



H2020 5G Dive Project
Grant No. 859881

D2.3 Final Specification of 5G-DIVE Innovations

Abstract

This deliverable D2.3 provides the final specification of the 5G-DIVE solution innovation. An overview of the framework governing the solution specification is presented first. This is then followed with detailed final specification for each of the targeted vertical pilots, namely Industry 4.0 and Autonomous Drone Scout. This deliverable is complemented with the final implementation reported in deliverable D2.4.

Document properties

Document number	D2.3
Document title	Final specification of 5G-DIVE innovations
Document editors	Timothy William (NCTU)
Document contributors	ADLINK: Ivan Paez, Luca Cominardi ASKEY: June Liu, KJ Liu IDCC: Filipe Conceição, Ibrahim Hemadeh, Alain Mourad AAU: Hergys Rexha, Sebastien Lafond III: Tzu-Ya Wang ITRI: Andee Lin, Samer Talat NCTU: Muhammad Febrian Ardiansyah, Timothy William UC3M: Milan Groshev, Carlos Guimarães, Laura Caruso TELCA: Aitor Zabala, Javier Sacido, Matteo Pergolesi RISE: Luca Mottola, Saptarshi Hazra ULUND: Chao Zhang, Per Ödling EAB: Chenguang Lu, Gyanesh Patra
Document reviewers	UC3M: Milan Groshev, Carlos Guimarães, Laura Caruso, Antonio De La Oliva ITRI: Samer Talat RISE: Bengt Ahlgren NCTU: Timothy William IDCC: Filipe Conceição
Target dissemination level	Public
Status of the document	Final
Version	1.0
Publication Date	June 30 2021

Disclaimer

This document has been produced in the context of the 5G Dive Project. The research leading to these results has received funding from the European Community's H2020 Programme under grant agreement N° H2020-859881.

All information in this document is provided “as is” and no guarantee or warranty is given that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

For the avoidance of all doubts, the European Commission has no liability in respect of this document, which is merely representing the author’s view.

Contents

List of Figures.....	5
List of Acronyms	7
Executive Summary	9
1. Introduction.....	10
2. 5G-DIVE Solution Design.....	11
2.1. 5G Connectivity	11
2.1.1. 5G NSA	12
2.1.2. 5G SA.....	14
2.2. DEEP Platform	16
2.2.1. Data Analytics Support Stratum – DASS.....	16
2.2.2. Business Automation Support Stratum – BASS.....	22
2.2.3. Intelligent Engine Support Stratum – IESS.....	38
3. 5G-DIVE Solution for I4.0 Use Cases.....	44
3.1. I4.0 Use Case 1: Digital Twin	44
3.1.1. Key Module Design.....	44
3.1.2. Mapping to the DEEP Platform.....	51
3.2. I4.0 Use Case 2: Zero Defect Manufacturing	57
3.2.1. Key Module Design.....	57
3.2.2. Mapping to the DEEP Platform.....	58
3.3. I4.0 Use Case 3: Massive MTC	61
3.3.1. Key Module Design.....	61
3.3.2. Mapping to the DEEP Platform.....	68
4. 5G-DIVE Solution for Disaster Relief Using Autonomous Drone	71
4.1. ADS Use Case 1: Drone Fleet Navigation.....	71
4.1.1. Key Module Design.....	71
4.2. ADS Use Case 2: Intelligent Image Processing	74
4.2.1. Key Module Design.....	75
4.3. ADS Mapping to the DEEP Platform	79
5. Conclusion.....	82
6. References	83

7. Appendix 87

7.1. DLT-Based Federation Support..... 87

7.2. BASS and OCS Integration Workflow 93

7.3. Data driven RAN Intelligence 94

7.3.1. O-RAN architecture 95

7.3.2. 3GPP Edge fabric standardization efforts..... 96

7.3.3. ATSSS xApp 97

List of Figures

FIGURE 2-1 ILLUSTRATION OF 3GPP CONNECTIVITY OPTION 1, 2 AND 3	12
FIGURE 2-2 5G NSA AND EDGE DATA CENTER SOLUTION FOR ADS TRIAL TRIAL	13
FIGURE 2-3 5G SA SOLUTION FOR I4.0 TRIAL	15
FIGURE 2-4: DASS ARCHITECTURE.....	17
FIGURE 2-5: DASS SESSION ESTABLISHMENT USER-PASSWORD AUTHENTICATION	18
FIGURE 2-6: DASS ZERO-COPY COMMUNICATION.....	21
FIGURE 2-7: ZENOH AND ZENOH.NET PROTOCOL LAYERS.....	22
FIGURE 2-8 BASS UPDATED ARCHITECTURE.....	24
FIGURE 2-9 CREATION OF A NEW VERTICAL SERVICE FROM A BLUEPRINT	27
FIGURE 2-10 VISUALISATION OF THE LIST OF SERVICES.....	28
FIGURE 2-11 VISUALISATION OF A RUNNING SERVICE WITH DETAILS ON ITS COMPONENTS	28
FIGURE 2-12 VISUALISATION OF VERTICAL REGIONS EXAMPLE VIEW	29
FIGURE 2-13 VSD EXAMPLE	31
FIGURE 2-14 BASS AND OCS INTEGRATION	33
FIGURE 2-15. ACTIVE MONITORING SIMPLIFIED WORKFLOW	36
FIGURE 2-16 IESS UPDATED ARCHITECTURE	39
FIGURE 3-1: SYSTEM BLOCK DIAGRAM FOR DIGITAL TWIN	45
FIGURE 3-2: BASE DIGITAL TWIN SYSTEM MODULE INTERACTIONS.....	46
FIGURE 3-3 REPLAY FEATURE MODULE DESIGN	47
FIGURE 3-4 SLA ENFORCER MODULE DESIGN	49
FIGURE 3-5: MOVEMENT PREDICTION MODULE INTEGRATION OPTIONS.....	50
FIGURE 3-6: DIGITAL TWIN END-TO-END DEPLOYMENT WITH BASS.....	52
FIGURE 3-7: DIGITAL TWIN DASS-ENABLED REPLAY FEATURE.....	53
FIGURE 3-8 DIGITAL TWIN IESS OBSTACLE AVOIDANCE.....	54
FIGURE 3-9 SLA ENFORCER E2E MECHANISM	55
FIGURE 3-10: DIGITAL TWIN IESS MOVEMENT PREDICTION	56
FIGURE 3-11 CUBES AS A PRODUCT OF THE FACTORY	57
FIGURE 3-12 ZDM SETUP WITH AWS WAVELENGTH.....	58

FIGURE 3-13 BASS SERVICE INSTANTIATION	59
FIGURE 3-14 DASS-ENABLED TELEMETRY DATA COLLECTION	60
FIGURE 3-15 MMTC DEPLOYMENT DIAGRAM.....	63
FIGURE 3-16 ZMQ PUB/SUB PATTERNS FOR DOWNLINK AND UPLINK SIGNALS	63
FIGURE 3-17 AN EXAMPLE OF EXPOSING LORA APPLICATION AS A KUBERNETES SERVICE	64
FIGURE 3-18 AN EXAMPLE OF KUBERNETES DEPLOYMENT YAML FILE FOR LORA	65
FIGURE 3-19: COMPARING PACKETS FROM THE SAME DEVICE USING SIAMESE NETWORK	66
FIGURE 3-20: COMPARING PACKETS FROM THE DIFFERENT DEVICES USING SIAMESE NETWORK	66
FIGURE 3-21: TRAINING PROCESS	67
FIGURE 3-22: AUTHENTICATION AND PERIODIC UPDATE FRAMEWORK.....	68
FIGURE 3-23 MMTC BASS DEPLOYMENT	69
FIGURE 3-24: RF FINGERPRINTING IESSS MAPPING	70
FIGURE 4-1 5G NSA AND EDGE SYSTEM BLOCK DIAGRAM FOR ADS.....	71
FIGURE 4-2 IDROS ARCHITECTURE.....	72
FIGURE 4-3 IDROS ORCHESTRATOR ARCHITECTURE.....	74
FIGURE 4-4 ADS USE CASE 2 SYSTEM OVERVIEW	75
FIGURE 4-5 EAGLESTITCH SYSTEM STITCHER MODULE PIPELINE	77
FIGURE 4-6 DRONE DATA PROCESSOR SYSTEM	77
FIGURE 4-7 ADS USE CASE MAPPING TO THE DEEP PLATFORM	79
FIGURE 7-1 SEQUENCE MESSAGE DIAGRAM FOR BASS FEDERATION SMART-CONTRACT AND ADMINISTRATIVE DOMAINS DURING FEDERATION.....	89
FIGURE 7-2: FEDERATION USING POA CONSENSUS: (TOP) SUMMARIZED PHASE; (MIDDLE) CONSUMER AD; (BOTTOM) PROVIDED AD; [46]	91
FIGURE 7-3: FEDERATION USING POW CONSENSUS: SUMMARIZED TIMES [46].....	91
FIGURE 7-4 BASS AND OCS INTEGRATION WORKFLOW	94

List of Acronyms

AD	Administrative Domain
ADS	Autonomous Drone Scout
AI	Artificial Intelligence
API	Application Programming Interface
ATSSS	Access Traffic Steering, Splitting and Switching
BASS	Business Automation Support Stratum
BSSID	Basic Service Set Identifiers
CPE	Customer Provided Equipment
CPU	Central Processing Unit
DASS	Data Analytics Support Stratum
DCAS	Drone Collision Avoidance System
DEEP	5G-DIVE Elastic Edge Platform
DLT	Distributed Ledger Technologies
EagleEYE	Aerial Edge-enabled Disaster Relief Response System; an end-to-end PiH detection and localization system
EagleStitch	An end-to-end 2D stitching system
EFS	Edge and Fog Computing System
EPC	Evolved Packet Core
FDU	Fog05 Deployment Unit
GRU	Gated Recurrent Units
GUI	Graphical User Interface
HTTP	Hyper Text Transfer Protocol
I4.0	Industry 4.0
IESS	Ingelligence Engine Support Stratum
IoT	Internet of Things
KPI	Key Performance Index
LSTM	Long Short Term Memory
LXD	Next generation system container manager
MANO	Management and Orchestration
ML	Machine Learning
MNO	Mobile Network Operator
NBI	Northbound Interface
NDN	Named Data Networking
NR	New Radio
NSA	Non-standalone
NSD	Network Service Descriptor
OCS	Orchestration and Control System
OPTUNS	Optical tunnel network system
PaaS	Platform-as-as-Service
PHY	Physical
PiH	Person in need of Help
PoA	Proof-of-Authority

PoW	Proof-of-Work
PUB/SUB	Publish/Subscribe
RAM	Random Access Memory
RL	Reinforcement Learning
ROS	Robot Operating System
RPC	Remote Procedure Call
SA	Standalone
SC	Federation Smart-contract
SIM	Simulator
SLA	Service Level Agreements
SLI	Service Level Indicator
SLO	Service Level Objective
TCN	Temporal Convolution Network
TLS	Transport Layer Security
UE	User Equipment
URI	Unified Resource Indicator
UUID	Universally Unique Identifier
VAPs	Virtual Access Points
VAR	Vector Autoregression Models
VIM	Virtual Infrastructure Manager
VSB	Vertical Service Blueprint
VSD	Vertical Service Descriptor
ZDM	Zero Defect Manufacturing

Executive Summary

This deliverable provides the final specification of the 5G-DIVE solution innovation for the targeted use cases in the Industry 4.0 (I4.0) and Autonomous Drones Scout (ADS) vertical pilots. The main achievements of this deliverable include:

- 1) Designed an Service Level Agreement (SLA) enforcement framework for the services deployed using the DEEP platform. This framework provides a mechanism to guarantee the fulfilment of the negotiated SLAs between the use cases and the platform.
- 2) Developed the final design framework governing the 5G-DIVE solution. This final solution was built on top of the initial design reported in D2.1. The final solution describes in this deliverable includes the underlying 5G connectivity, and the DEEP platform with its three supports systems namely BASS (Business Automation Support System), DASS (Data Analytics Support System), IESS (Intelligence Engine Support System).
- 3) Applied the final design framework to the targeted I4.0 use cases, namely i) digital twinning, ii) zero defect manufacturing (ZDM), and iii) massive machine type communications (mMTC). With each use case using its own customized design to fulfil the objective. Each customized design is applied in the context of specific intelligence engines such as movement prediction and replay in digital twinning, object defect detection in ZDM, and RF radio security in mMTC.
- 4) Applied the design framework to the targeted ADS use cases, namely i) drones fleet navigation and ii) intelligent image processing for Drones. With each use case using its own customized design to fulfil the objective. Each customized design is applied in the context of specific intelligence engines such as image analytics, geolocation and object detection in ADS Use Case 2.

The final specification in this deliverable served as a basis for the implementations reported in the software deliverable D2.4. It is noteworthy that not all specifications in this deliverable are or will be implemented. All of the inputs reported in this document will serve as feedback for WP3.

1. Introduction

This deliverable D2.3 is a continuation of the work carried out in D2.1 [1] and targets the final specification of the 5G-DIVE solution innovation for each use cases in the I4.0 and ADS vertical pilot. In this deliverable, final development results of the DEEP platform are reported. This development results include the final specification that serves as a basis for the final implementations which are reported in an accompanying software deliverable D2.4. Details on 5G-DIVE solution design tailored for each use case are also provided, while experimental results for the final implementations will be explored further in WP3. The integration of the 5G-DIVE solution design to each use cases will allow for vertical to gain insight from data through the DASS, vertical services management and automation through the BASS, and provisioning of AI/ML related services for the vertical through the IESS. The organization of this deliverable as well as this deliverable achievement are listed as follows.

In Section 2, the final 5G-DIVE solution design are presented. This section is further divided into two subsections. First, Section 2.1 details the 5G connectivity solution used to support each uses cases. For I4.0 use case, 5G SA solution is used. While for ADS use case, 5G NSA solution is used. Second, Section 2.2 details the updated DEEP platform design framework, which includes the DASS, BASS, and IESS. For the DASS, use cases have already adopted it to perform data preprocessing, storage, as well as data dispatching. As for the BASS, each use cases utilizes it to serve as a common reference framework for lifecycle management. And lastly, the IESS is used by the use cases for facilitating the training and cross-validation of AI/ML models.

Section 3, and Section 4 details the final sytem design for the I4.0, and ADS use case respectively. In these two sections, refinement of the system key modules, as well as addition of new modules are reported. In addition, details on how each use case maps to the DEEP platform as well as how each use cases interacts with the the DEEP component, namely the DASS, IESS, and BASS are also reported in the end of these sections.

Finally, the conclusions for this deliverable is outlined in Section 5.

Appendixes are also provided at the end of this deliverable covering the following topics:

- Section 7.1: DLT-Based Federation Support
- Section 7.2: BASS and OCS Integration Workflow

The above appendixes provide feasibility study of DLT-based federation, as well as workflow details on the integration of BASS and OCS. These information are deemed valuable for the reader to read in conjunction with the innovation specifications in the main body of this deliverable.

2. 5G-DIVE Solution Design

In this section, we describe the final 5G connectivity solution, as well as 5G-DIVE DEEP stratum for supporting the I4.0, and ADS verticals. We will be using the edge computing infrastructure to enable support for an end-to-end Platform-as-a-Service (PaaS) service model. Details on the edge computing infrastructure have been previously reported in D2.1 [1]. However, in an effort to make this document self-contained, some of the relevant terminology will be briefly described:

- **Edge and Fog Computing System (EFS):** A logical system which serves as an environment for hosting virtualized functions, services, and applications.
- **Orchestration and Control System (OCS):** A logical system in charge of composing, controlling, managing, orchestrating, and federating one or more Edge and Fog Computing System.

The rest of this section are organized as follows. In Section 2.1, we will describe the details on the final 5G connectivity solution for both use cases. 5G Non-standalone (NSA) setup will be used for supporting ADS, while 5G standalone (SA) setup will be used for supporting I4.0. In Section 0, we will describe the updates and improvements made to the DEEP strata.

2.1. 5G Connectivity

There are multiple connectivity options defined in the 3GPP architecture for 5G deployment [2]. As shown in as shown in Figure 2-1, 3GPP connectivity option option 3 and option 2 have been adopted in the industry to evolve from the baseline option 1 (LTE only) to support 5G NR. Option 3 is referred to as 5G NSA (non-standalone), where option 2 is referred as 5G SA (standalone). Option 3 enables a rapid introduction of 5G NR to market by software upgrading the existing 4G EPC (Evolved Packet Core) (referred as 5G EPC). 5G SA (option 2) represents the next step of 5G deployment, which requires a newly developed 5GC (5G Core). In 5G NSA, it requires an LTE connectivity anchor. The control plane and mobility management are done with LTE and EPC. NR connectivity is only used for user plane data. Therefore, UEs have dual connectivity connecting both LTE and NR carriers simultaneously. 5G NSA allows new 5G services to be introduced quickly while maximizing the reuse of existing 4G networks. With the 4G anchor which UEs always connect to, 5G can be added with spotty coverage as a capacity boost for traffic hotspots. It doesn't require a national deployment of 5G, which takes time and costs to deploy. In 5G SA, NR is deployed alone with 5GC without the need for the LTE anchor. When a large scale of 5G network is deployed, it is natural to evolve to 5G SA to unlock the full potential of 5G connectivity. 5G SA is also suitable to some industrial use cases, where the 5G connectivity is locally deployed, e.g. in factories. In case of the mobility handling between 4G and 5G, it is done by the 3GPP interworking interface between EPC and 5GC. Other 3GPP connectivity options are lack of industry support since supporting so many options simultaneously increase significantly the network and UE operation complexity. Therefore, in 5G-DIVE project, the main stream options of 5G NSA (option 3) and 5G SA (option 2) will be trialed, where 5G NSA will be used in the ADS trial, while 5G SA will be used for the I4.0 trial.

