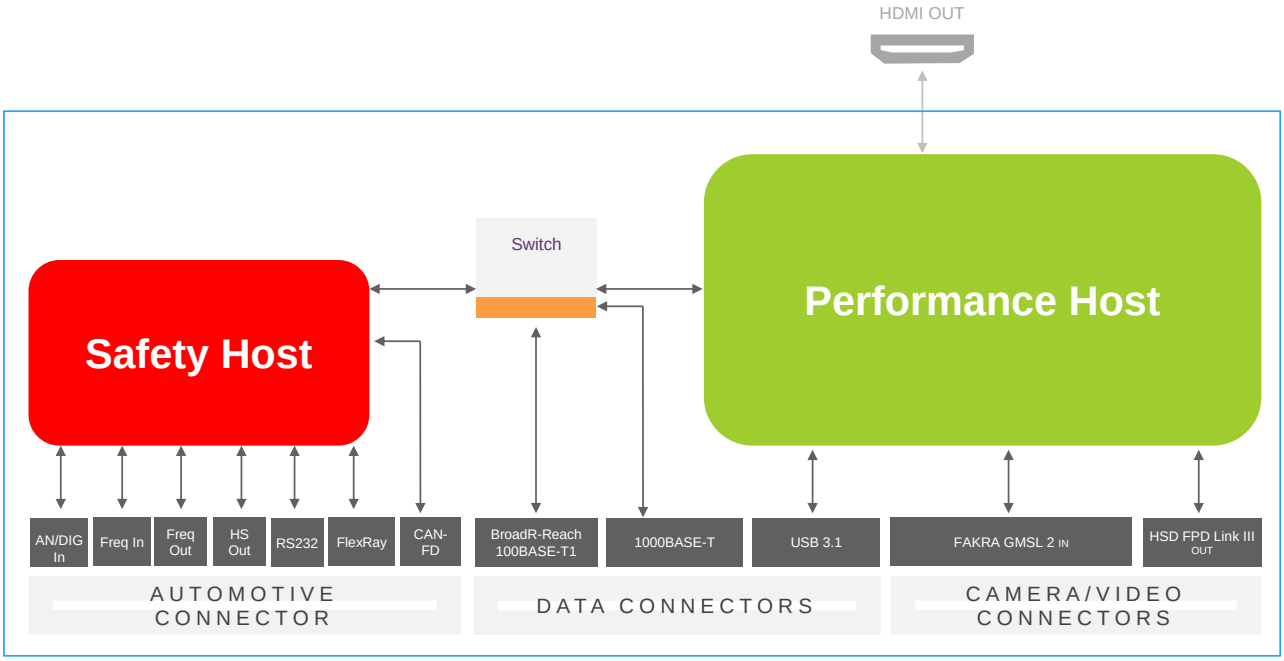


Figure 55: Schematic overview of the Tractor Controller.

The tractor controller prototype supports a Linux Ubuntu platform and will run ROS Melodic for hosting the different apps on the platform. The tractor controller will support a Docker environment, enabling to develop, deploy and run applications with containers. It provides the ability to package and run an application in a loosely isolated environment on a given host, being in this situation the host on the tractor controller. Docker uses a client-server architecture, where the client talks to the Docker daemon, which does the heavy lifting of building, running and distributing the Docker container.

To test the tractor controller as an individual component, a test setup has been created at the TTControl site in Vienna. The platform is connected to a power supply and a debug board (via a vehicle connector cable). The debug board enables a connection to the controller and provides the possibility to access all vehicle connector interfaces (like e.g., Ethernet or CAN interfaces). With a remote connection (SSH), the developer has direct access to the platform via a command window. A monitor connected to the controller provides feedback to the developer and debugging possibilities.



Figure 56: Test setup for tractor controller

The system is first tested if it is up and running.

```
xavier@fast-prototype:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.4 LTS
Release:        18.04
Codename:       bionic
```

Figure 57: tractor controller Ubuntu bionic console.

After this, the platform has been connected to the internet, for installing additional packages. The Host controller runs the following commands to enable IP forward (*sysctl net.ipv4.ip_forward=1*), to set up the necessary rules in the iptables.

```
fiona@ubuntu:~$ sysctl net.ipv4.ip_forward
net.ipv4.ip_forward = 0
fiona@ubuntu:~$ sudo sysctl net.ipv4.ip_forward=1
net.ipv4.ip_forward = 1
fiona@ubuntu:~$ sudo iptables -A FORWARD -i enx00249b2045a7 -o ens33 -j ACCEPT
fiona@ubuntu:~$ sudo iptables -A FORWARD -i ens33 -o enx00249b2045a7 -m state --state RELATED,ESTABLISHED -j ACCEPT
fiona@ubuntu:~$ sudo iptables -t nat -A POSTROUTING -o ens33 -j MASQUERADE
```

Figure 58: Tractor Controller setup IP tables.