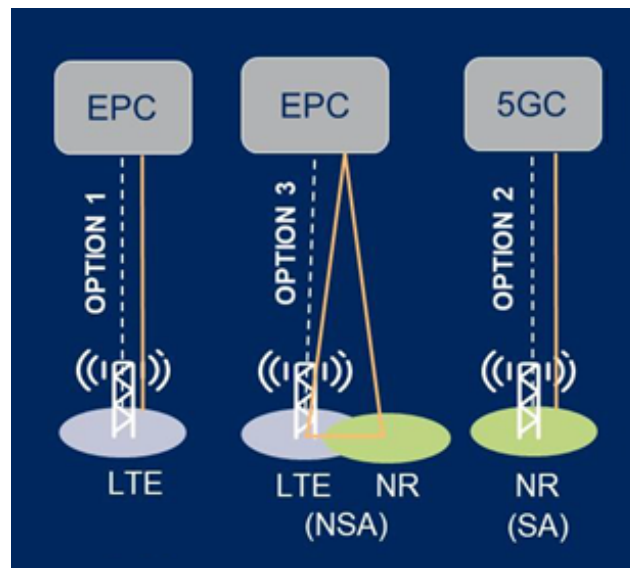


FIGURE 2-1 ILLUSTRATION OF 3GPP CONNECTIVITY OPTION 1, 2 AND 3

2.1.1.1. 5G NSA

For the ADS trial, 5G NSA will use leveraging the developed 5G gNB and 5G EPC supporting NSA. As described before, this allows for 5G User Equipment (UE) to not only transmit data through 4G eNB (both user and control planes) but also additionally through 5G gNB (only user plane). The initialization phase for a UE to connect to 5G NSA is described as follows:

1. When a UE boots up, it will attach log in to the NSA network through the eNB.
2. During the UE's running time, it will continuously measure for 5G NR Synchronization Signal Blocks (SSB) emitted from the gNB, and reports the measurements back to the eNB.
3. Once the signal strength between the UE and a gNB is sufficiently strong, and the UE's throughput requirement demands a 5G connection, the serving eNB starts to signal to target gNB, and tells the EPC to modify the data plane bearer of the UE from the current eNB to target gNB.
4. After the modification is done, gNB takes over eNB to continue serving the UE. Since the UE is mobile, the UE keeps reporting the gNB signal strength via the LTE connection to eNB, just in case that UE moves away from gNB coverage. When it happens, the eNB asks the UE to fall backs to LTE, and thus the data flows of the UE will not be disconnected.

The NSA Option 3 grants gNB to set up a split data bearer in both the gNB and eNB. By doing so, a UE may attach to both base stations at the same time. This enables the UE to leverage higher aggregated bandwidth.

Figure 2-2 below depicts the final solution of 5G NSA and edge data centre for ADS deployed at NCTU. From the left of the figure, there are the integrated 4G LTE and 5G NR base station from ASKEY, 5G EPC from III, iMEC from ITRI, the localized drone application servers developed by NCTU on virtualized computing platform (Kubernetes or Openstack), and lastly, OPTUNS [3] optical tunnel network system to interconnect all of the components together.

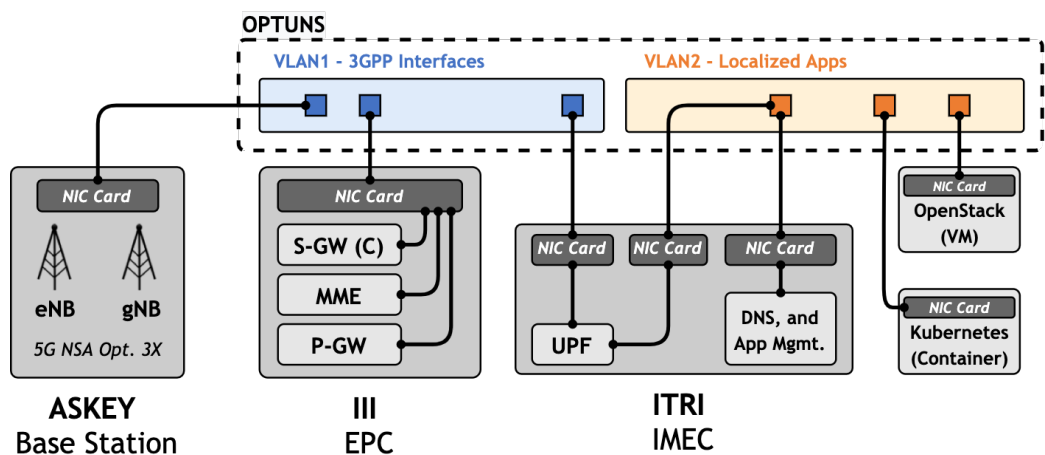


FIGURE 2-2 5G NSA AND EDGE DATA CENTER SOLUTION FOR ADS TRIAL

2.1.2. 5G SA

5G SA is based on 5GC completely redesigned to realize full 5G capabilities. Service-based architecture (SBA) is adopted, which facilitates the cloud-native design and automation, as well as increasing flexibility and scalability. It simplifies network operations, increase service creation agility, supports ultra-low latency features, supports advanced network-slicing functions, and facilitates new vertical use cases.

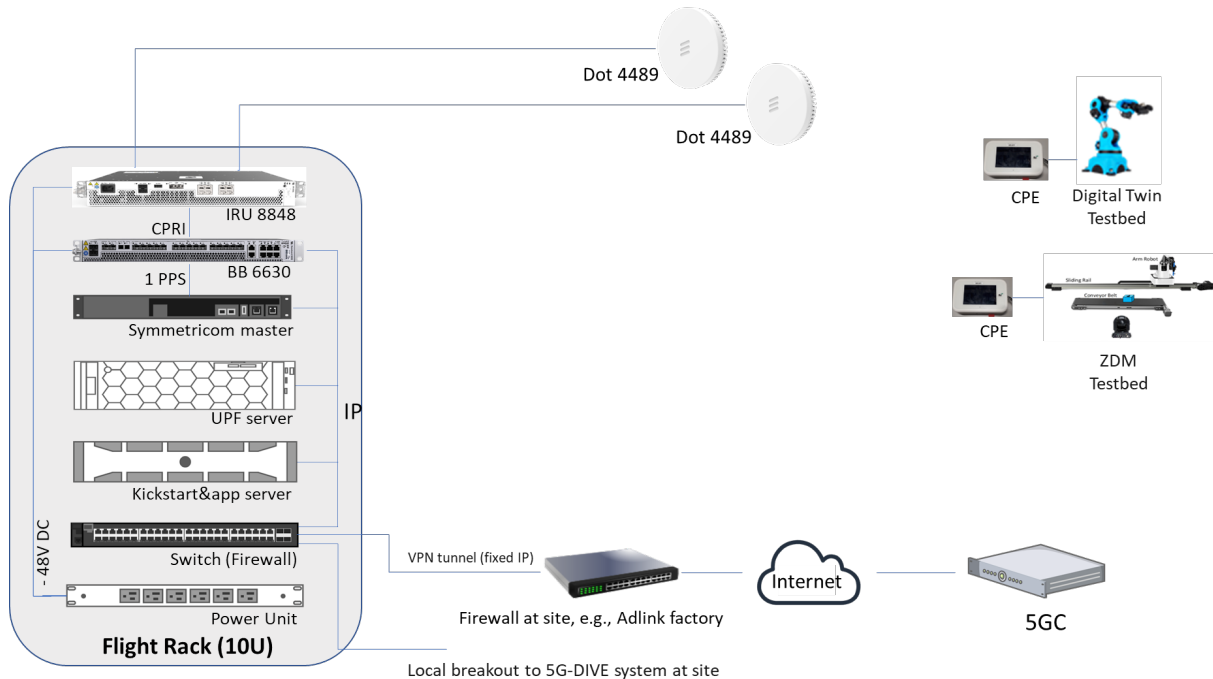


Figure 2-3 shows the 5G SA solution which will be used in the I4.0 trial. The RAN part (i.e. gNB) of the solution is based on Ericsson Radio Dot System, which is composed of Dot 4489, IRU 8848 and BB 6630. Symmetricom Sync Master is used to provide 1 PPS to BB 6630 for synchronization. 5GC is deployed remotely, while an UPF function is deployed locally in a server collocated with BB 6630. The UPF breaks out the user plane traffic locally to the 5G-DIVE Edge system to minimize the latency. A kickstart and application server are also included in the setup. It is used for the UPF installation. After installation, it is kept in the setup for maintenance and troubleshooting. A switch (router) is used to provide IP connectivity in the setup. It also serves as a firewall for network security. A VPN tunnel will be configured between the setup and the 5GC for the core network connectivity. The local equipment is installed in a 10U rack, referred to as Flight Rack, which is made for easy transport as a whole. 5G SA capable CPEs are used to provide the 5G NR connectivity to the I4.0 testbeds of Digital Twin and ZDM.

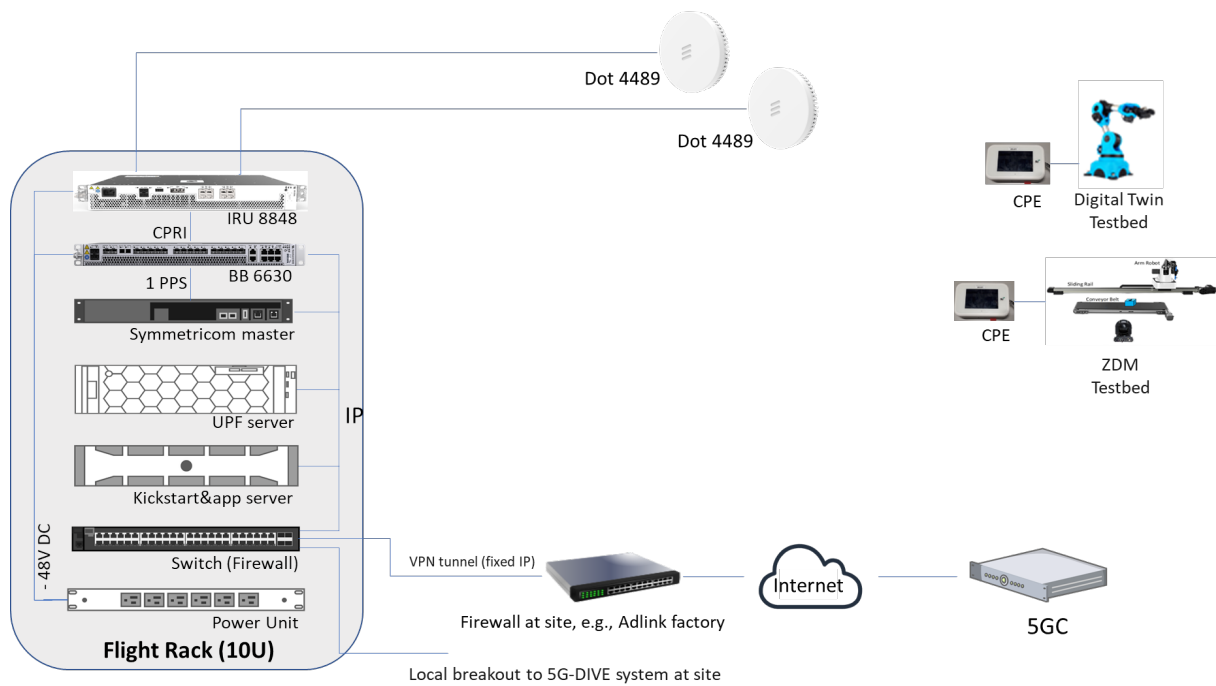


FIGURE 2-3 5G SA SOLUTION FOR 14.0 TRIAL

2.2. DEEP Platform

The DEEP platform, as designed in D1.3 [4], comprises three main supporting strata: DASS, BASS, and IESS. In this section, we present an updated version of the implementation of each supporting stratum.

2.2.1. Data Analytics Support Stratum - DASS

According to D2.1 [1], the DASS was conceived in the 5G-DIVE project as a data analytics platform suitable for distributed and heterogeneous edge and fog environment. This provides to the vertical industries the necessary support for gaining useful insight from the data generated from their business processes which can be potentially enriched with a varied set of context information. Moreover, the geo-transparent locality offered by the edge and fog system can be exploited to access data with a data-centric network based on NDN [5] and at the same time process and analyze sensitive data where they are generated, thus enabling strict privacy and low latency response for mission critical systems. Finally, DASS enhances EFS and OCS operations by providing data analytics tools for the infrastructure management.

2.2.1.1. Architecture

In Figure 2-4, the second release for the DASS architecture implementation can be seen. Components with green background are components already implemented during the first release of the DASS and its implementation details defined in D2.1 [1]. Components with a yellow background are component which are implemented as part of the second release of the DASS. In the following subsections, we will describe in detail each of the newly developed components.

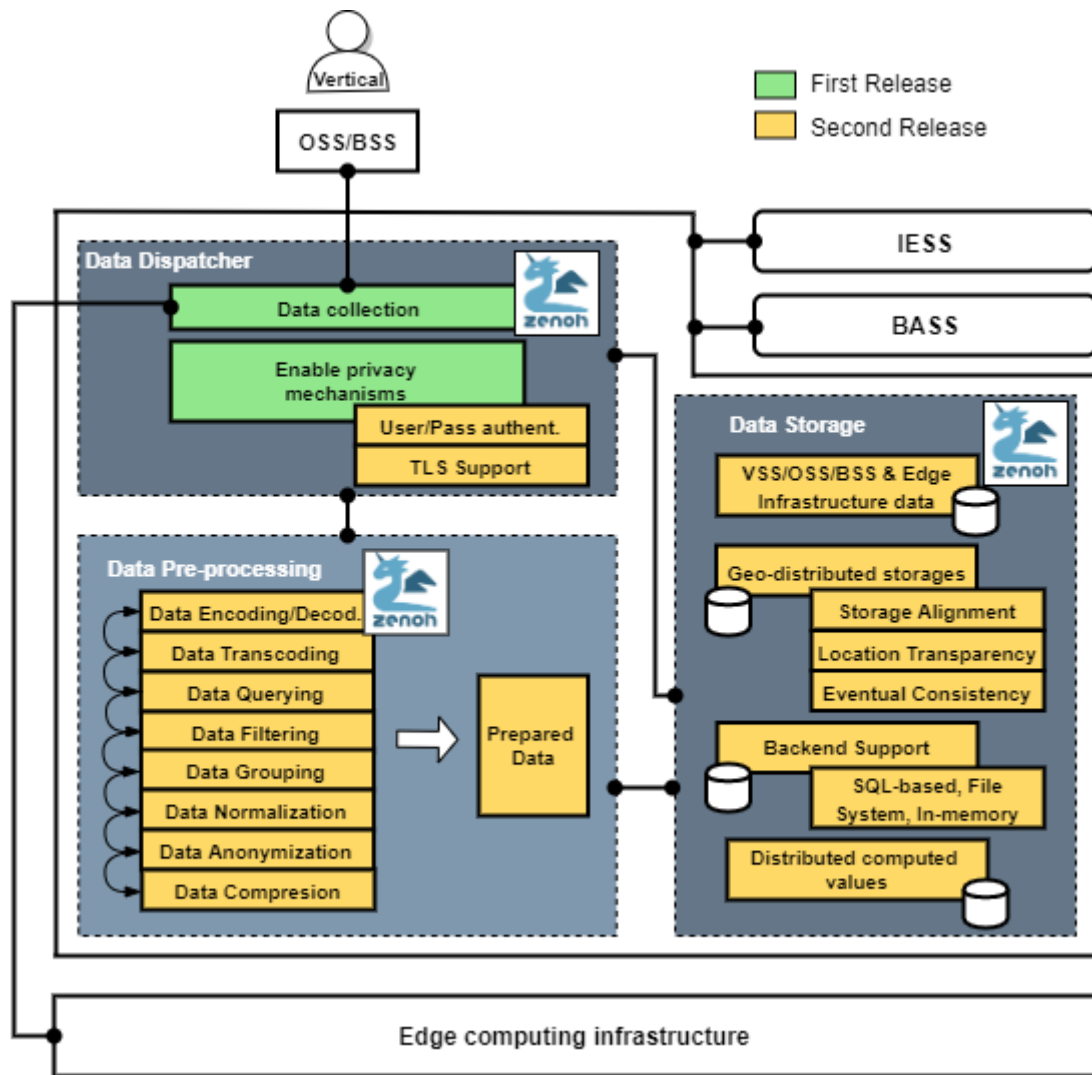


FIGURE 2-4: DASS ARCHITECTURE.

Data Dispatcher

The data dispatcher element now supports two privacy preserving mechanisms, a basic user-password authentication and TLS as a transport protocol. DASS's clients and peers can use user and password for authentication against a router or a peer. The configuration of credentials is done via a configuration file defining certain properties. Figure 2-5 shows the session establishment steps with user-password authentication.

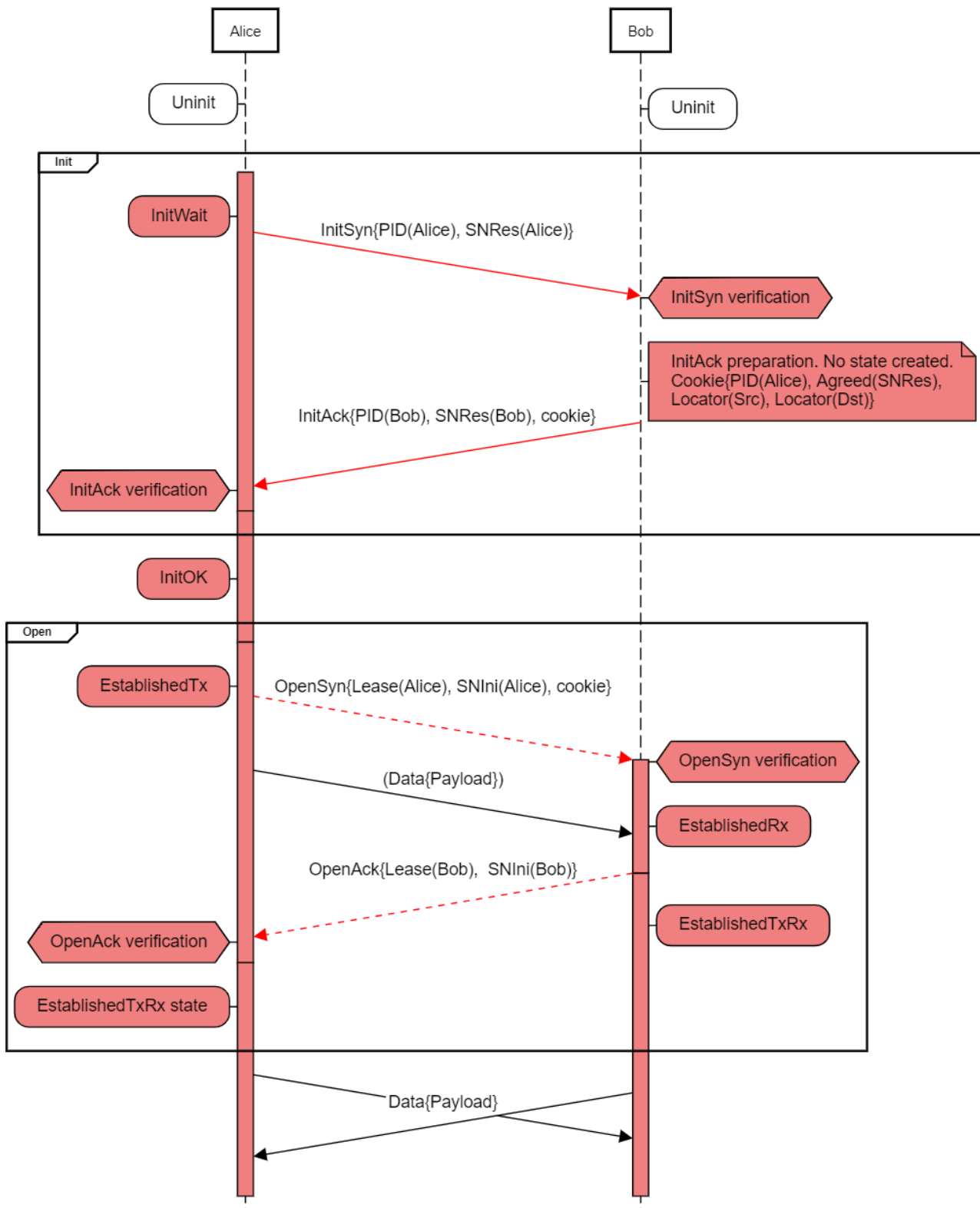


FIGURE 2-5: DASS SESSION ESTABLISHMENT USER-PASSWORD AUTHENTICATION

At the moment of writing, the only supported TLS authentication mode is server-authentication: clients validate the server TLS certificate but not the other way around. That is, the same way of operating in the web, where the web browsers validate the identity of the server via means of the TLS certificate. In

order to use TLS as a transport protocol, we need first to create the TLS certificates. While multiple ways of creating TLS certificates exist, we used Minica [6] for simplicity.

Data Pre-processing

Data pre-processing can be achieved by native support of multiple data encoding, such as JSON, Properties, Relational, Raw, etc., along with transcoding across supported formats. The DASS encoding describes the value format, allowing the DASS to know how to encode/decode the value to/from a bytes buffer. By default the DASS is able to transport and store any format of data as long as it's serializable as a bytes buffer. But for advanced features such as content filtering (using selector) or to automatically deserialize the data into a concrete type in the client APIs, the DASS requires a description of the data encoding. The current version of DASS supports the following encodings for filtering and automatic deserialization:

- **Raw:** the value is a bytes buffer
- **StringUTF8:** the value is an UTF-8 string
- **Json:** the value is a JSON string
- **Properties:** the value is a string representing a list of keys/values separated by ';' (e.g. "k1=v1;k2=v2...")
- **Integer:** the value is an integer
- **Float:** the value is a float
- **Custom:** the value is a bytes buffer with a free string allowing for instance to describe its encoding.

The DASS also defines a canonical query syntax based on URIs syntax that enables filtering and querying for a particular subset of the data. The data pre-processing element implements a distributed query representation and supports the get, subscribe and eval functionalities. The get functionality defines a selector which implements a string which is the conjunction of a path expression identifying a set of paths and some optional parts allowing to refine the set of paths and associated values.

The structure of a **selector** is composed of three parts:

- **expr:** is a path expression.
- **filter:** a list of predicates separated by '&' allowing to perform filtering on the values associated with the matching keys.
- Each predicate has the form “field-operator-value” where:
 - field* is the name of a field in the value (is applicable and is existing, otherwise the predicate is false)
 - operator* is one of a comparison operators: <, >, <=, >=, =, !=
 - value* is the value to compare the field's value with
- **fragment:** a list of fields names allowing to return a sub-part of each value. This feature only applies to structured values using a “self-describing” encoding, such as JSON or XML. It allows to select only some fields within the structure. A new structure with only the selected fields will be used in place of the original value

The subscription functionality is implemented via a **selector** and registers an interest for being notified whenever a key/value with a path matching the subscriber's **selector** is put, updated or removed on a Zenoh infrastructure. The **eval** functionality is a computation registered at a specific path. An **eval** function can be used to pre-process data on-demand, remove null values, normalize or anonymize it, also it can be used to build a remote procedure call (RPC) system.

Data Storage

The DASS also provides a storage backend plug-in API, that facilitates the integration of third parties storage technologies. At the moment of writing this deliverable the DASS supports SQL-Based backends, in-memory backend, file system backend and time series backends. The DASS's backends are managed via the DASS's **admin space** using operations on such a given resource path (e.g. `/@/router/<router-id>/plugin/storages/backend/<backend-id>`). Where `<backend-id>` is a free identifier for the backend (it must be unique per router identified with `<router-id>`).

- **Adding a backend:** this operation implies loading a new backend technology e.g., SQL-Based, in-memory, file systems, or time-series backend. Once the backend is created, the user/application can create one or several storages of that backend type.
- **Removing a backend:** this operation refers to removing a registered backend. This operation will delete all the storage within that backend.
- **Checking the status of a backend:** this operation will return the description (in JSON format) of the available backends and their related storages.

Other key innovation is that the data storage now supports zero copy by leveraging shared memory. Specifically, the Zenoh-based implementation maps a memory segment (`/tmp/zenoh/shm/pid`) in the address space of a process, e.g., App1, so that several processes i.e. App2 can read (and optionally write) in that memory segment (`/tmp/zenoh/shm/pid`) without calling operating system functions.

Zenoh then uses these shared memory zones to allocate user data and then only exchange with processes on the same memory domain the information necessary to access the data. As a consequence the overhead of sending large payload becomes constant and equal to sending the addressing information, as illustrated in Figure 2-6.

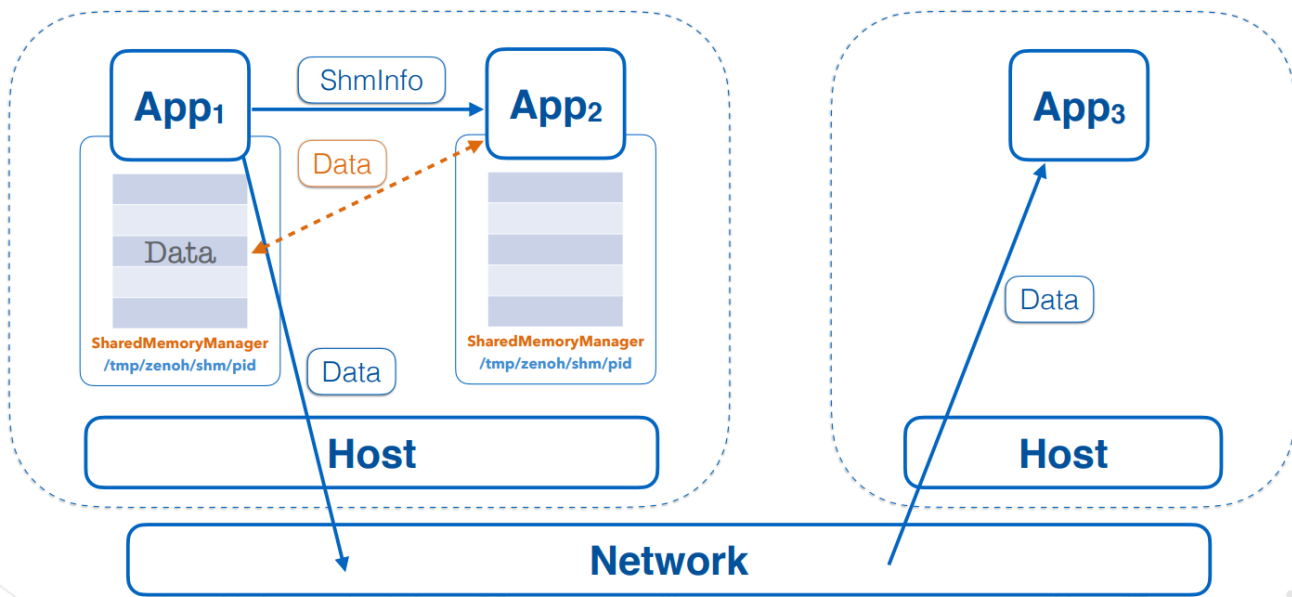


FIGURE 2-6: DASS ZERO-COPY COMMUNICATION

This zero copy mechanism can be used in robotic and autonomous driving domains, where vehicle's applications share large data between processes on the same host. In some cases these large data samples are images coming from cameras in other cases are point-clouds coming from RADARs/LIDARs. In any case, as these payloads can be several megabytes if not tens of megabytes, the transmission delay can become a bottleneck. In several of these applications, ideally we would want to pass around "pointers" to the data, but in a safe manner. Likewise, for processes that are remote we would want to transparently use the networking stack and send the actual data.

2.2.1.2. High level API based on NDN

It is worth noticing that the initial design of the DASS described below has been contributed and integrated into the Eclipse Zenoh open source project [7]. The terms DASS and Zenoh are therefore used interchangeably below since Zenoh is an actual implementation of DASS. The DASS functionality provides geo-distributed storages, unifying different kind of backed such us SQL-based, non-SQL bases, time-series, and file systems. Therefore, DASS provides a data-centric abstraction in which applications can read and write data autonomously and asynchronously. The data read and written by Zenoh applications are associated with one or more resources identified by a URI. These URIs represent a hierarchical organization of data. For example, each region comprises several houses identified by unique IDs. This results in a key structure like **/factory01/floor01/**. Moreover, each data produced by each house, can be stored in a specific key, e.g. **/factory01/floor01/room01/temperature** can be used to store the temperature reading of a specific room.

Data can be transparently accessed by the careful usage of selectors over the key space. For example, the wildcards in the key **/factory01/*/*temperature** produce as result that the temperature of every room of every house in region01 is returned, regardless where they are stored. The routing infrastructure takes care of doing the necessary pattern matching between keys, selectors, publishers, and subscribers. By properly designing the key space, it is also possible to achieve the desired level of

privacy for the data. E.g., data meant to be publicly available could be stored under a specific path (e.g., /factory01/*/public/**) and stored only on specific public locations (e.g., regional data centres). Once data is successfully retrieved, data analytics can be eventually performed.

In order to support a wide heterogeneity of scenarios, networks, and devices, we adopt a two-level protocol design as illustrated in Figure 2-7. The data pre-processing and the data storage components are implemented by the Zenoh layer. The Zenoh layer is a higher level API providing the same abstractions as the zenoh-net API in a simpler and more data-centric oriented manner as well as providing all the building blocks to create a distributed storage.

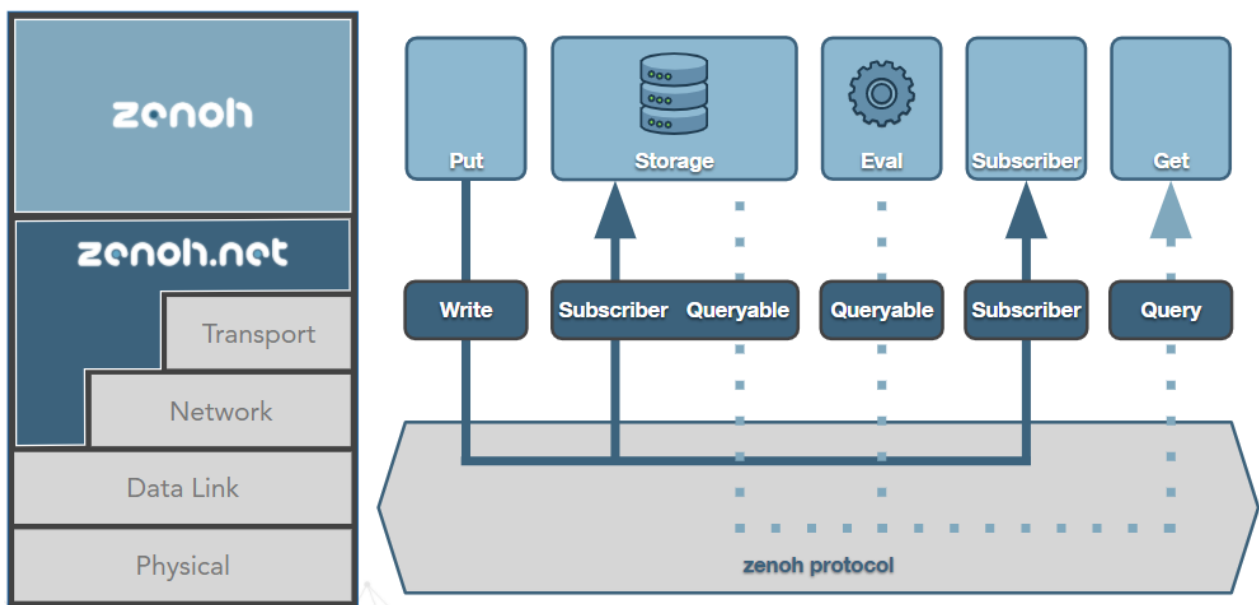


FIGURE 2-7: ZENOH AND ZENOH.NET PROTOCOL LAYERS.

The Zenoh layer is aware of the data content and can apply content-based filtering and transcoding. The key Zenoh primitives include:

- **put:** push live data to the matching subscribers and storages.
- **subscribe:** subscriber to live data.
- **get:** get data from the matching storages and evals.
- **storage:** the combination of a zenoh-net subscriber to listen for live data to store and a zenoh-net queryable to reply to matching get requests.
- **eval:** an entity able to reply to get requests. Typically used to provide data on demand or build a RPC system.

2.2.2. Business Automation Support Stratum - BASS

According to D2.1 [1], the Business Automation Support Stratum (BASS) has been conceived in the 5G-DIVE project as an evolution of the current control systems where an operator oversees the business processes' administration. The BASS provides the interface to plug OSS/BSS systems into the DEEP platform and acts as a gateway to access all of its features. Verticals can integrate their services by

describing them with high-level data models (see section 2.2.2.4), manage the lifecycle of their instances through the BASS North-Bound Interface (see section 2.2.2.2), and include their own local computing and network resources so they are managed by the BASS (see section 2.2.2.5).

Additionally, the BASS provides novel Management and Orchestration (MANO) automation for business processes, with a productionization of a Platform as a Service on the Edge. Some of the benefits include: i) not requiring highly skilled operators, ii) seamlessly optimizing the deployed services and iii) declarative vertical service control, leveraging the ability to describe desired states, so the vertical only needs to know the desired state, not how to deploy and manage it.

The BASS will automate the orchestration of the resources and their lifecycle. Besides, the BASS in this second release will be capable of verifying the end-to-end business process KPIs identifying anomalies and minimizing the business impact using the enforcement of service level agreements (SLAs). These SLAs can be ensured by leveraging the AI/ML capabilities of the Intelligence Engine Support Stratum (IESS).

Additionally, a study on the applicability of Distributed Ledger Technology (DLT) as a mechanism of the external federation support element of the BASS is presented in Appendix Section 7.1.

2.2.2.1. Architecture

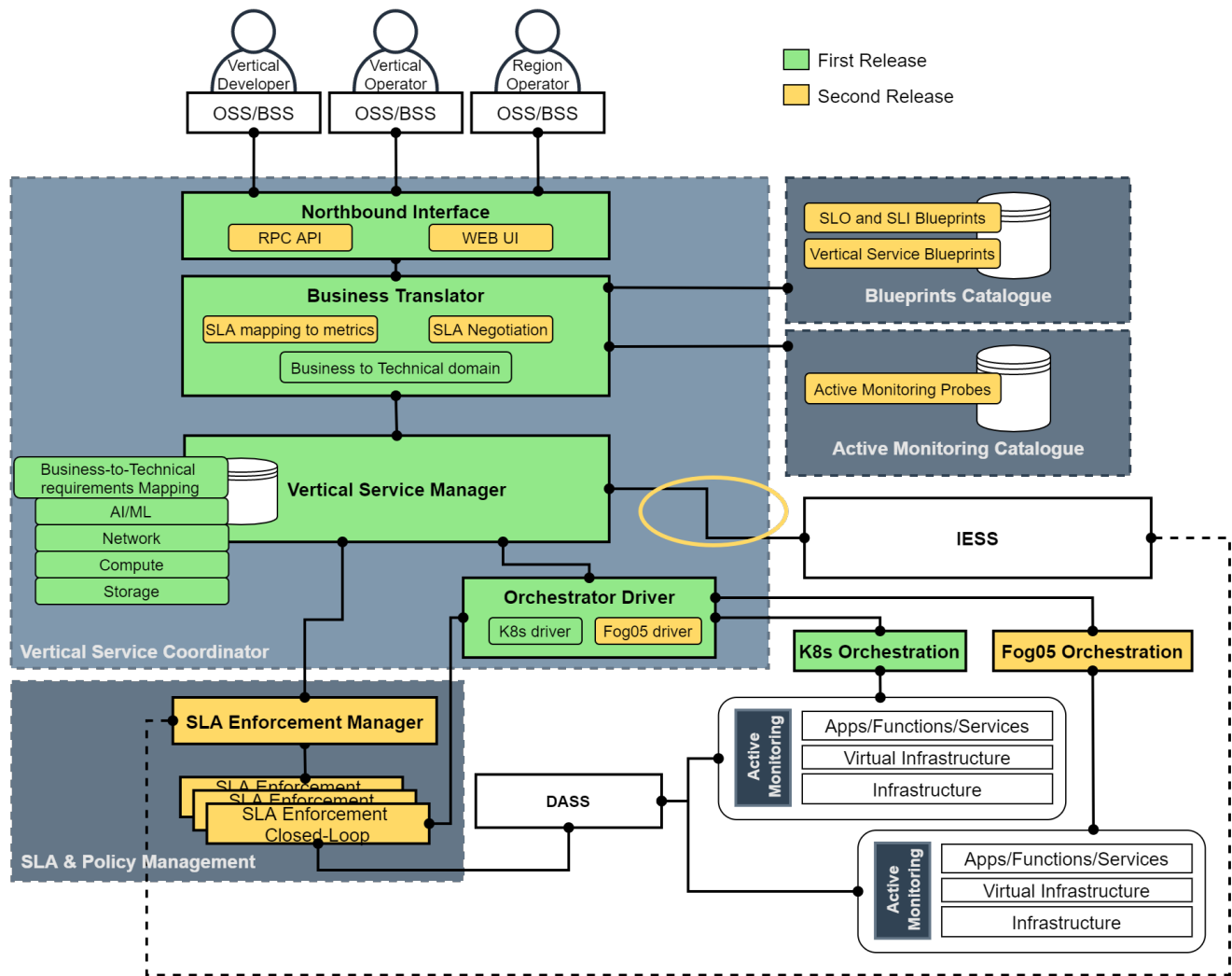


FIGURE 2-8 BASS UPDATED ARCHITECTURE

In Figure 2-8, the second release for the BASS architecture implementation can be seen. Components with green background are components already implemented during the first release of the BASS and its implementation details defined in D2.1 [1]. Components with a yellow background are component which are implemented as part of the second release of the BASS. In the following subsections, we will describe in detail each of the newly developed components.

SLA & Policy Management

As part of the second release feature of automatic vertical service life-cycle management through the SLA enforcement framework, two architecture components will be implemented from the SLA & Policy management, the SLA Enforcement Manager and the SLA Enforcement Closed-Loop.