On the platform, it needs to be made sure that the default gateway is set to the eth interface of the connected PC. To test and validate the connection, the system will try to ping the internet (e.g., *ping 8.8.8.8*).

```
xavier@fast-prototype:~$ sudo route add default gw 192.168.55.100 eth0:0
```

The Docker functionality has been tested by checking if it has been correctly installed and a test hello-world application has been deployed and executed to make sure that everything runs correctly and smoothly.

```
xavier@fast-prototype:~$ docker --version
Docker version 19.03.6, build 369ce74a3c
xavier@fast-prototype:~$ sudo docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (arm64v8)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/
```

Figure 59: Functionality testing of Docker with hello-world app.

The tests that will be used in the following phase of the use case are integration and functionality tests of the apps that will be deployed and hosted on the controller. Additionally, tests will be performed in the upcoming periods for integrating the controller on the physical tractor and get the required data from the tractor to the apps hosted on the controller. Different interfaces will be required to support all the functionalities of the different apps that will be hosted. These will vary from WebRTC (for streaming video data to the human operator), MQTT for exchanging data between the client apps on the controller and the server apps on the MEC) and potentially other interfaces that are still under definition.

As an example of testing apps on the controller, the DLT app from partner AAU is considered. For running and testing the DLT application on the controller, *docker-compose*, *NodeJS* and *Python3* libraries are required. The missing packages are installed with:

`sudo apt install docker-compose` (respectively *nodejs* and *python3*)

and/or check installed version with *docker-compose -version*

The DLT test application can be found on the IntellIoT Gitlab under: https://gitlab.eurecom.fr/intelliot-project/security/distributed-ledger-technology. To run it on the controller, one has to clone the *sim-dlt-net* directory on the controller and run the DLT manager for testing.

```
xavier@fast-prototype:~/DLT$ cd sim-dlt-net/
xavier@fast-prototype:~/DLT/sim-dlt-net$ sudo docker-compose up --build
[sudo] password for xavier:
Building ganache-cli
Step 1/5 : FROM trufflesuite/ganache-cli:latest
 ---> 9924d0bf3f13
```

Figure 60: tractor controller runDLT manager.

The other applications that will run on the controller, are being validated at the moment and individual tests for the applications are being performed. Later, in the project, these applications will also be integrated on the controller and a complete test of the overall integrated system will be performed.

3.1.2 INTEGRATION WITH THE AGRICULTURE AI MODELS

By relying on the components for the global and local AI, as described in Section 2.2.4, an AI model is developed and provisioned for each use case. The objective of the Agriculture AI model is to provide maneuverer control decisions to the tractor controller upon facing obstacles while the tractor navigates through the waypoints.

INPUT: The input of the Agriculture AI model is a sequence of visual data from the camera mounted on the tractor. This data consists of 3D point cloud as well as a 2D projection corresponds to RGB video frames and information of the obstacle and tractor positions over time instances. The input is pre-processed to filter out the most informative data to be fed into the AI model.

OUTPUT: The inferred output of the AI model corresponds to the tractor control decisions at each time instance that are consist of linear and angular velocity components normalized by predefined maximum velocities. In addition, a confidence measure of the inferred control decisions is calculated to identify the need of retraining and/or escalating to a human operator.

DATA REPOSITORY: The Agriculture AI model uses supervised training, in which, inputs and outputs are accounted as a labelled dataset. The training dataset corresponds to the traces of observed visual inputs and control decision outputs of each maneuverer sequences is stored on the local storage that can be accessed during Local AI training. The traces are named by a unique identifier corresponding to the date and time when the maneuverer sequence is started.

TEST ENVIRONMENT: The training and testing are carried out in a simulated environment prior to integration. The simulation environment is a Python development environment running on a Windows workstation. Here, a testing dataset (a fraction of data is partitioned from the original training data to generate training and testing datasets) is stored in a local storage and then randomly selected traces are exposed to the Agriculture AI model. Therein, the inferred outputs of the AI model are compared with the labelled outputs of the selected trace to evaluate the test accuracy of the AI model inference.