According to D2.1 [1], the SLA Enforcement Manager, is responsible for the life-cycle management of the SLA enforcement closed-loops, in the second release:

- This component will be in charge of identifying any active monitoring requirements, selecting the optimum active and passive monitoring probes from the catalogue (according to the SLA requirements) and attaching the selected probes to the vertical service deployment.
- Additionally, it will identify the SLA enforcement capabilities, selecting the appropriate SLA enforcement model, and configuring its inputs and outputs.
- As shown in Figure 2-8 above, and as discussed in D1.3 [4] Section 2.2.3, the SLA Enforcement Manager has a privileged connection with the IESS in order to request AI/ML services directly.

The SLA Enforcement Closed-Loop, according to D2.1 [1], it is responsible of making the auto-scaling decisions over a vertical service and enforces the defined SLAs based on the monitoring information gathered via the DASS and the active monitoring, in the second release,

- This component will be capable of performing auto-scaling decisions, by using either heuristics or ML/AI based models loaded through the platform.
 - o An example of application of AI to SLA enforcement is the exemplary algorithm developed for the Digital Twin use case, where through reinforcement learning the components of a vertical service can be automatically scaled vertically, to comply with the Service Level Objectives (SLOs). In the second release this model will be integrated jointly with the evaluation of other models trained to enforce SLOs in most of the pilots.
- Decision making models will be fed with the required active or passive monitoring information from the infrastructure and/or vertical services.
- The closed-loop action pool will be determined by the Orchestrator driver, for the second release there will be support for Kubernetes and Fog05.

Orchestrator Driver

For the second release the Orchestrator driver component will be enhanced with the support for Kubernetes advanced features and Eclipse Fog05 [8], which is an End-to-End Compute, Storage and Networking Virtualisation solution.

Northbound Interface

The northbound interface was implicitly presented in the previous version of the architecture and included a RPC interface implemented over the HTTP protocol. A Web User interface has been added, allowing the vertical to define, manage and monitor their vertical services in a more user-friendly environment. Additionally, the RPC API has also been extended to support additional workflows, actions, and roles, such as the Vertical Service Blueprints, Descriptors, and authentication and authorization different user roles such as the Vertical Service Developer, Operator and the Region Operator.

Business Translator

The Business translator component has been extended to support the mapping of SLAs to KPIs and the ability to Negotiate SLAs from a pool of providers. Additional details of the implementation are provided in Section 2.2.2.7.

Blueprints Catalogue

The Blueprints Catalogue has been extended to store and provide i) SLO and SLI blueprints to the vertical and the infrastructure providers, ii) Vertical service blueprints.

Monitoring Catalogue

The Monitoring Catalogue stores and manages the collection of active monitoring probes, either for basic or advanced metrics. Reusable probes are available to Vertical Services in order to collect common metrics, while other probes are customizable in order to extract metrics that are more service-specific.

BASS - IESS interface

As part of the BASS and IESS interface, the BASS includes additional logic to support the training of intelligent components, by means of the AI Component Entity. On the other side, the IESS supports a descriptor for defining training jobs and their runtime, called Training Component, and an inference descriptor called Inference Application Packaging. All these new entities coordinate the state and data exchange between the BASS and the IESS, seamlessly offloading AI decisions to the IESS and deployment and business decisions to the BASS.

2.2.2.2. Northbound Interface

The BASS Northbound interface offers two ways of interaction: an RPC interface implemented with HTTP and a Web interface. While the RPC interface is more suitable for machine-to-machine communications and programmatic interactions, the Web User Interface is more pleasant for the interaction of human users. Furthermore, the Web User Interface provides an endpoint to the OpenAPI document for the RPC interface.

The Web user interface includes the following features:

- **Vertical Service Blueprint loading.** It allows the vertical to load a vertical service blueprint to the BASS Vertical Service Blueprint Catalogue.
- **Visualization of the blueprints stored in the Vertical Service Blueprint Catalogue,** including blueprints shared by other users.
- **Generation of a Vertical Service Descriptor from a Vertical Service Blueprint.** It enables the vertical service operator to select the corresponding vertical service blueprint, and fill and/or override the deployment parameters.
- **Vertical Service life-cycle manager.** It allows the vertical service operator to manage the whole vertical service states in a declarative way, expressing the next state for the vertical service to move. It allows also to manage the vertical service components, individually. Additionally, it allows to monitor in real-time the state of every component in the vertical service.
- **Automatic status update for the Vertical Services.**
- **Vertical Service Component update.** It gives the capability to change or update the vertical service component, either to upgrade or downgrade its version, or change its deployment and operational parameter.
- **IESS training state monitoring and results**
- **Vertical Service AI Component life-cycle management**

- **Login with multiple roles**
- **Management of regions**

The Web User Interface is implemented with plain Javascript, HTML and CSS. For the CSS we selected the popular Twitter Bootstrap framework [9] providing support for responsive layouts and mobile devices out-of-the-box. To interact with the Business Translator the Web UI uses the RPC API mentioned above.

Figure 2-9, Figure 2-10, Figure 2-11, and Figure 2-12 show some screenshots of the Web UI.

The screenshot displays the 5G-DIVE Business Automation Support Stratum (BASS) web interface. The top navigation bar is blue with the 5G-DIVE logo on the left and the text 'Business Automation Support Stratum (BASS)' on the right. A dark sidebar on the left contains a menu with items: 'Main', 'Vertical Services', 'Add New Service', 'Blueprints', 'Blueprints Catalogue', 'Add New Blueprint', 'DEEP Regions', 'OPTIONS', 'Messages' (with a blue badge showing '5'), 'Help', 'OpenAPI Definition', and 'Powered by Telcaria'. The main content area has a breadcrumb trail: 'Blueprints Catalogue / test-blueprint / Create Service from Blueprint'. The title 'Create service from test-blueprint' is prominently displayed. Below the title, there are three input fields: 'Service Name' (containing 'test-service'), 'Region' (with a placeholder 'Enter the region where to deploy'), and 'test-component.imageRepository' (containing 'nginx:1.18.0'). A note 'It must follow RFC-1123 specification' is below the Service Name field, and 'Optional, otherwise goes to default region' is below the Region field. A fourth field, 'test-component.exposedPorts' (containing '80'), is also present. A blue button labeled 'Create New Vertical Service' is at the bottom of the form.

FIGURE 2-9 CREATION OF A NEW VERTICAL SERVICE FROM A BLUEPRINT

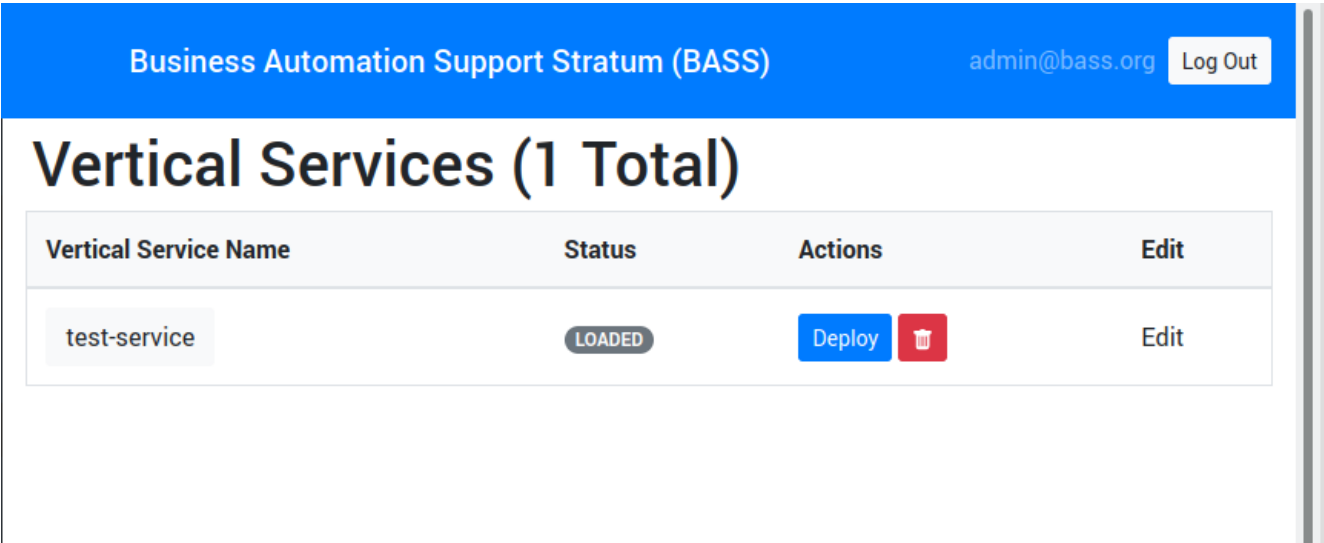


FIGURE 2-10 VISUALISATION OF THE LIST OF SERVICES

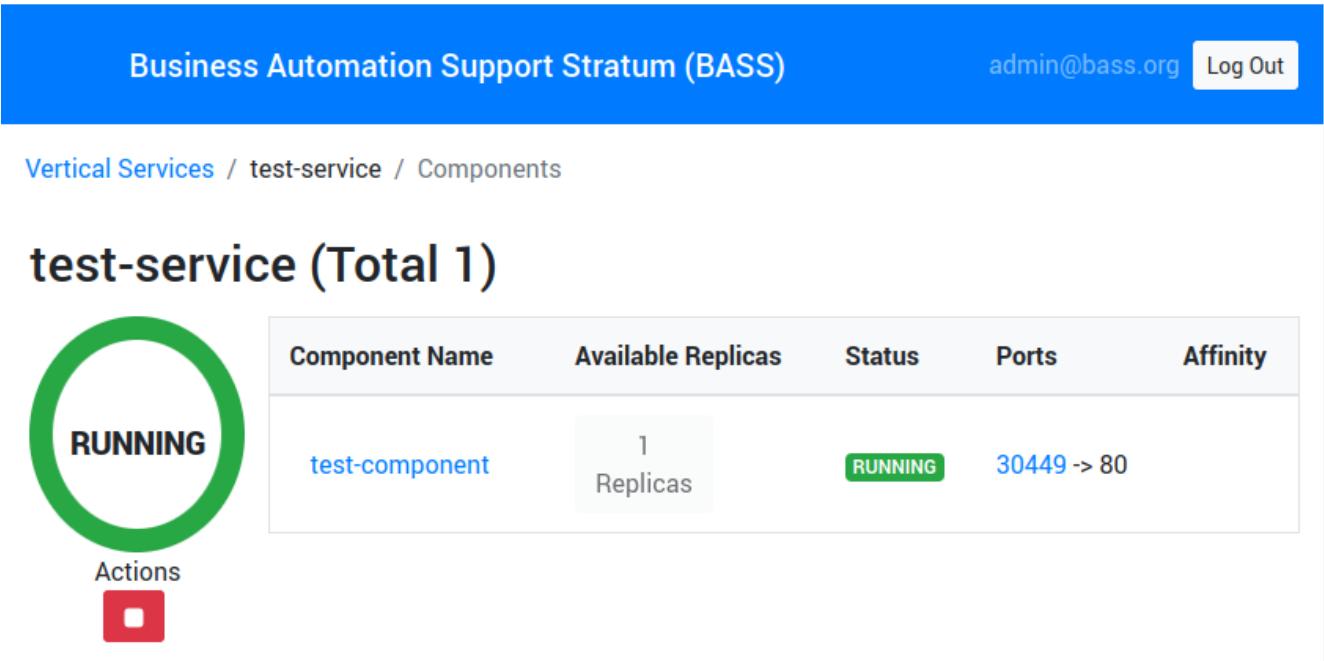


FIGURE 2-11 VISUALISATION OF A RUNNING SERVICE WITH DETAILS ON ITS COMPONENTS

Business Automation Support Stratum (BASS)

Vertical Regions (4 Total)


Region Name	Region Type	Region Affinities
fog05-test		
kubernetes-test		os -> linux location -> edge,fog instance-type -> k3s arch -> amd64
kubernetes-test2		os -> linux instance-type -> k3s arch -> amd64

FIGURE 2-12 VISUALISATION OF VERTICAL REGIONS EXAMPLE VIEW

2.2.2.3. Role Based Access Control

This section discusses the implementation of the features of vertical service abstraction, designed and presented in D1.3 [4] Section 3.1.1.

The users interacting with the BASS can be assigned one or more roles, determining the resources and operations they have access to. There are three main roles in the BASS:

- Vertical Developer
- Vertical Operator
- Region Operator

The implementation of roles, authentication and authorization in the BASS is implemented with Spring Security [10] the de-facto standard way to secure Spring-based applications. According to the framework specifications, the entity representing a vertical (any user of the BASS) implements the UserDetails interface, establishing generic methods to load user-specific data. The interface enforces the inclusion of a username and a password, as well as a collection of authorities, the roles assigned to each user. Users' passwords in the BASS are stored securely by using the bcrypt password hashing function.

For what concerns authentication, the Spring Security framework adds a chain of security filters that intercepts all the requests incoming at the BASS NBI. We use the default UsernamePasswordAuthenticationFilter for the very first authentication of the user. Username and password are provided in the Authorization header of the HTTP request for login, in base64 encoding, following the 'Basic' HTTP Authentication Scheme (RFC 7617) [11]. The secure transmission of credentials over the wire is realized by means of a TLS connection. After the initial login, the BASS generates a JSON Web Token (JWT) (RFC 7519) [12] with a relatively short expiration time to be used by the client for subsequent communications. The JWT can be included in the Authorization header if the client access programmatically to the NBI or in a cookie if the interactions happen through a browser. A custom Spring Security filter validates tokens and manage this kind of authentication

method. The `JwtTokenFilter` takes precedence over the username and password authentication to avoid the repeated transmission of the user credentials.

The authorization to access BASS resources has two control levels on the NBI. At the first level, a role check is performed: the user issuing the request must have the appropriate role to access the resource. This is implemented by using the '@Secured' annotation (JSR 250) [13] on NBI methods. For example, a user which is assigned only the role of Vertical Operator does not have access to write and update operations on Vertical Service Blueprints or Vertical Regions. The second control level checks the ownership of the resource by the user. Some resources managed by the BASS, like regions and Vertical Service Blueprints are accessible from all users for reading, as long as the user has the appropriate role. Write and update operations are permitted only if issued by the owner of the resource.

2.2.2.4. Templating Services with the Vertical Service Blueprint

This section discusses the implementation of the features of vertical service abstraction, designed and presented in D1.3 [4] Section 3.1.2.

The Vertical Service Blueprint (VSB) is a template of a vertical service that can be partially customized. With respect to the Vertical Service Descriptor (VSD), the VSB provides a more generic definition of the vertical service in order to separate as much as possible the functional definition of the service from the configuration needed to run it on a specific environment. The latter, that we may see as the deployment specific configuration, is encoded into the form of a list of parameters. The parameters can have two effects: first, they can be used to override some values of the VSD and its components in order to adapt it for the specific deployment (e.g., set a URL or a password), second, they can add additional information to express qualitative aspects wanted for the service (e.g., quality of service, geographic availability).

The implementation of the first group of parameters, for overriding some parts of a VSD, falls into the problem category of data templating. Given a data structure, possibly very complex and nested, there is the need to dynamically change some parts of it at runtime. For example, Ansible [14] a configuration management software, uses data templating to customize system configurations and perform variable substitution right before applying them on the targeted system. Some popular templating solutions we can mention are the Jinja templating engine [15], the Jsonnet templating language [16], and YAML anchors [17], even if very limited with respect to the previous two. The three technologies just mentioned have been evaluated to be applied to the VSB but they have been found unsuitable. Indeed, there is a lack of support for the integration with the Java language and moreover, they are focused on generating configuration files in plain text adding extra serialization and deserialization operations in our use-case. A custom solution has been implemented to enable parameter overriding in the VSB by leveraging a powerful feature of Java (provided also by many other programming languages), reflection [18] allowing a program to inspect and change the behaviour of its classes, interfaces, methods, and fields at runtime. We use these properties of reflection to declare parameters in the VSB in the form of formatted string and inspect the VSD model in order to retrieve or set the corresponding field values.

```

{
  "name": "web-service",
  "components": [
    {
      "name": "web-ui",
      "numReplicas": 1,
      "imageRepository": "web/web:v1.0.0",
      "exposedPorts": [
        80
      ]
    },
    {
      "name": "postgresql-db",
      "numReplicas": 1,
      "imageRepository": "postgres:9.6"
    }
  ]
}

```

FIGURE 2-13 VSD EXAMPLE

Figure 2-13 shows a trivial example of a VSD for a web application, composed of simply two components, the web part and a database. A simple parameter to customize this service, by changing the image of the web part is “web-ui.imageRepository”. The declaration of the parameter in the VSB enables the overriding at deployment time. When the BASS receives the VSD, together with the parameter above (and its new values), it inspects the VSD data structure recursively thanks to Java reflection, retrieves the corresponding field, and substitutes the value.

The solution, while being simple, has two main advantages. Potentially, any field of the VSD can be parametrized, enabling extreme flexibility and freedom in the VSB definition by the Vertical Service Developer. On the other side, once the list of parameters is established, the other fields of the service are protected against any change. This prevents the Vertical Service Operator to perform potentially destructive changes to the service since their action is limited to the list of declared parameters.

When a VSB is onboarded, the BASS validates its parameters by checking that they target existing fields. The validation logic is integrated into Hibernate [19], a Java framework implementing the Bean Validation Reference [20] and reuses the Java reflection features described above.

The second group of parameters in the VSB includes several qualitative aspects to customize the deployment of the service.

- SLA templates (SLI and SLO)
- Parameters for AI/ML requests to the IESS
- Geographic constraints
- Lifetime settings

SLA related parameters are described in Section 2.2.2.7. Parameters for the IESS are used to customize the AI Component descriptor discussed in Section 2.2.3.2. The geographic constraints allow for the selection of the location of the resources targeted for the deployment. The selection is performed by

means of human-friendly definitions, like “continent”, “country”, and so on. The BASS selects the region that best matches the constraints provided and properly orchestrate its resources, as described in Section 2.2.2.5. Finally, the lifetime settings specify the period of time the service should remain active. At the end of this period the BASS will automatically stop the Vertical Service in order to release allocated resources, but it will leave it in a “loaded” state in order to quickly redeploy it in case of need.

Since blueprints include a more generic description of the service and hide deployment-specific parameters through templating, they can be shared between all the users of the BASS. A trivial Boolean field establishes if a VSB is shared or not and it can be changed only by the author. By default, at the moment of its creation, a VSB is not shared in order for the Vertical Service Developer to perform several iterations of improvements and fixes on its blueprint without worrying about disclosing sensitive data. Once the VSB is ready it can be marked as shared to make it available to other users. All blueprints are collected in a catalogue at the BASS implemented by means of MongoDB, a popular document-oriented database. In fact, being the blueprints encoded as documents in JSON structured format, they perfectly fit the document-oriented data model. Furthermore, MongoDB includes distribution and replication features that can be used to avoid or mitigate data losses on the blueprint catalogue.

2.2.2.5. Multi-region orchestration

Support for managing multiple regions in the BASS is implemented by providing drivers for several resource orchestrators. Previously, the BASS was only able to be configured to manage one single Kubernetes region making use of the driver developed for this resource orchestrator.

The Region Manager Service oversees managing the different regions instantiated in the BASS. During runtime they are stored in-memory but also backed to the internal DB used by the BASS. An initial set of regions can be specified in the BASS configuration file. The BASS controller has been extended to support the creation and deletion of regions by a Vertical user with the Region Operator role. Offers great flexibility as it supports deploying different components of the same Vertical Service to different regions, allowing scenarios where different EFS and OCS are involved.

Additionally, support for Fog05 has also been implemented in the form of a new kind of OCS driver in the BASS. Figure 2-14 shows graphically the driver based architecture to manage both K8s and Fog05 clusters. Each controls its own set of computing, networking and storage resources, used to deploy Vertical Services.

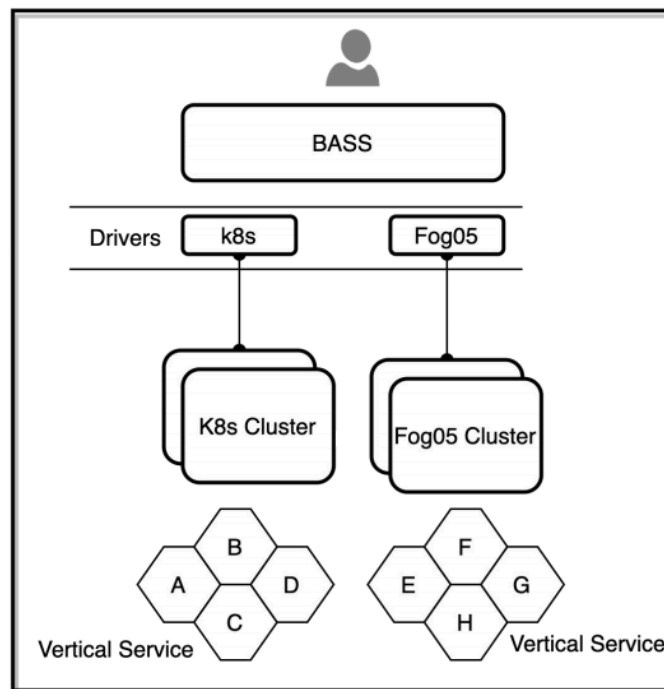


FIGURE 2-14 BASS AND OCS INTEGRATION

The descriptor used to create new regions in the BASS is composed of four parameters:

- **Name:** unique identifier of the region
- **Region Type:** refers to the driver needed to manage this region, now there is support for Kubernetes and Fog05 regions.
- **Region Config:** parameters needed to configure and manage the region, related to the driver used to manage the OCS like, for example, connection related parameters.
- **Geographic information:** location of the resources managed by this region in terms of continent, country, city.

The “Region Config” parameter has been implemented using polymorphism. This means that there can be different types of descriptors that conform the configuration of a region, in this case there are two types of region configurations, one for Kubernetes and the other one for Fog05.

Nevertheless, both regions share these parameters, related to BASS configuration for that region:

- **maxWaitTime:** time in seconds that the BASS will wait until a Vertical Service Component is reported as “ready” status once its deployed in the EFS.
- **maxRetries:** max number of times that the BASS will redeploy a Vertical Service Component in case a “soft error” is reported by the OCS.
- **backoffTime:** time in seconds that the BASS will wait to redeploy a Vertical Service Component in case of “soft error”.

For Kubernetes regions these are the parameters that can be used to configure the driver:

- **URL:** address of the Kubernetes API of target cluster.
- **User:** user who will be used to authenticate with the Kubernetes API.
- **Password:** password used to authenticate.

- **Kubeconfig:** location of the “kubeconfig” file with all the connection details of the target cluster, including certificates for secure connection.
- **SSL:** indicates whether to validate cluster certificates or not.
- **Debug:** indicates if extra logs will be captured in the driver connections to the Kubernetes cluster.
- **ConnectionTimeout:** time in milliseconds to wait for stale connections to the target cluster.
- **ReplicaMinAvailability:** percent of available replicas that should be atleast running before marking the deployment as “failed”.
- **InCluster:** special parameter that tells the BASS to try to connect the Kubernetes cluster using the “serviceaccount” provided in the BASS deployment. This is only for the case when the BASS is deployed in the same Kubernetes cluster to manage as a region.

Meanwhile, for Fog05 there is only one customizable parameters, *host* and *port*, that point to the Fog05 instance HTTP endpoint available for the BASS to connect to.

Internally, the BASS uses a different data structure to store and manage the regions. The structure for a region is composed of these attributes:

- **Id:** internal identifier of the region.
- **Name:** name of the region.
- **Type:** type of the region, e.g., either Kubernetes or Fog05.
- **Driver:** instance of the orchestrator driver object, used to manage the VS deployments in the region’s OCS.
- **AffinityLabels:** additional information or metadata about the region.

During runtime, additional information is probed from the regions. This information allows the BASS to have an idea about the capabilities of the regions regarding the workloads that can be deployed there and can be mapped to a set of affinities that the Vertical can impose in the Vertical Services to deploy. An example would be a Vertical Service requiring specific hardware requirement like processor architecture or making use of a particular device. This information is stored in the “AffinityLabels” attribute of the region object. Additional details on the BASS and OCS integration workflow can be found in the Appendix Section 7.2.

2.2.2.6. Active Monitoring Framework

The Active Monitoring framework provides full monitoring pipeline support to the Vertical Services deployed by the BASS. The BASS will leverage the ingestion of application metrics of a Vertical Service deployed, storing the data in the DASS, and offering the data stored to the Vertical. Ingested data can also be provided to the SLA enforcement framework to enforce the negotiated SLAs by considering the extracted Vertical Service metrics.

The BASS exposes the Monitoring Catalog to the Vertical, containing all the metrics and information that can be automatically extracted from the applications.

The catalog is divided in two parts:

- **Basic metrics:** the common infrastructure metrics are automatically available to all verticals interacting with the BASS and the set of probes available depends on the capability of the resource orchestrator. Some of the metrics that these probes can capture are:
 - CPU usage
 - RAM usage
 - Network usage
 - GPU usage
- **Advanced metrics:** in this case the captured metrics depends on the deployed application. These metrics involve target different application technologies or metrics exposed by the application itself. The Vertical can also specify custom metrics that can be gathered from its application and which type of probe is needed to extract that information. Some examples are:
 - Metrics exposed by the application using the available Prometheus client libraries [21].
 - Database applications like MongoDB, Redis, MySQL, etc.
 - Messaging applications like RabbitMQ, Kafka, etc.
 - Latency between different application or services.

The metrics to collect are defined in the Vertical Service Descriptor in a per-component basis. Only some of the “basic” metrics, like CPU and RAM usage, can be defined at a higher level, in that case the metric will be collected for each of the components inside the Vertical Service.

The monitoring stack, deployed initially by the BASS operator because its dependant on the Region Operator is composed of these services:

- **Vector** [22]: Used to recollect application logs.
- **Telegraf** [23]: Collects metrics at EFS level with enriched information about the running services, for example, Kubernetes related metadata.
- **InfluxDB** [24]: Stores the collected metrics and logs. Belongs to the DASS.

This stack is used for recollecting most of the “basic” metrics, for the “advanced” metrics specific probes are then deployed and configured by the BASS to recollect metrics in a custom and in a per-specific-component way.

Then, the BASS interacts with the DASS to extract the recollected data and make it accessible to the Vertical or to the SLA Enforcement framework.

For deploying the advanced probes in Kubernetes regions, the sidecar pattern [25] is used, this pattern involves deploying multiple containers in a same deployment, with the extra containers fulfilling specific functions, like monitoring the main container. To deploy this monitoring sidecars BASS makes use of Telegraf Operator [26], that takes care to manage and instantiate this extra monitoring containers based on metadata configured by the BASS for the main container, mapped from the Vertical Service Component. In Figure 2-15 the architecture and a simplified general workflow for the Active Monitoring component are represented:

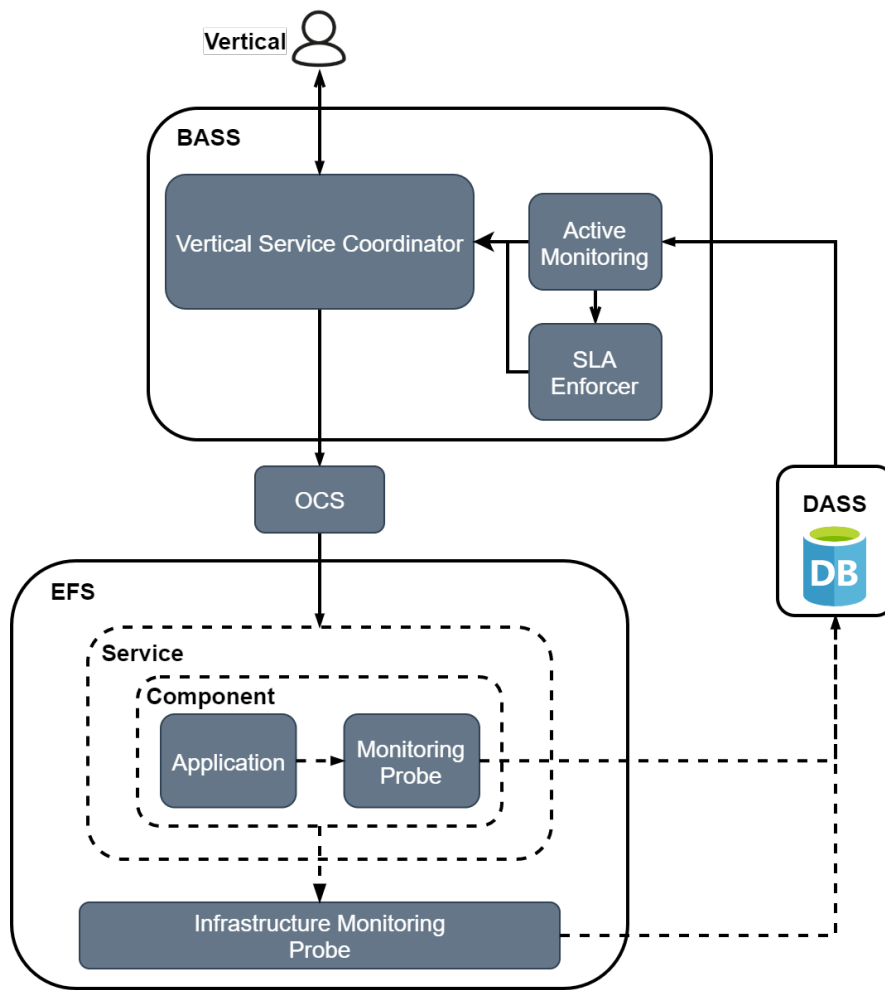


FIGURE 2-15. ACTIVE MONITORING SIMPLIFIED WORKFLOW

2.2.2.7. SLA Negotiation and Management

The Blueprints Catalogue offers a collection of generic and reusable SLO and SLI. As discussed in D1.3 [4] Section 3.2.2, the Vertical Service Blueprint includes the list of SLI and the relevant set of SLO defined on top of them. SLI and SLO are hence usually very specific to the service they are defined to. Anyway, the BASS offers a catalogue of generic SLO and SLI that can be potentially applied to any service, in order to support the developer in the definition of the Vertical Service Blueprint. For example, a generic SLI applicable to many vertical services can be based on the CPU usage of one component and it could include the following information:

- Name: Service Load
- Component: Component A
- Metric: CPU usage
- Formula: Average of CPU usage over 1 minute

The Vertical Developer can customize the SLI in order to target its component. The catalogue can then offer a set of generic SLO defined on top of the previous SLI. For example, we can define the following two SLO:

- Name: Critical Service
- SLI: Service Load
- Target: < 70 %
- Name: Regular Service
- SLI: Service Load
- Target: < 90%

The first SLO defines a more demanding target with respect to the second one. The Vertical Developer can directly reuse the SLOs proposed by the catalogue, customize them, or even define new ones.

The catalogue of SLO and SLI implements a catalogue of SLA Templates pre-negotiated between the DEEP platform and the providers of the computing, network and storage resources. In the case of customization or definition of new SLI and SLO, a further iteration of the negotiation process with the providers may be required. See also D1.3 [4] Section 3.2.1.

When the SLI and SLO have been defined, and included in a Vertical Service Blueprint, several instances of the same service can be created, each one with its own SLA. The Vertical Service Operator, in charge of deploying and managing vertical service, simply selects the set of SLO that best fit the performance requirements and budget availability of the service instance to be created. The Vertical Operator is not required to deal with the technical details of the service components and their related metrics. They simply defines the business objectives for the service deployment and the Business Translator (see Figure 2-8) takes care of mapping and translating the request into technical requirements for the infrastructure. See also D1.3 [4] Section 3.2.2.

2.2.2.8. SLA Enforcement Framework

As introduced in D1.3 [4] Section 3.2.3 the SLA Enforcement framework is implemented in the BASS as a mechanism to guarantee the fulfilment of the negotiated SLAs between the Vertical and the platform.

Implementation wise a new component has been defined, the SLA Manager, in charge of the following tasks:

- Tracking the available SLIs, region aware, that can be used to create and define a SLA for an specific application, using the Active Monitoring component to achieve this.
- Negotiation of the SLAs with the Vertical while considering region capabilities and metrics.
- Manage the lifecycle of implemented SLA Enforcers, providing the complete SLA enforcing closed-loop and configuring them.
- Application of enforcing actions, dictated by the deployed SLA Enforcers through the Vertical Service Coordinator and the Orchestrator Driver components.
- Tracking the fulfilment status of the negotiated SLAs, providing the data to the Verticals through the BASS web GUI

Regarding the framework for the SLA Enforcer component, the one with the real logic of enforcing the SLOs and choosing enforcing actions, two different interfaces have been defined, which support the main enforcing component:

- Input interface: provides the retrieved SLIs of the target SLOs to be enforced from the SLA Manager.
- Action interface: provides the valid set of enforcing actions that can be done at a time.

Both interfaces are built as REST interfaces, using the OpenAPI v3 specification [27] and Swagger [28] for generating the documentation, leveraging and making it easier to build a middleware between both interfaces with the enforcing logic, retrieving and applying actions based on the technical requirements and capabilities of the enforced application.

2.2.3. Intelligent Engine Support Stratum - IESS

The Intelligence Engine Support Stratum (IESS) is an Artificial Intelligence Platform which uses data-driven algorithms to make predictions, classifications, and decisions. This provides a tool kit to develop and train intelligent models at the Edge/Fog.

The IESS offers AI/ML related services for the vertical services managed by the BASS. It manages both the training of the model, as well as the packaging of the latter into a minimal application (microservice) capable of serving prediction results.

For model training, the IESS abstracts the interaction with several AutoML engines and AI frameworks and it automatically selects the proper engine or framework based on contextual information in the received request from the BASS. When trained, the model is packaged and stored in a catalogue in order to be reused for future deployments of the same vertical service. If retraining is requested, a new model is going to be trained and stored.

To the best of our knowledge, there is no automated flow of AutoML/AutoAI against target accuracies or losses in the market, and the possibilities of offering distributed training on integrated software is very limited. The IESS provides a pluggable AutoML/AI platform to that integrates into the DEEP platform enriching its features.

In the following subsections we discuss in more details the several features offered by the IESS.

2.2.3.1. Architecture

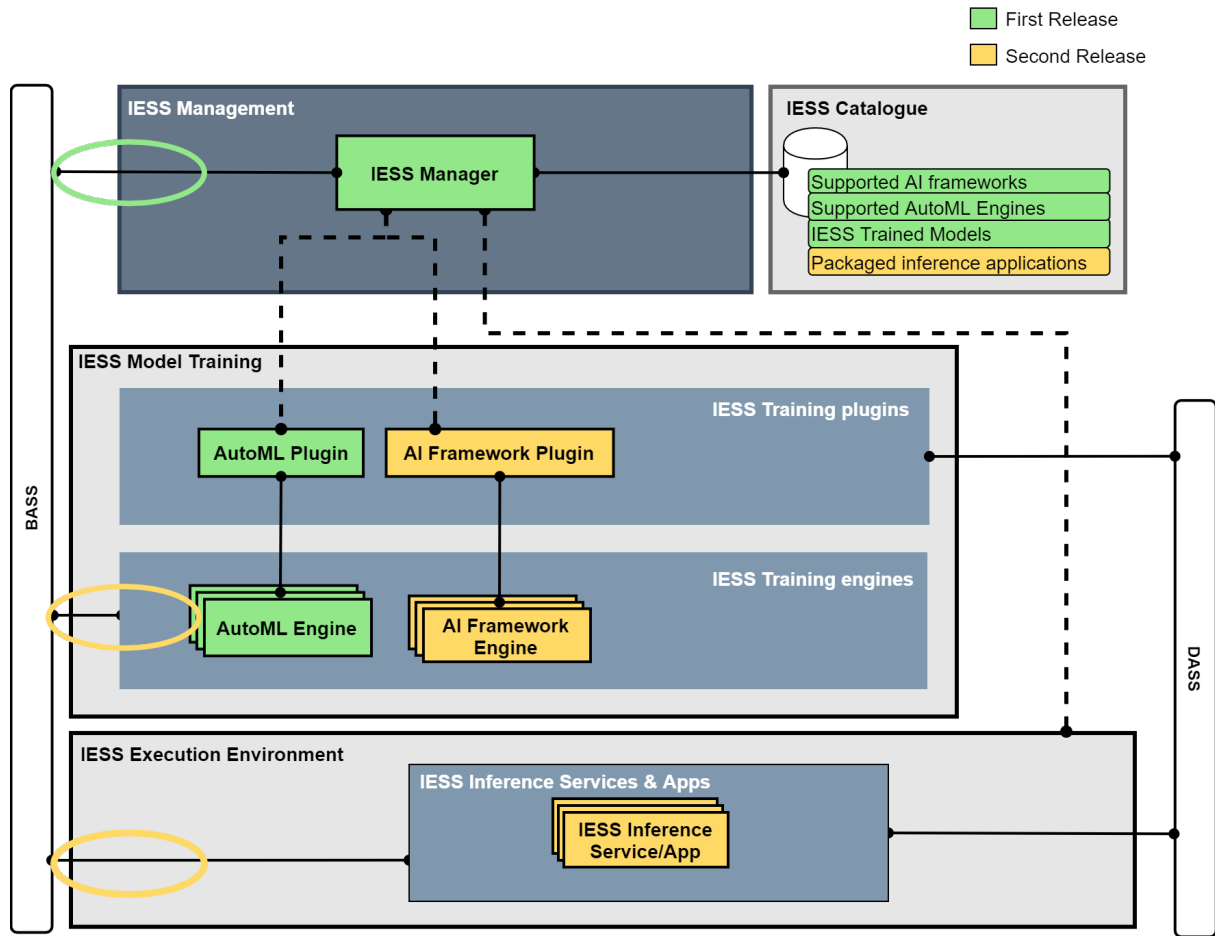


FIGURE 2-16 IESS UPDATED ARCHITECTURE

Since the first release, presented in D2.1 [1], minor modifications have been applied to the IESS architecture in order to simplify the interactions between the logical entities of the IESS and make it better understandable for the reader. The functional features of the IESS have not been changed. Components with a yellow background are components which are implemented as part of the second release of the IESS.