3.1.3 REMOTE OPERATOR DEVICE

In UC 1 the HIL Application will connect to the Client Application installed on the Oculus Quest 2. For this use case, a VR environment is more beneficial, as it increases focus and displays a clearer video stream. Furthermore, the Operator does not have to move around to observe. Instead, the experience is closer to a driving simulator, where the simulated vehicle moves in the real world too, just far away from the driver.

The Operator directly interfaces with the HIL Application's UI visible through the Oculus Quest 2. The device's controllers connect via Bluetooth to the device.

As mentioned earlier in this deliverable, ISAR supports a custom send function. Hence, the controller input can be fed back to the HIL Application, which will send the controls to the machine control via the established WebRTC connection to the tractor.

Custom Send Function

On the Server-Side, CustomSend objects in the scene in Unity are enabled. *ServerApi*, *ApiConfig*, *ConnectionCallbacks* and *ConnectionHandle* are initialized on the Start, not recommended to modify anything unless event handler for receiving incoming messages. Currently, default message (ping) from Unity is running based on timer *OnTimerElapsed* function in CustomSendExample.cs and can be modified to the use case. The message to be sent can be assigned under HIrCustomMessage defined under ConnectionApi.cs. And through the *ConnectionApi*, *PushCustomMessage* the message can be sent to the client. An example to send "OpenKeyboard" message on a button press event to Client:

```
public void OpenKeyBoard()
{
        if (IsConnected)
        {
        byte[] msg = Encoding.ASCII.GetBytes("ShowKeyboard");
        IntPtr unmanaged = Marshal.AllocHGlobal(msg.Length);
        Marshal.Copy(msg, 0, unmanaged, msg.Length);
        HlrCustomMessage message = new HlrCustomMessage();
        message.Length = (uint) msg.Length;
        message.Data = unmanaged;
        _serverApi.ConnectionApi.PushCustomMessage(_handle, message);
        }
}
```

Figure 61: Code Sample - Custom Send Function - Server Side.

Incoming messages from the client are received under *OnCustomMessageReceived* event handler and can be verified and updated further depending on the use case.

```
OnCustomMessageReceived (in HlrCustomMessage message)
{
        int length = (int)message.Length;
        byte[] managedData = new byte[length];
        Marshal.Copy(message.Data, managedData, 0, length);
        string msg = Encoding.ASCII.GetString(managedData);
        Debug.Log($"Received custom message: {msg}");
}
```

Figure 62: Code Sample - Custom Received Function

On the Client-Side, the message is received (for reference) under: Remote Rendering in ImmersiveAppView.cpp under Init function in the callback *m_customMessageCallback*.

Furthermore, message can be verified or used to trigger an event by registering in the *register_custom_message_handler* of the *ConnectionApi*.

An Example to open System Keyboard on the HoloLens based on message from Server-Side:

```
m_customMessageCallback = [ ](HlrCustomMessage* message, void* user_data) {
        auto* immersive = static_cast<ImmersiveAppView*>(user_data);
        uint32_t length = message->length;

        std::string msg {
                message->data, message->data + length};
        };

        if (msg == "ShowKeyboard") {
        immersive->RequestUserInput ();
        }
};

m_isarApi.connection.register_custom_message_handler (
        m_connectionHandle,
        m_customMessageCallback, this
);
```

Figure 63: Code Sample - Custom Send Function - Client Side.

This triggers function RequestUserInput() in the client to open System keyboard on the incoming message.

Currently, default message from client (pong) to the server is assigned in the *onConnectionStateChanged* in the ImmersiveAppView.cpp running on timer periodically and can be changed according to the usage.

To send a message back is similar to the Server-Side code and can be achieved by assigning to *HlrCustomMessage* and using ConnectionApi *push_custom_message*.

3.2 UC2: Healthcare

3.2.1 SMART DEVICES

The details of the collection of the data from the smart devices has been described in Deliverable D3.3. Also, the clinical requirements and technical requirements as well as the relevance of the data collected has been described in Deliverable D1.4.

The smart devices (sensors) will connect to the Local AI through Bluetooth Low Energy (BLE) at certain periods as defined by clinical protocol. For example, the patients might be required by physicians to measure their body temperature two times per day and so a connection between the thermometer and the Local AI will occur two times per day. This will occur similarly for the other sensors.

The frequency of the connection between the smartwatch and the Local AI especially will be described in future deliverables. The smartwatch collects what are clinically the most important measurements (heart rate and electrocardiogram) yet limitations of the hardware components such as battery duration as well as the possible size of the inputs to the Healthcare AI Models might ultimately determine the frequency.