With respect to the architecture presented in Figure 2-24 in D2.1 [1], the IESS Manager retains its role of main entity inside the IESS, in charge of controlling and coordinating all of the operations and interactions of the other entities. The IESS Manager directly interacts with the IESS Catalogue in order to persistently store and retrieve information, such as the supported AI Frameworks and AutoML engines together with their features, pre-trained models to be reused, packaged inference applications. As we can see from the figure, it controls the two main areas of operation of the IESS (shown as big grey boxes):

- IESS Model Training: the logical portion of the IESS related to the training of AI/ML models.
- IESS Execution Environment: the logical portion of the IESS related to the packaging of inference applications and their offering to the BASS.

In the previous version of the architecture presented in D2.1 [1], the IESS Model Training part was managed by a dedicated, more specialized component. The development process suggested that having the IESS Manager as a single controller simplified the whole operation and workflow. The IESS manager manages a set of IESS Training Plugins, selecting the most appropriate one for each request received by the BASS and makes use of each plugin features to deploy the corresponding IESS Training Engine, in charge of actually carry on the training of a model. The IESS does not manage any computing or networking resources: training engines as well as inference apps are deployed by the BASS leveraging resources managed by the latter. In fact, the BASS offers advanced orchestration features (i.e., distributed deployments) and specialized resources (i.e., GPU equipped nodes for faster machine learning training). The interaction between IESS and BASS is then bi-directional, with each component offering its services and features to the other and avoiding duplication of functionalities and over-complication.

With respect to the architecture presented in D2.1 [1], it has been made clear that the IESS Manager also controls the portion of the IESS Execution Environment, that offers features to package trained AI/ML models into inference applications and offering them to the BASS for deployment, alongside of the vertical service requesting them.

2.2.3.2. [Serving Intelligence Requests From the BASS](#)

The BASS uses two different kind of descriptors to define the components inside of a VSD, one for regular (non-AI powered) components and the other one for AI components. The AI component descriptor requires the BASS to interact with the IESS and to employ more complex workflows in order to achieve the deployment.

The descriptor for an AI Component contains these extra parameters with respect to a regular component:

- Dataset: Endpoint to download the dataset from.
- AutoAIPlatform: AutoML platform to use, for now only H2O.ai is supported.
- AI Type: type can be either “classification” or “regression”.
- Selected algorithm: specific algorithm to use from the pool of algorithms provided by the AutoML platform.
- Min Loss: minimal loss the model needs to have to be deemed a valid model, if invalid the model will be retrained.
- Min Accuracy: minimal accuracy the model needs to have to be deemed a valid model, if invalid the model will be retrained.
- Column Predict: target column of the dataset used for the prediction.
- Max seconds training: max number of seconds that the model can be trained.
- Max training retries: max amount of training attempts until the model’s loss or accuracy meets the vertical’s requirements.

The full pipeline of an AI component consists of three main phases: training, packaging and inference. The BASS interacts with the IESS to create the request to manage these AI Components. The full workflow consists of the following steps:

1. The BASS forwards the AI Component descriptor to the IESS.
2. The IESS maps the request and prepares a training component in order to train the AI model. The IESS sends back to the BASS a request to instantiate the training component.
3. The BASS instantiates the training component and notifies the IESS when is ready.
4. The IESS sends the training request to the deployed training component and waits until training finishes.
5. Once training is finished, the produced model is uploaded to the catalog and the training component is uninstalled.
6. The IESS creates the inference application from the trained model. Also the application is uploaded to the catalog.
7. The IESS notifies the BASS that the training and packaging phases have completed successfully and that the inference application is ready to be deployed.
8. The BASS deploys the inference application which contains the trained model and API endpoints to generate predictions by the Vertical or by another application. This realizes the inference phase.

2.2.3.3. Supported ML Platforms

The IESS includes a plugin system to support the interaction of several AutoML engines and AI frameworks. The system is designed to be easily extensible, in order to enable the future addition of new engines and hence new features to the IESS.

As explained in D2.1 [1], in its first version, IESS supports only the H2O.ai AutoML platform [29]. During the second release the support for two additional training engines have been added, YOLOv3 in Pytorch [30] and Keras (with Tensorflow as backend) [31]. The selection of the new engines has been dictated by the requirements of the use cases (see Section 3 and Section 4).

When a request for an AI/ML service arrives at the IESS Manager, the latter selects the most appropriate engine to serve the request. The selection is based on contextual information included in the request and describing the AI/ML problem to be solved. Anyway, if more advanced control is needed, the selection can also be overridden. For example, for a classification or regression problem on tabular data, the IESS Manager selects the H2O.ai framework that provides an automated procedure to build an optimized model. The model is a combination of many models trained by the AutoML engine and combined together to give the best results in the resolution of the problem. On the other hand, YOLOv3 is used for problems of object recognition on datasets of images, while Keras is left for the resolution of other problems, like time series forecasting. In the latter case, the model definition should be provided to the IESS by onboarding it in the IESS catalogue. The IESS takes care of packaging the model into a training runtime container (the training component mentioned in the workflow of Section 2.2.3.2), deploy it on the BASS, run it, and collect the results.

The IESS does not manage any resources on its own. The deployment and execution of the engines is demanded to the BASS, since the latter provides powerful orchestration features that can cover all of the IESS requirements. In a sense, the BASS and IESS are peers in the DEEP platform since one uses the services of the other and vice-versa. During the second release this interaction have been greatly improved and the IESS can request the deployment of training engines with advanced configurations.

For example, the H2O.ai training engine is deployed as a distributed cluster and, depending on the number of resources available, the cardinality of nodes can be increased or decreased. On the other side, the training of the YOLOv3 model works better on GPU and the IESS can request the BASS to deploy the engine on nodes equipped with GPU.

2.2.3.4. IESS Catalogue

The IESS catalogue is dedicated to persistently store data and artifacts related to the services offered by the IESS. The type of the data elements managed by the catalogue is very heterogeneous and includes:

- Supported AI frameworks and ML Engines together with their characteristics, features, metadata and implementation.
- AI/ML pre-trained models, both in binary or serialized format in order to be reused without the need for retraining.
- Runtime environment artifacts for supporting and enabling the execution of model training and model serving (inference).

Given the different characteristics of the data elements to store in the IESS catalogue, we have built a storage stack composed of several technologies, each one dedicated to a specific type of data. MongoDB [32], a document-based database, stores metadata and pointers to frameworks and engines, pre-trained models, and runtime environments. It is used as a main knowledge base by the IESS Manager and as an index to retrieve other data elements. Minio [33], an object storage server compatible with Amazon S3 API [34], is dedicated to store binary artifacts, such as trained models, build files for runtime environments. Finally, a Docker Registry [35] is dedicated to the storage and distribution of runtime environments in the form of container images. The IESS Manager is capable of building container images, push them to the registry, and instruct the BASS to retrieve them in order to deploy new components for both training and inference serving.

2.2.3.5. Packaging and Deploying Inference Apps

For packaging the application BentoML [36] is used, BentoML supports a lot of different ML framework and models, for example H2O.ai specific models. Based on the input and the model type BentoML autogenerates a docker image with a REST API that will provide inference or predictions results on response.

For each supported model in the IESS, a template docker image is created and made available to the IESS. The templates are built by using BentoML as a dependency in a custom Python script.

Because of the difficulty of building a new Docker image or OCI compliant image without using the Docker daemon, which limits the environments where the IESS can be deployed, there are also template images for the inference applications, built based on the original BentoML generated inference image template.

Once the model is trained and retrieved in the IESS, the inference template image is downloaded, and the trained model artifacts are added as a new layer in the image using Jib [37] in a docker daemon-less way. The new image is then uploaded to the catalogue and made it available to the BASS.

3. 5G-DIVE Solution for I4.0 Use Cases

This section provides the refined and final key modules design for I4.0. This will include updates and refinements on the modules already introduced in D2.1 [1], as well as the addition of new modules in Use Case 1 Digital Twin, Use Case 2 Zero Defect Manufacturing, as well as Use Case 3 Massive MTC. Details on Use Case 1 will be described in Section 3.1. Details on Use Case 2 will be described in Section 3.2. Details on Use Case 3 will be described in Section 0. Finally, but yet importantly, the mapping of the three use cases to the DEEP platform will be presented in the end of the respective subsections.

3.1. I4.0 Use Case 1: Digital Twin

The Digital Twin, widely presented in D2.1 [1], is one of the key I4.0 use cases, which consists of a unified system mapping the physical world of an industrial machinery into a virtual world. In the scope of the 5G-DIVE project, we are focusing on robotic systems, namely on a robotic arm manipulator.

This section is structured as follows. First, in Section 3.1.1 we present a refined and final version of the Digital Twin system design, including the updates on its key modules design. Second, in Section 3.1.2 we detail the main workflows of Digital Twin operation as well as its integration with the DEEP platform.

3.1.1. Key Module Design

According to our system architecture design, the EFS for this use case is composed of three parts: i) edge servers; ii) robotic arm; and iii) remote operator user equipment. Thus, the modules comprising the Digital Twin system are distributed over this infrastructure composed by Edge and Fog resources. Notwithstanding, the IESS modules implementing more computational demanding AI/ML tasks (e.g., training of the AI/ML models) that, ideally, could be also further spread to Cloud servers leveraging on Training-as-a-service platforms, wherever the computational power of the edge is believed to be insufficient.

This section provides the refined and final Digital Twin service design (Figure 3-1). In Section 3.1.1.1 we first provide the updates and interactions of the base modules already introduced in D2.1 [1]. In Section 3.1.1.2 we then provide the design of the intelligent modules such as Replay, Movement Prediction, Obstacle avoidance and SLA enforcement that aim to provide enhancements for the Digital Twin system.

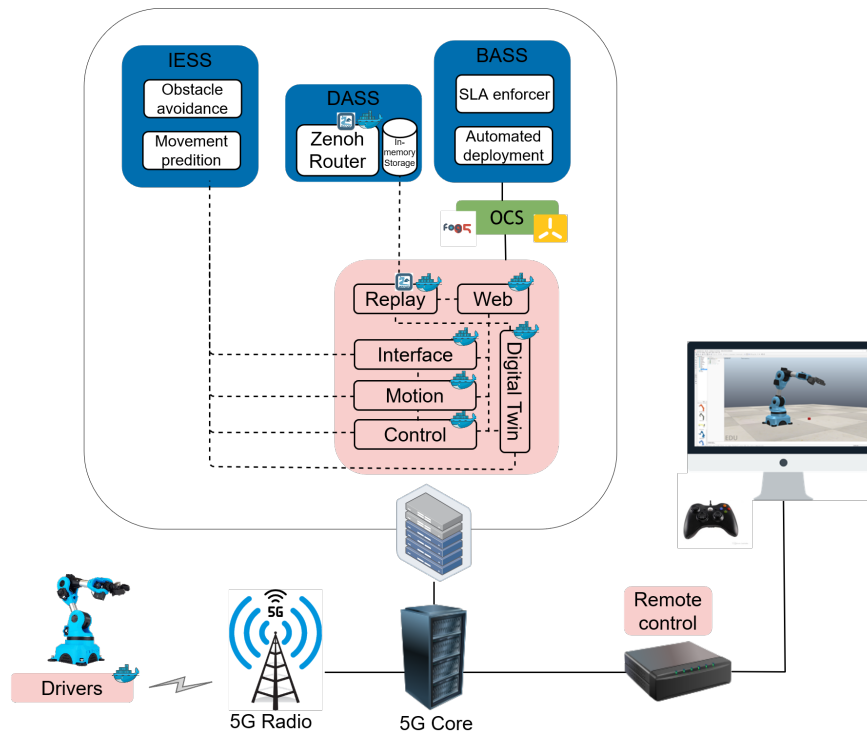


FIGURE 3-1: SYSTEM BLOCK DIAGRAM FOR DIGITAL TWIN

3.1.1.1. Base Digital Twin modules

Figure 3-2 shows the Base Digital Twin modules interactions that were introduced in D2.1 [1]. Accordingly, each of the modules has the following functionality:

- **Drivers:** directly interact with the physical object hardware and are responsible for: (i) making available sensor data and operational states to the other layers, and (ii) executing instructions or navigation commands received from the Control layer. The drivers are available for both the physical (PHY) or simulated (SIM) robotic arms.
- **Control:** is defined as an abstraction layer that allows physical object manipulation. It receives a navigation command or instructions and runs them in a control loop towards the Drivers. The loop is then closed by the physical object continuously sending-back the current state.
- **Motion Planning:** is responsible for finding inverse kinematics and building a path for the robot. The path created consists of a series of navigation commands sent to the control layer.
- **Interface:** is the User Interface module, it enables the interaction with the Digital Twin user features.
- **Digital Twin Application:** implements 3D models and control mechanisms to visualize the variations of the physical object while the control mechanism enables remote control and maintenance.
- **Remote Controller:** in the first release, the *Remote Controller* was part of the Digital Twin application module, allowing the robot to be controlled using the graphical interface located in the Edge server. However, the need to plug the remote controller hardware (e.g., joystick) to the machine running the *Digital Twin Application* make this non-feasible in a real scenario. Thus, the

Remote Controller is now decoupled from the *Digital Twin Application* made available as a standalone module to be deployed preferentially in the operator user equipment.

- **Web Interface:** implements a high-level abstraction for the core controlling functionalities, such as moving any of the joints of the robotic arm, calibration or failure debugging features. Along with the *Remote Controller*, it is another interfacing option between the operator and the physical robot.

Figure 3-2 depicts the interactions between the different modules of the Edge Robotics Digital Twin service. When a user needs to remotely control a robotic arm, it issues a **move joints** (step 1) manipulation command using the *Remote Control* or *Web Interface* module. The **move joints** command is sent to the *Interface* module which offers a custom-made interface (e.g., Python or REST API), translating it in a robot specific movement command. Then, the *Interface* module sends the **movement command** (step 2) to the *Motion Planning* module. When the **movement command** is received, the *Motion Planning* module performs several command validations and generates the **trajectory path** consisting of an array of **position commands**. For each **position command**, each joint is given a specific position, velocity, and acceleration. Once the *Control* module receives the **trajectory path** (step 3), it runs a control-loop against the *Robot Drivers* module. The control-loop starts with the *Robot Drivers* sending the **joint states** (step 4) of the robotic arm to the *Control* module. This information is also propagated to the *Digital Twin* in order to update the virtual model. Next, the *Control* module interpolates the received **trajectory path** to get the next **position command**. The control-loop is then closed when the *Control* module sends the **position command** (step 5) to the *Robot Drivers*. Note that the *Remote Control* and *Web interface* modules can also be configured to send actions to the robotic arm directly via the *Motion planning* or *Control* modules (steps 2 and 3).

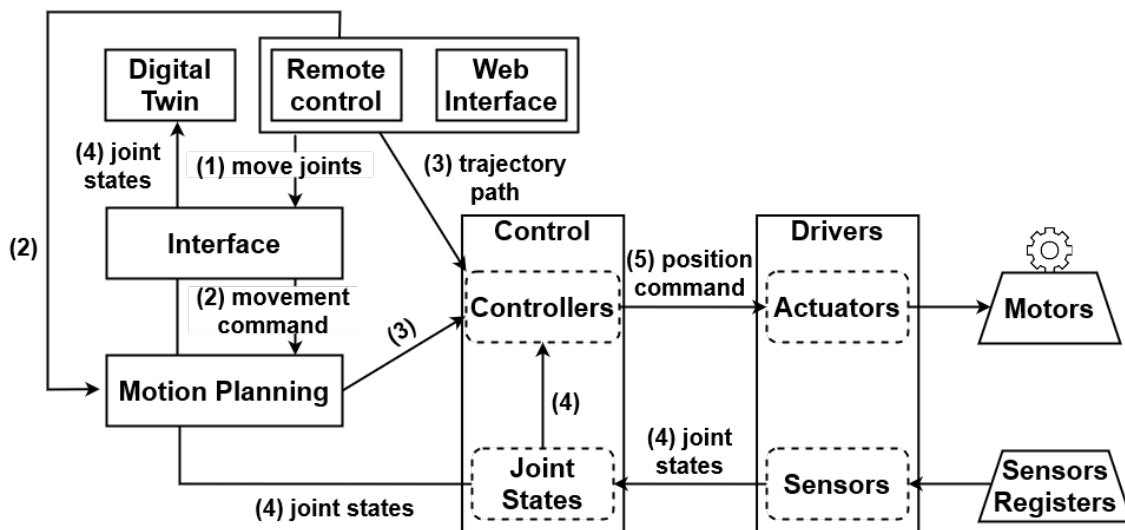


FIGURE 3-2: BASE DIGITAL TWIN SYSTEM MODULE INTERACTIONS

3.1.1.2. Intelligent Digital Twin modules

To enhance the Base Digital Twin system, 4 new intelligent modules, namely Replay, Movement Prediction, Obstacle avoidance and SLA enforcer are introduced, and their design is described in the following sections.

1. Replay Module Design

By the *Replay* feature, we refer to a digital twin replica that re-plays the movements performed by the physical robotic arm, during a given time interval (e.g., look-back on the last 30s) and at a specified movement speed. This feature is useful for failure analysis and debugging in I4.0 environments, allowing an operator to carefully review the past robot movement that led to a malfunction. This feature implements a publish/subscribe mechanism for storing and distributing the robot joint state sequences, which can be then queried by the *Replay* module based on time-series and pushed to the Digital Twin app for the replay visualization.

The Replay feature module design is depicted in Figure 3-3. In a real case scenario, the robot is controlled by an operator. The joint states being produced are continuously pushed from the *Master* node to a *Pub/sub* module, that stores their timeseries in a database. Using the *Web interface* of the Digital Twin service, the operator can trigger the *Replay* module to fetch the most recent sequence of joint states, which is published to the topic of a digital replica in charge of replaying the movements in the *Digital Twin app*.

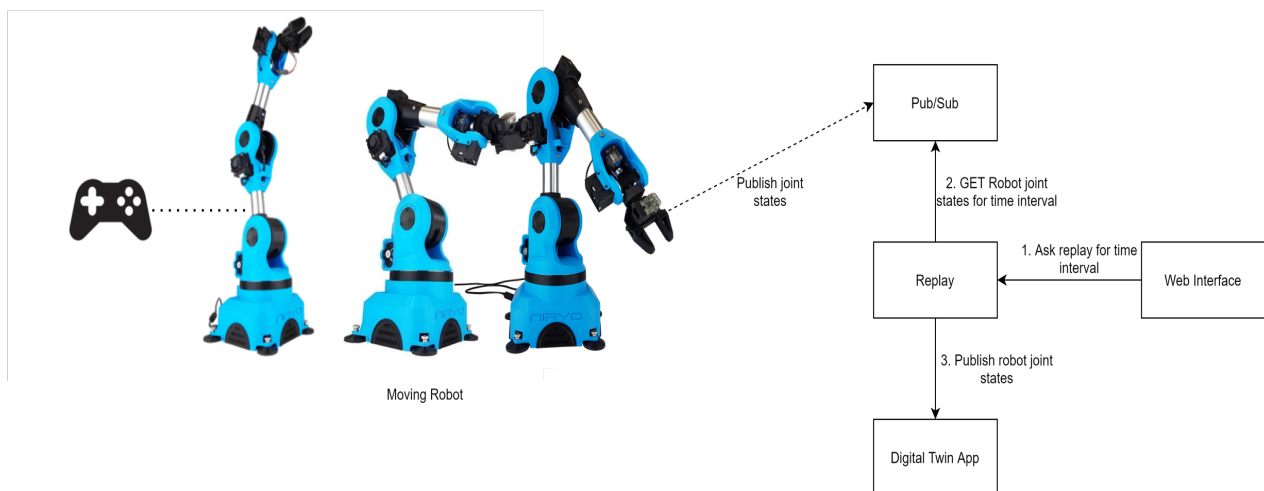


FIGURE 3-3 REPLAY FEATURE MODULE DESIGN

2. Obstacle Avoidance Module Design

The *Obstacle Avoidance* module enables the robotic arm to learn on how to move from an initial position to a target destination, avoiding an obstacle potentially impeding its movements. The solution leverages on Reinforcement Learning (RL)-based algorithms. Traditionally, the moving of objects is either performed by a human operator controlling the robot coherently or by pre-calculating the trajectory of the robotic arm. Automating this task in presence of obstacles require some trajectory planning, which is not possible in dynamic industrial environments with frequently changing obstacles and target locations. An example for a typical scenario is a logistics robot, transporting things from one

destination to another in an ever-changing environment full of misplaced objects. In this sense, this feature represents a fundamental step for the development of more complex automated *pick-and-place* tasks performed by robotic arms in real industrial environments.

The trajectory planning can be inferred using Q-Learning, similarly to what has been frequently used for mobile robots.

In concrete, the implementation design of the feature consists in the following steps:

1. The obstacle is identified whether it is in the real world (then a camera is needed to calculate its relative position and its dimension) or it is virtually generated in the Digital Twin simulated environment.
2. The robot's field of action is discretized into a set of possible states, i.e., the positions that the robot manipulator, our RL agent, can cover in a discretized 3D-space, reproduced in a Python environment. The discretization step affects the training time and, as it can be easily imagined, the granularity of the trajectory. The agent is allowed to move from one state to another, according to a *policy* composed of a finite set of actions ("up", "down", "left", "right", "up-left", etc.). A source state and a destination state are chosen and a random obstacle, a parallelepiped, which can represent the bounding box of any physical object, is placed in between.
3. Q-Learning is run in the IESS module. After a given number of *episodes* where the robot/agent keeps trying to move from the initial state to the destination without hitting the obstacle following the policy (otherwise the episode ends and the agent must restart from the source position), the trajectory is output.
4. The inferred trajectory is mapped to the real or simulated environment coordinates and translated into a joint state sequence using a module which calculates the inverse kinematics. The robot is fed with the sequence, so that it can perform the *pick-and-place* task accordingly.

The validity of the procedure listed above is currently under investigation. Traditional Q-Learning is not suitable to be used in dynamic environments where the obstacles may continuously change, as the training happens online and strictly depends on the environment. Another option would be to train a Deep Q-Learning Network (DQN), with several environment permutations: different positions of source, destination and obstacle, multiple obstacles with variable shapes, etc. By this, the training of the neural network could be performed offline, and the robot would be able to re-plan its trajectory on-the-fly, whenever a change in the environment is detected by a camera in physical world or the action field of the robot simulator is re-arranged.

3. SLA Enforcer Module Design

The *SLA Enforcer* implements AI/ML or heuristic-based mechanisms to guarantee that the SLA requirements for the Digital Twin service are met. The feature monitors a set of meaningful indicators (SLI) of the service. For example, the application latency time between the issue of a command and its execution in the physical robot is a key metric for a real-time remotely control of physical robot through its digital twin replica. If these SLIs do not meet an agreed objective (SLO), e.g. to keep a certain metric below a certain threshold, then the SLA is violated, leading to costly business implications. To prevent

this from happening, the SLA enforcer mechanism optimizes the resource utilization by performing e.g., resource scaling or service migration.

This module (see Figure 3-4) is as well split into a set of submodules: the *Monitoring Probe* submodule reads the SLI data from the *Digital Twin Application*, extracting, for example the times of the:

- Actuation: amount of time required for the robot arm to begin movement after a new command is received
- Synchronization: accuracy of synchronization between digital twin and physical robot
- Automated job execution: amount of time required for a robot to complete an automated job (e.g. *pick-and-place*)

and the network parameters on which the actions listed above depend consistently:

- Latency
- Bandwidth
- Jitter
- Packet loss

This data is pushed to a *Pub/Sub* module which relays it to the *SLA Enforcer*, where the AI/ML algorithm training takes place and the model is continuously updated with up-to-date data. The model parameters and the SLA parameters are passed to another submodule which is in charge of detecting the violations of the SLO. If the SLO threshold is crossed, the algorithm running in this module will predict the optimal hardware (CPU, memory, etc.) configuration that the infrastructure must adopt to prevent the violation of the SLA and then trigger the scaling of resources.

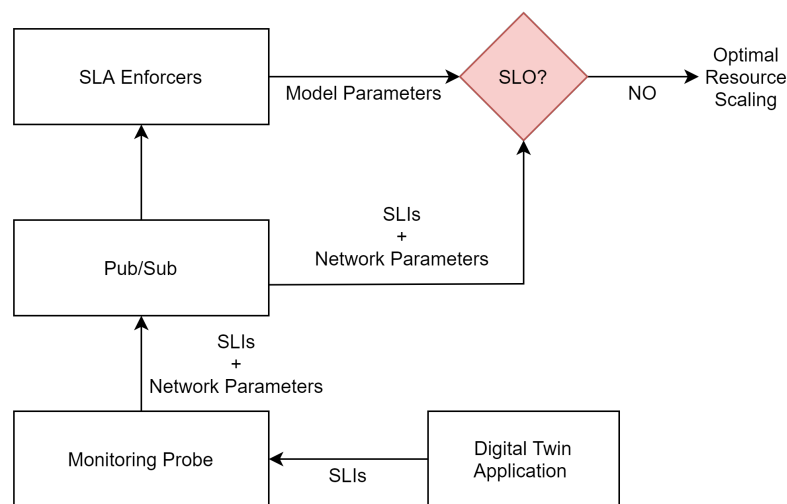


FIGURE 3-4 SLA ENFORCER MODULE DESIGN

4. Movement Prediction Module Design

By being continuously fed with the command history required to execute a given task, the *Movement Prediction* creates AI/ML-based models for movement prediction. In doing so, it can infer the next movement command upon disrupted connectivity between the physical system and digital replica, triggering its execution in order to guarantee an uninterrupted flow of commands. Finally, a feedback

loop between the robotic arm and this module is established so that the AI/ML-based model is iteratively refined to reach optimality.

More in detail, the *Movement Prediction* module is designed to support the plug and play of any kind of AI/ML algorithms, such as VAR, LSTM, TCN or GRU, given that they get as input a list containing the historic of commands and produce as the output the predicted next command(s) to be executed. In this sense, depending on the number of commands predicted by the AI/ML algorithm, the *Movement Prediction* module is flexible to adapt the rate at which predictions are requested. As part of the *Movement Prediction* module, it supports the training of AI/ML models using the historic of commands to execute one or more tasks. However, already trained AI/ML can be given to the *Movement Prediction* module, being used solely for inference tasks.

Different options for the integration of the *Movement Prediction* module were studied, as depicted in Figure 3-5. After assessing the pros and cons, Option 1 was selected due to its less disruptive approach, allowing not only a seamless integration with the Digital Twin service but also the possibility to enable or disable this feature without impacting the vanilla Digital Twin service. Moreover, it fully relies on the ROS communication capabilities, already exploited by the vanilla Digital Twin service.

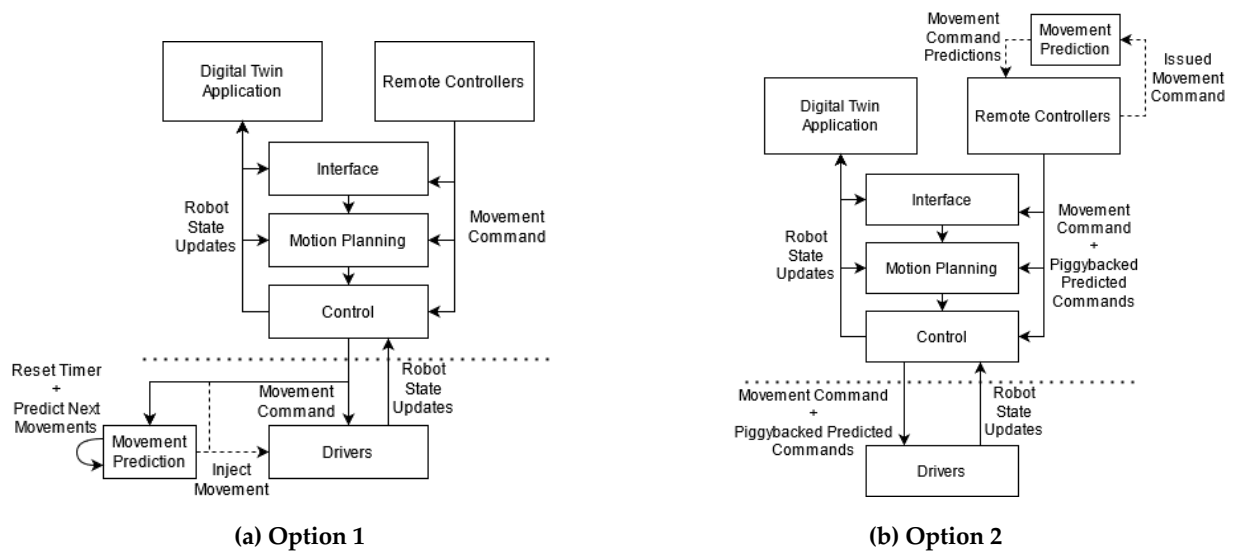


FIGURE 3-5: MOVEMENT PREDICTION MODULE INTEGRATION OPTIONS

A step-by-step description of Option 1 is described as follows:

1. Movement instructions are issued by the Digital Twin controlling interfaces (e.g., *Remote Controller* or *Web Interface* modules), which are forwarded across the robotic stack (i.e., *Interface*, *Motion Planning*, *Control* modules).
2. When the *Control* module sends the movement instructions to the *Drivers*, so that they are executed in the robotic arm, the *Movement Prediction* module is able to intercept those same messages. To do so, the *Movement Prediction* module is subscribed to the same ROS topics as the *Drivers*, making the whole process transparent.
3. The *Movement Prediction* module stores the received movement commands in a circular buffer, so that only the required *look-back* commands are stored (i.e., history of commands used for computing the predictions).
4. Whenever the *Movement Prediction* module receives a movement command, it restarts an internal timer. An eventual timeout means that a command is lost and, therefore, a prediction must be computed.
 - a. The computed movement prediction is sent (i.e., injected) to the *Drivers*, being executed in the robotic arm. Even though this command was issued by the *Movement Prediction*, it is fed back with it due to the way ROS communications are handled.
5. If the real movement instruction is delayed, it is discarded by the *Drivers* since its lifespan will be already expired.

Using the previous procedure, predicted movements are only issued if a missing command is detected. Therefore, the *Movement Prediction* module does not impact or interfere with the normal operation of the Digital Twin service, if the latency requirements are met.

Apart from the *robot drivers*, which are intended to run directly on the robotic arm, and from the Digital Twin application, which can be either edge functionality or an application running directly in the operator user equipment, the remaining modules can be deployed on both fog devices and on edge servers.

In the preliminary integration stages, we found out that distributing part of the modules to the Edge server, as envisioned here, did not degraded the performance of the Digital Twin system in terms of accuracy, reliability, and bandwidth utilization, with respect to a localized robot-based solution. Moreover, the offloading of many of these modules represented savings in terms of the robot computational and power resources.

3.1.2. Mapping to the DEEP Platform

This section presents mapping of Digital Twin use case to the DEEP platform, being presented several workflows of its integration with the BASS, DASS and IESS. Section 3.1.2.1 presents the workflow for an end-to-end deployment of the Digital Twin solution over a distributed EFS using the BASS. Section 3.1.2.2 depicts the workflow for the *DASS-enabled Replay* feature. Lastly, Sections 3.1.2.3, 3.1.2.4 and 3.1.2.5 present the workflow for the *Obstacle Avoidance*, *SLA Enforcer* and *Movement Prediction* intelligence engines provided by the IESS.

3.1.2.1. End-to-End Digital Twin Service Instantiation

Figure 3-6 shows the mapping of the designed Digital Twin use case with the BASS for end-to-end deployments of the Digital Twin service. In the Digital Twin use case, the BASS is used to ease the service creation, deployment, instantiation, and management tasks over the EFS infrastructure on behalf of the Remote Operator. By making use of BASS GUI described in details in Section 2.2.2.2, the Remote Operator fills in a Vertical Service Blueprint with business-oriented parameters, which is then used to request the Digital Twin Service deployment (step 1). The Vertical Service Descriptor holds information related to the Digital Twin key modules such as type of remote-control mechanism that the operator intends to use (e.g., joystick, web interface, etc), type of robot, type of operation to perform, etc. Next, the BASS translates the business-oriented parameters also included in the descriptor into one or more network service descriptors that holds detailed deployment and management information, such as image location, IP addresses and ports of each module, deployment location (e.g., fog, edge or cloud) and dependencies between the modules. Once the network service descriptors are available, the BASS requests their deployment and instantiation through the Orchestrator Driver (step 2). The OCS receives this request and through the VIM deploys each module of the Digital Twin service in the distributed EFS (step 3). Each node that is involved in the deployment (e.g., robot arm, user device, edge server) reports back to the OCS the status of Digital Twin module (deploying, running, stopped, error, etc) (step 4). This status information is propagated back to the BASS (step 5), being presented to the Remote Operator through the BASS GUI (step 6). When all the modules have the status running, the Remote Operator can start using the Digital Twin service.

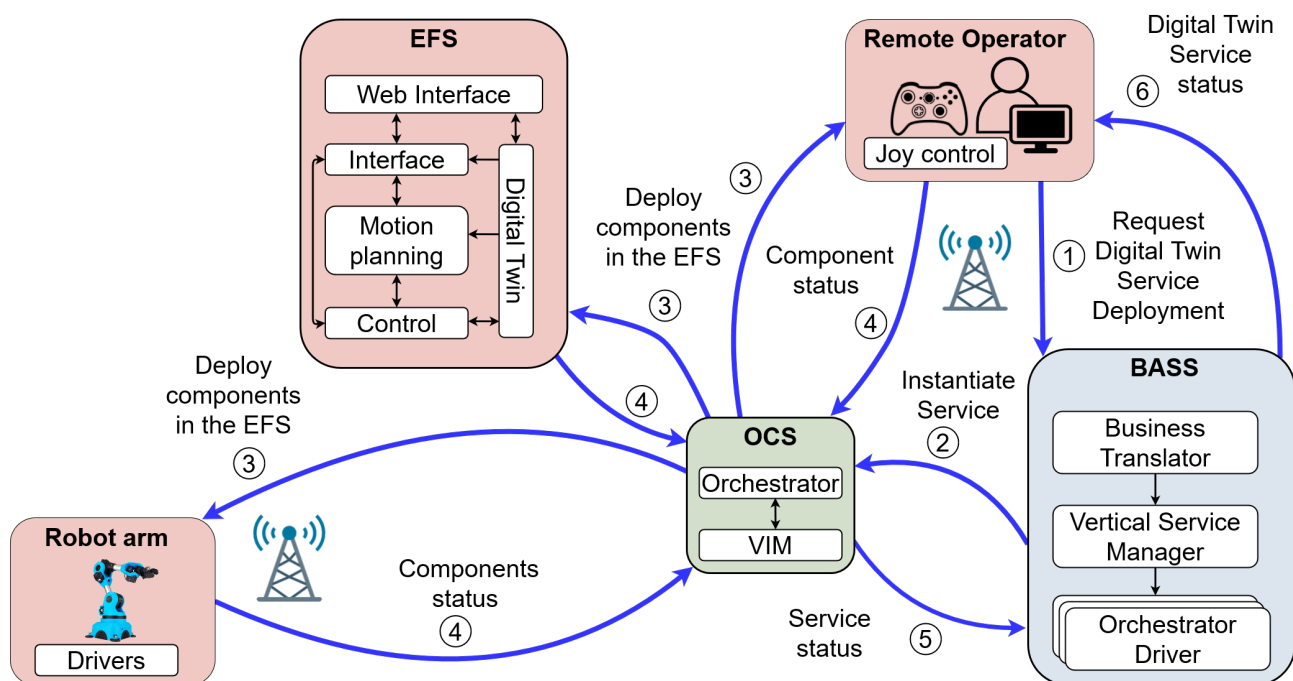


FIGURE 3-6: DIGITAL TWIN END-TO-END DEPLOYMENT WITH BASS

3.1.2.2. DASS-enabled Replay

Figure 3-7 shows the mapping of the Digital Twin *Replay* feature that exploits the functionalities made available by the DASS. The Remote Operator, with the help of the Digital Twin application and a remote-control mechanism (e.g.: joystick , web interface), can remotely control the physical robotic arm (step 1). Consequently, the robot arm in real time updates the Digital Twin to keep a tight synchronization between the physical and digital worlds (step 2). Simultaneously, the *Replay* feature continuously collects in real time the Digital Twin states from the corresponding application using the DASS Data Dispatcher, storing them in the DASS Data Storage (step 3). In a specific moment of the remote operation (e.g., due to robot misbehaviour), the Remote Operator can request for a replay of a past sequence of movements through the Web Interface GUI by specifying the desired time interval (step 4). The *Replay* module queries the DASS Data Storage about the Digital Twin states associated to requested time interval (step 5). Once the past sequence data is obtained (step 6), the *Replay* module starts to playback the data in a loop fashion. The Remote Operator is informed that the action replay is ready (step 7) via the Web Interface GUI and in that moment, he can add a new virtual replica in the Digital Twin application in order to visualise the replay data.