3.2.2 INTEGRATION WITH THE HEALTHCARE AI MODELS

By relying on the components for the global and local AI, as described in Section 2.2.4, an AI model is developed and provisioned for each use case. For the Healthcare use case, an AI Models has been pretrained by researchers in a development environment of their choice, they are serialized to a file and packaged with the Local AI for deployment on the Edge device. Also, as mentioned in Deliverables D2.3 and D3.3, the Local AI collects data from the UC2 sensors, namely the smartwatch, thermometer, blood pressure meter, weight scale, and pulse oximeter. The known AI models described in Deliverable D3.2 and the extensions that will be described in Deliverable D3.6 can then use the collected data as inputs.

In addition, the Local AI takes care to schedule the local training of the AI models as part of the federated learning process. The schedule will most likely need to be constrained to occur during periods in which the Edge device is idle and charging to not render the device as unusable for the user. These restrictions on the timing of the training process might be extended to include that the Edge device be connected to a 5G network. Alternatively, the training will be performed and only the sending of model updates to the Global AI will be constrained to occur when the Edge device is connected to the 5G network.

3.2.3 INTEGRATION WITH THE PATIENTS' DATA REPOSITORY

The data collected from the UC2 sensors is also needed for analysis by physicians. The necessary subset of collected data will be thus sent to the Patients Data Repository through a protected and secure API and a HTTP over TLS connection.

3.3 UC3: Manufacturing

3.3.1 ROBOT CONTROLLER

The robot controller is an abstraction simplifying the control of the very flexible UR5 robot arm to a limited number of fixed tasks necessary for the manufacturing use case demonstrator.

It provides an HTTP interface that allows agents in the HyperMAS and other apps to make the robot perform fixed tasks, move it to predefined positions and read out its status / actual position. It controls the UR5 arm via (non-real-time) TCP commands / URScript. For real-time control of the robot, which is required for the HIL scenario, HIL application will directly interact with the UR5 robot arm.

The robot's HTTP interface has been defined together with HSG as the main "users" of it in form of an OpenAPI specification, using Swagger. From the Swagger description, a W3C WoT Thing Description has been created to enable agents to be programmed against the robot's affordances.

The latest version of the interface specification can be found in https://docs.google.com/document/d/1qWyrCS9p57yd1aCUA-pawPoA8IAqLt7StHaMDn2nrjw/edit; https://app.swaggerhub.com/apis-docs/danaivach/robot-controller/1.1.0 contains examples of the API usage.

The implementation of the API has been started using a Node-RED based workflow and it is tested in most parts, using the Swagger-internal functionalities. Implementing the robot controller functionality itself is in progress as well.

3.3.2 ENGRAVER APP

This text addresses the integration of the engraver service edge app with the laser cutter machine as well as the milling machine.

The Engraver App is an edge app, which provides a service interface for controlling the engraving machines. In IntellIoT we support two types of engraving: per laser cutter and per milling machine. Both machine types are different in nature, but, for the purpose of the manufacturing use case, we covered difference by a uniform web interface.

By December 2021, the *Mr Beam* lasercutter (https://www.mr-beam.org/) is supported. For the support of the Milling Machine, the same service implementation will be used (although it might be instantiated in an own app identity). It is the goal, that the same web API interface towards the client (HyperMAS) is used for both laser cutter and milling machine. Depending on the parametrization the one or the other machine can be selected.

The process of engraving of text should be similar on both machine types. It consists of three phases:

1. preparation
2. engraving
3. postprocessing

Each phase requires service call on the web interface of the engraver service. The web service requests include parameters, which kind of machine is used, and which concrete machine (id) is used. Although, we considered that each machine type has its own type of engraver service and accordingly machine type might be redundant. For the moment, we keep this for flexibility in the future.

Figure 64 and Figure 65 show the UML sequence diagram of production of a workpiece with the Mr Beam laser cutter and respectively with the milling machine. Note that this diagram merely shows one of several possible executions since the concrete HTTP queries are only configured at system set-up time using the Web-based IDE discussed in Section 4.2.2 and in Deliverable D3.1.

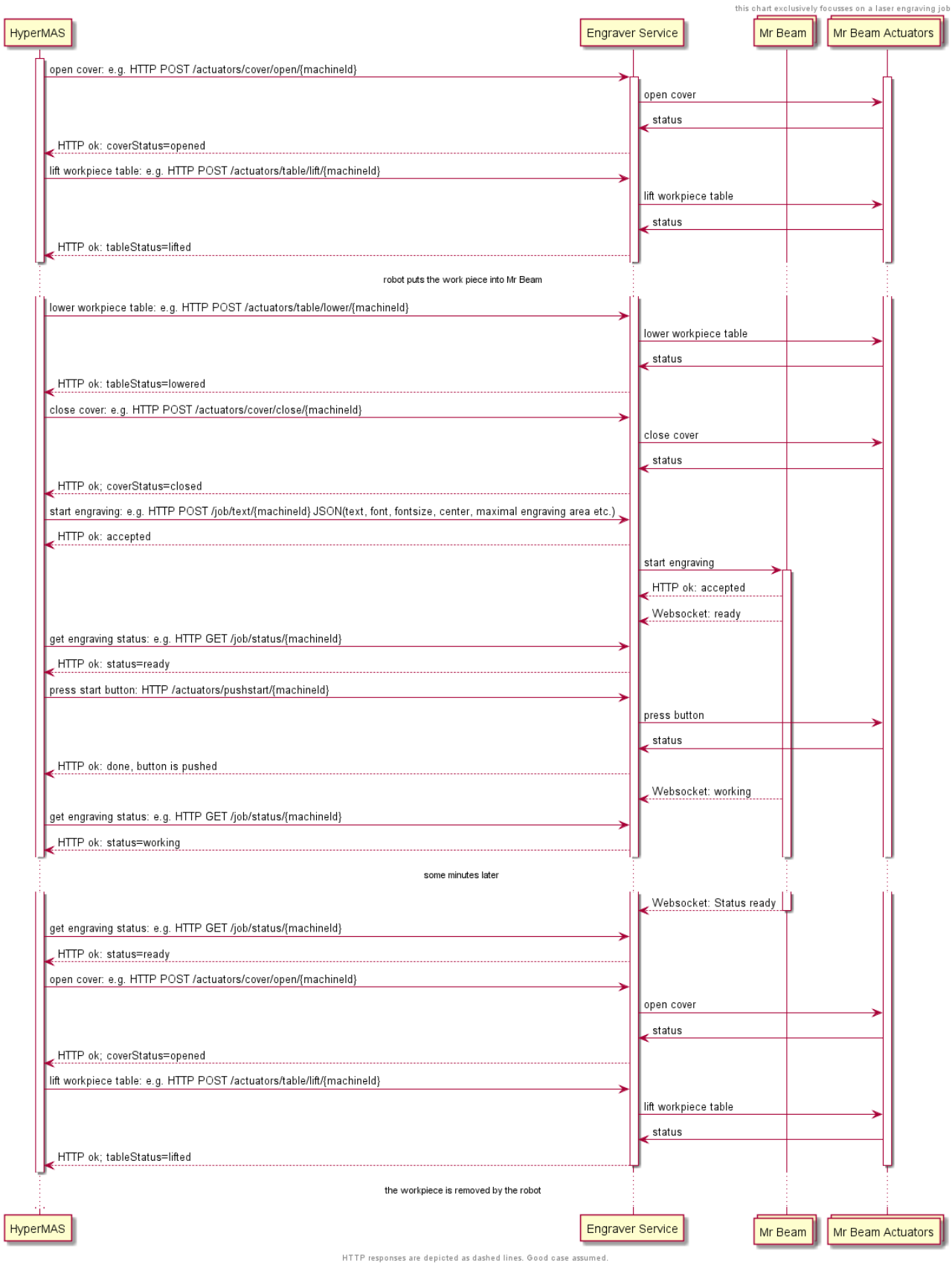


Figure 64: Example workflow of production with laser cutter.

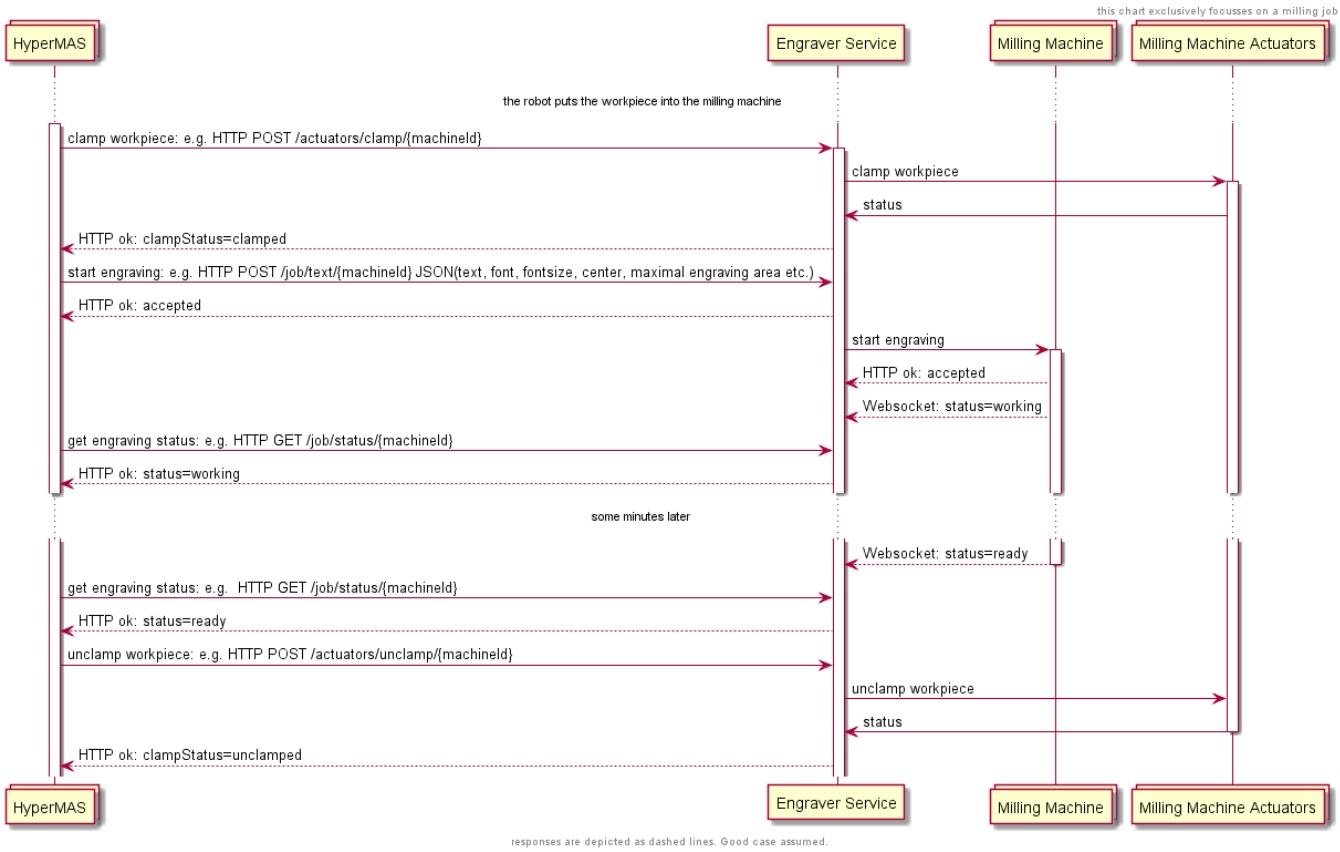


Figure 65: Example workflow of production with milling machine.

3.3.2.1 PHASE 1 AND 3: PREPARATION AND POSTPROCESSING

Phases 1 and 3 include physical actions so that a production process (engraving a text) can be started. This is necessary due to the nature of the manufacturing use case. Here work pieces are put into the machine by a robot arm before the production can start and picked up by the same robot arm, when the production is finished.

The phases are differing whether a laser cutter or a milling machine is used. For the preparation of the production process with the laser cutter, the lid has to be opened, a table has to be lifted and lowered and a button has to be pressed, to start the laser cutter. The post processing includes the same steps in reverse order. In Figure 64, the different action of preparation and postprocessing can be deduced.

For milling machine, the only preparation and postprocessing procedure is to clamp and unclamp the work piece. Figure 65 shows the workflow.

We use actuators on the basis of Tinkerforge elements (https://www.tinkerforge.com). Those are devices, which are able to control the mechanics for the actions. They are connected via TCP/IP over Ethernet to the Engraver Service. The Engraver Service, in turn, offers a REST façade to expose their functionality to HyperMAS.

3.3.2.2 PHASE 2: ENGRAVING

To start the engraving job, the interface of the Engraver app includes a distinct service function. Incoming requests are processed by the engraver service in such a way, the engraving can be started by the native interface of the machine. This requires service translation, as well as the preparation of the input parameter in such a way, the machine can deal with that. An important capability of the Engraver Service is, that a vector graphics in the SVG format is created out of the ascii text input. Here, the text is rendered into path elements, consisting of lines, arc and Bézier curves.

The typical client will be HyperMAS. The web API for managing the engraving hob will be the same for both milling machine and laser cutter. These APIs will be expressed in a machine-readable way as W3C WoT Thing Descriptions to enable agents to be programmed against them.

To start an engraving job, a HTTP POST on `{{base_url}}/job/text` has to be issued.

An example for the message body is as follows:

```
{
  "machineType": "laser",
  "machineId": "laser-1",
  "text": [
    "here comes the text"
  ],
  "font": "ABeeZee",
  "variant": "regular",
  "fontsize": 76,
  "positionReference": "center",
  "maxWidth": 150,
  "maxHeight": 80,
  "x": 120,
  "y": 220
}
```

Figure 66. /job/text example.

With `machineType`, the Engraver Service learns which implementation is used. Machine types could be laser or milling-machine. With `machineId`, the concrete machine is identified. Although we have just one laser and one miller, we should keep it for the future. The value can be arbitrary for the moment. The value of `font` describes the text font, which is used. `variant` is its corresponding variant (e.g., regular, bold, italics). The font has to be available at the engraver. The values for font and variant must correspond to the file name of the font file. In this case, the file name was `ABeeZee-regular-esDR31xSG-6AGleN6tKukbcHCpE.ttf`. The pattern `{{font}}-{{variant}}-{{whatever}}` is used by all available fonts. We use Google Fonts and have 4500 font/variant combinations available (~500 different fonts). For practical reasons, we currently support only ABeeZee (regular, italics), Abel (regular) and Roboto (regular).

- `fontSize` is the size of the font.
- `x`, `y` is the coordinates of the origin of the engraved text in the coordinate system of the machine.
- `positionReference` is the origin of the engraving. Possible values are currently `northwest`, `center`.

The value of `text` is the content to be engraved. It is encoded in an array. Each array element is a line. Currently only single line (one array element) is supported. If multi line text will be supported in the future is up to be decided. Also, how they are oriented (left, center). The concrete machine has to be configured with respect to its network addresses, its working area and the orientation of the coordinate system.

The interface to the machine is exposed to the engraver service. The engraver service itself is an edge app, which runs on Industrial Edge devices. It is connected via TCP/IP with the engraving machines. The Engraver service will start the engraving via HTTP REST and JSON binding. Accordingly, the milling machine has to provide service functions to start engraving and to provide information about its status. For the start engraving request, the engraver service sends the SVG body plus extra information in a JSON body, which is required for the engraving. The SVG includes the representation of the text encoded in a SVG path (PATH tag). The SVG includes also the

definition of a transformation matrix, so that the text is already translated to its position on the work piece. The concrete interface is internal between Engraver Service and milling machine and not further discussed here.

3.3.2.3 TESTING THE ENGRAVING

For integration tests, a test instance is available. This instance includes a simulator of an engraving machine, which can be used to test protocol interaction. The API is documented as OpenAPI specification, which is tracked in IntellIoT's gitlab environment, and a W3C WoT TD has been created from this. Integration tests with the HyperMAS components – both, the HyperMAS Infrastructure and the Web-based IDE for HyperMAS – have been successful.

3.3.3 INTEGRATION WITH THE MANUFACTURING AI MODELS

By relying on the components for the global and local AI, as described in Section 2.2.4, an AI model is developed and provisioned for each use case. The Manufacturing AI model carries out two main tasks: detection of engraving area and calculating the grab spot of work pieces.

INPUT: Despite of the task (detection of engraving area or computing grabs spot), the input of the Manufacturing AI model is an image from the camera mounted over the workbench or engraver/milling machine. This image consists of the top view of the workpiece and special markers placed on the surface that are used to determine the physical dimensions. The input is pre-processed to filter out the most informative data to be fed into the AI model.

OUTPUT: The inferred output of the AI model corresponds to one of the following depending on the task: (I) the engraving area corresponds to a circular region (the largest in-circle) with the coordinates of the centre and length of the radius or (II) the grabbing location in terms of a tuple corresponds to x and y coordinates and angle referring to the grabbing arm orientation. In addition, a confidence measure of the inferred output is calculated to identify the need of retraining and/or escalating to a human operator.

DATA REPOSITORY: The Manufacturing AI model uses supervised training, in which, inputs and outputs are accounted as a labelled dataset. A training datapoint for engraving task consists of the camera image of the workpiece and the coordinates of the centre and the radius of largest in-circle labelled by a human operator. For the grabbing task, a training datapoint corresponds to the camera image of the workpiece and the x-y coordinates along the angle of which the robot arm gripper has to be oriented. Each of the datapoint is named by a unique identifier corresponding to the captured date and time followed by an identifier for the task ("_D" for area detection and "_G" for grabbing).

TEST ENVIRONMENT: The training and testing are carried out in a simulated environment prior to integration. The simulation environment is a Python development environment running on a Windows workstation. Here, a testing dataset (a fraction of data is partitioned from the original training data to generate training and testing datasets) is stored in a local storage and then randomly selected images are exposed to the Manufacturing AI model. Therein, for each selected test image, the inferred outputs of the AI model (both the engraving area and the grab spot separately) are compared with the labelled outputs to evaluate the test accuracy of the AI model inference.

3.3.4 REMOTE OPERATOR DEVICE

Use Case 3 is very similar to UC 1 regarding used method. The main difference here is the hardware. Whereas in UC 1 a VR solution was selected as best fitting platform, in UC 3 an AR solution better suits the use case's needs. Analyzing a digital twin and providing new waypoints for the robot arm requires physical movement around a virtual element.

To improve the accuracy of the waypoint definition, this UC uses the Holo-Stylus (in short Stylus) as input device, rather than the HoloLens 2 hand gesture input.

The Holo-Stylus is a two-component solution consisting of the Stylus-Pen and the HMU (Head Mounted Unit). The HMU is clipped onto the HoloLens 2 and amongst other things, uses 2 built-in cameras to track the Stylus' movements. This information is sent to the Client Application via a Bluetooth connection.

The Client Application sends the Stylus input via the above-mentioned custom send function to the HIL Application. From there, the HIL application transmit the control input via the WebRTC connection (setup up by Siemens) to the Robot controller.

3.3.4.1 TESTING THE REMOTE OPERATION

The test environment will need a powerful internet connection and protection from or not available signal blocking technology which could interfere with the Bluetooth signal. For UC3 the room's setup should support AR applications, by means of lighting. It should not be too bright, or too dark, as this would impact the clarity of the images. The setup should ideally also have fully charged power banks (3.0 amp capable) with a long enough cable available, so that the devices can be plugged in at all times.

4 CONCLUSIONS AND FUTURE WORK

Here in D5.1 the integration of the components from D2.3 is described in a first version without having them deployed and tested in the 'real' demonstrators yet. How to do so is currently (M16) being worked out and shall be described in D5.2 (M17). The insights from Milestone 4 (M19 - *first version of IntellIoT framework integrated, demonstrated and evaluated, OC1 finished*) will be used as a start for the final version of this document that is D2.4.

During Cycle 1, WP3 has been focusing on the definitions, implementations, and preliminary testing of use case specific AI components, HIL services, HyperMAS, and distributed ledger technologies. These preliminary versions are then to be integrated and further developed during Cycle 2 in collaboration with the components developed under WP4. It is to be noted that some changes are expected within WP3 components to cope with the integration process, which will be reflected under future deliverables accordingly.

The communication and computation infrastructure components and interfaces give support to the three UCs, with the variations explained in this deliverable and in D4.1-D4.4. Cycle 1 has focused on the definition, implementation and testing of the three main WP4 parts separately: the 5G communication network, the computation infrastructure, and the trust infrastructure. In cycle 2, the integration within them and with WP3 components will be addressed. The activities in WP4 will be completed during cycle 2, and this can lead to additional components or variations of the current plan if new research findings are deemed important for IntellIoT goals. These potential changes, if any, will be reflected in the next version of D5.1 and the rest of affected deliverables in WP2 -WP5.

REFERENCES

[1]	Ciortea, A., Boissier, O., Ricci, A. (2019) Engineering World-Wide Multi-Agent Systems with Hypermedia. In: Weyns D., Mascardi V., Ricci A. (eds) Engineering Multi-Agent Systems. EMAS 2018. Lecture Notes in Computer Science, vol. 11375. Springer. https://doi.org/10.1007/978-3-030-25693-7_15	
[2]	Ciortea, A., Mayer, S., Bienz, S., Gandon, F., Corby, O. : Autonomous Search in a Social and Ubiquitous Web. Personal and Ubiquitous Computing, 2020, https://doi.org/10.1007/s00779-020-01415-1	
[3]	Castro, Inma T., Luis Landesa, and Alberto Serna. "Modeling the energy harvested by an RF energy harvesting system using gamma processes." *Mathematical Problems in Engineering* 2019 (2019).	
[4]	T. S. Rappaport, Wireless communications: principles and practice, vol. 2. 1996.	