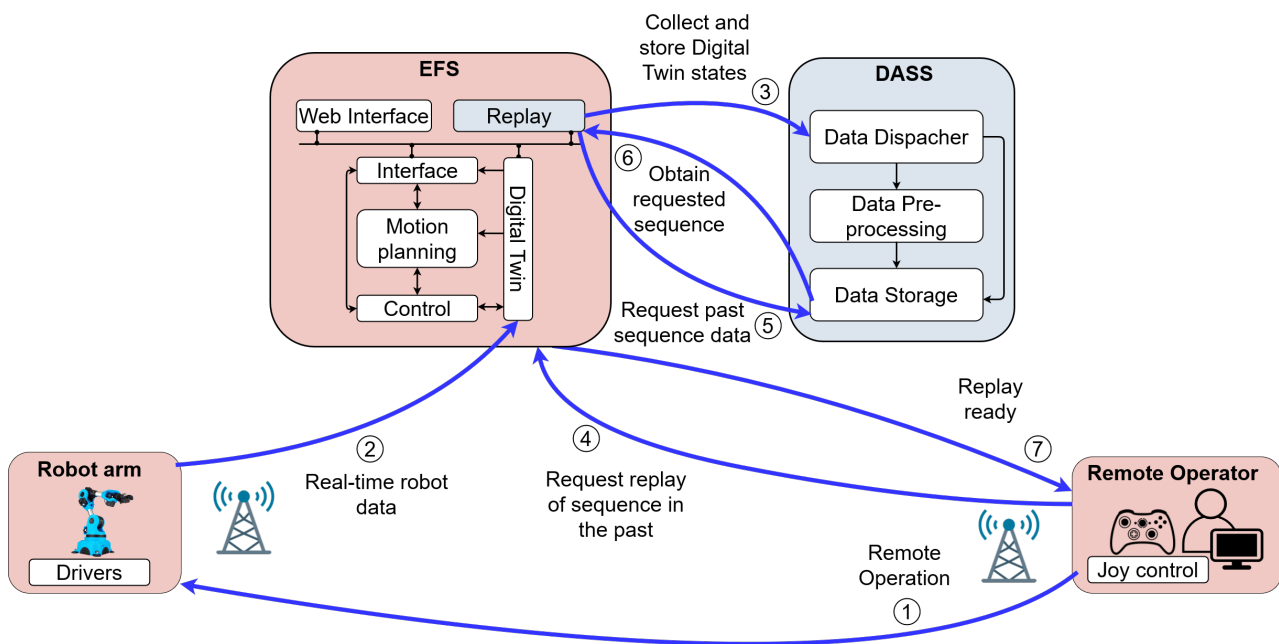


FIGURE 3-7: DIGITAL TWIN DASS-ENABLED REPLAY FEATURE

3.1.2.3. IESS Automation for Obstacle Avoidance

Figure 3-8 shows the mapping of the *Obstacle Avoidance* module with the IESS platform. In Step 1 the *Obstacle Avoidance* communicates to the IESS the positions of the source and the destination, as well as the position and the shape of the obstacle. The RL algorithm is run in the IESS to compute the trajectory the robot has to follow to move an object from A to B without hitting the obstacle. This trajectory path is passed back to the *Obstacle Avoidance* (step 2), that computes the inverse kinematics of the coordinates, yielding the corresponding sequence of robot joint states. The corresponding commands

are published to a topic of the module of the robot stack (*Interface*, *Motion* or *Control*, step 3). Eventually, the commands are executed by the *Drivers* (step 4) of the robot.

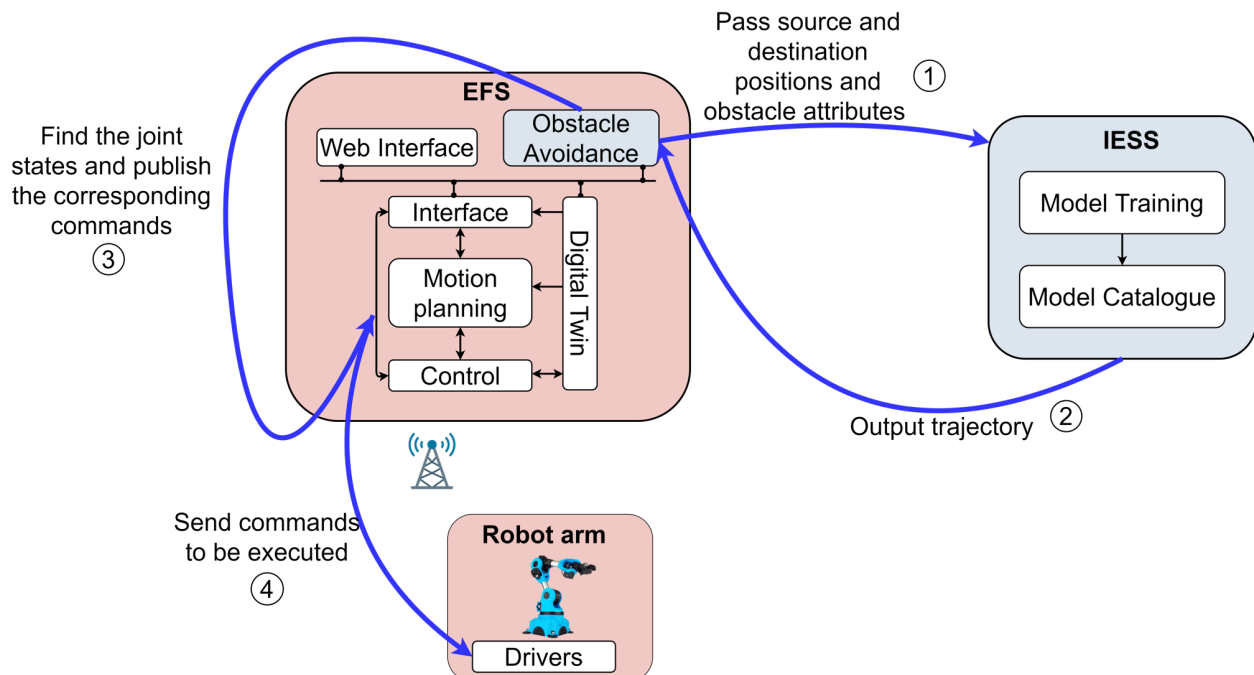


FIGURE 3-8 DIGITAL TWIN IESS OBSTACLE AVOIDANCE

3.1.2.4. DEEP Integration for SLA Enforcer

Figure 3-9 shows the mapping of the *SLA Enforcer* module with the IESS, the DASS and the BASS. The *Monitoring Probe* module continuously extracts the SLIs values (actuation, synchronization times, etc.) from the *Digital Twin Application* and monitors the network parameters (latency, bandwidth, jitter, packet loss), pushing this data to the DASS (step 1). The DASS relays the data to 1) the BASS, so that it can check if the thresholds associated to the SLOs are met, 2) to the IESS so that the AI/ML model can be updated (step 2, 3). The *SLA Enforcer* in the BASS continuously predict the optimal configuration. When the BASS detects a violation, it triggers the OCS Orchestrator to scale resources. This translates into a change in the Network Service Descriptor (NSD) file for adapting the resource allocation. The BASS pushes the new configuration file in the Orchestrator that can enact the scaling of the EFS virtual hardware (step 4). Eventually, the VIM executes the new configuration (step 5). Also, the BASS checks if the SLIs value get back below a lower-threshold, to enact the same mechanism but this time to scale resources down, so that the initial EFS hardware configuration can be restored back.

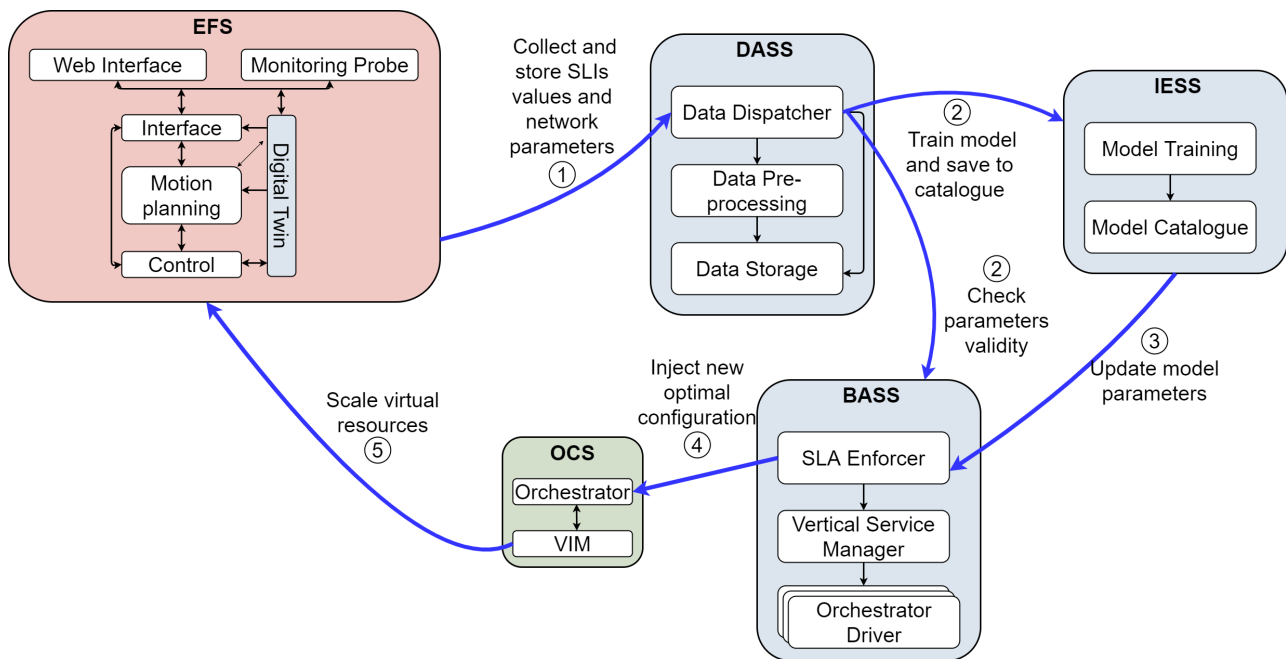


FIGURE 3-9 SLA ENFORCER E2E MECHANISM

3.1.2.5. IESS Automation for Movement Prediction

Figure 3-10 shows the mapping of the *Movement Prediction* module with the IESS. The Remote Operator remotely controls the physical robotic arm (step 1). Consequently, the robotic arm updates the Digital Twin application in real time (step 2). Simultaneously, the Digital Twin application uses the DASS Data Dispatcher and Data Storage to collect and store the joint states of the virtual replica (step 3). The stored robot data is continuously used by the IESS Model Training to train a movement prediction AI/ML model (step 4). When this AI/ML model is trained and after passing all the cross-validation tests, the IESS stores the model in its catalogue (step 4) and sends it back to the BASS to be loaded and included in the Digital Twin service (step 5). The BASS interacts with the OCS through the Orchestration Driver, requesting the instantiation of the Movement Prediction module (step 6). The OCS performs the robot on-device deployment (step 7) with the status of the operation sent back to the OCS (step 8). Finally, the BASS GUI presents the status of the deployment (step 9 and 10), informing the Remote Operator that the *Movement Prediction* module has been added to the Digital Twin service.

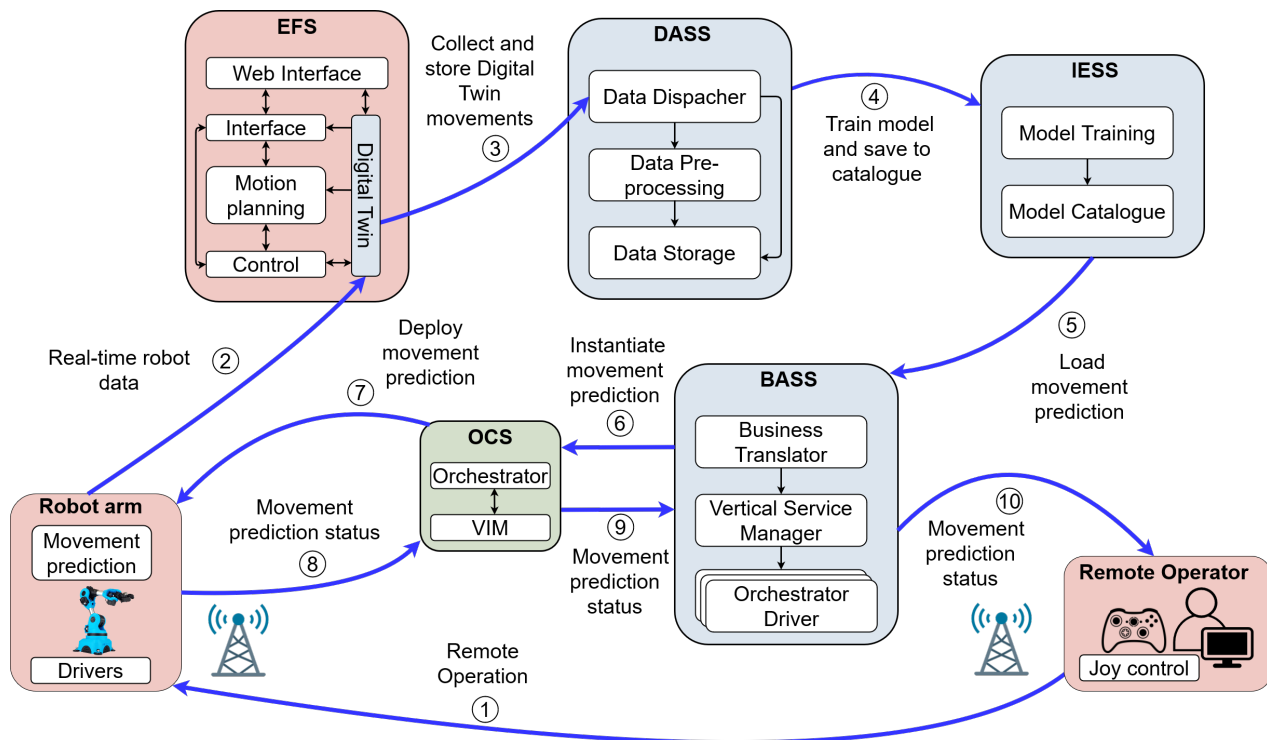


FIGURE 3-10: DIGITAL TWIN IESS MOVEMENT PREDICTION

3.2. 14.0 Use Case 2: Zero Defect Manufacturing

The Zero Defect Manufacturing (ZDM) used case has been presented in D2.1 [1]. The goal of the use case is to attain automatic remote control of a factory. The factory is producing goods that are being monitored as they leave the production line for possible defects.

This section presents the most recent developments in the ZDM use case. First, we present the updates on the use case and its key modules design. Second, we detail the main workflows of ZDM operation as well as its integration with the DEEP platform.

3.2.1. Key Module Design

This section focuses on describing the latest evolution for the Zero Defect Manufacturing (ZDM) use case, which relates to a new defect detection engine and the setup adaptation to use with a new Edge node, the AWS Wavelength.

3.2.1.1. Defect Detection Engine

Defect detection is becoming an increasingly important task during a manufacturing process. The early detection of faults or defects and the removal of the elements that may produce them are essential to improve product quality and reduce the economic impact caused by discarding defective products. This point is especially important in the case of products that are very expensive to produce [38]. In order to simulate a more realistic factory environment, a new object detection engine has been trained using YOLOv3 [39]. In the newly trained engine, circular black marks were chosen as defects for the objects. Figure 3-11 shows the cubes that are placed in the production line. The mono-coloured cube is considered as a non defective object. The cubes with circular black marks are considered defective objects.

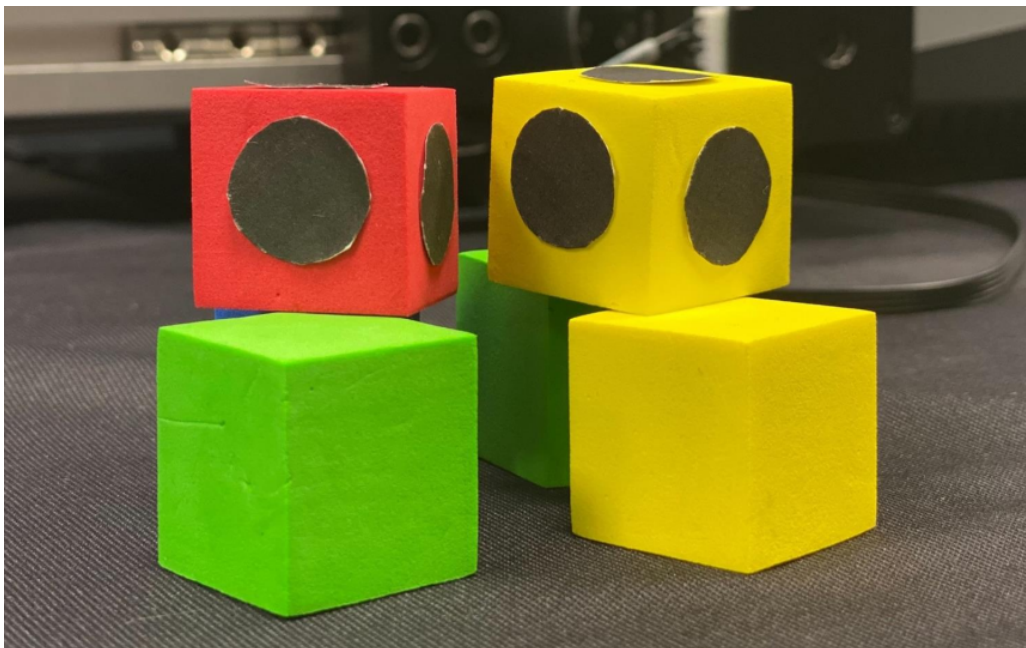


FIGURE 3-11 DEFECTIVE AND NON-DEFECTIVE CUBES AS A PRODUCT OF THE FACTORY

3.2.1.2. AWS Wavelength

AWS Wavelength is an AWS Infrastructure offering optimized for mobile edge computing applications. Wavelength Zones are AWS infrastructure deployments that embed AWS compute and storage services within a mobile network operator's datacenters at the edge of the 5G network. By deploying computing resources co-located with a Mobile Network Operator (MNO)'s Core Network, application traffic from 5G devices can reach application servers running in Wavelength Zones without leaving the telecommunications network. This eliminates the latency that would result from application traffic having to traverse multiple hops across the Internet to reach their destination, enabling customers to take full advantage of the latency and bandwidth benefits offered by modern 5G networks.

The ZDM setup has been tested with an Edge Node compute capabilities located at a Wavelength zone and the setup is as depicted in Figure 3-12.

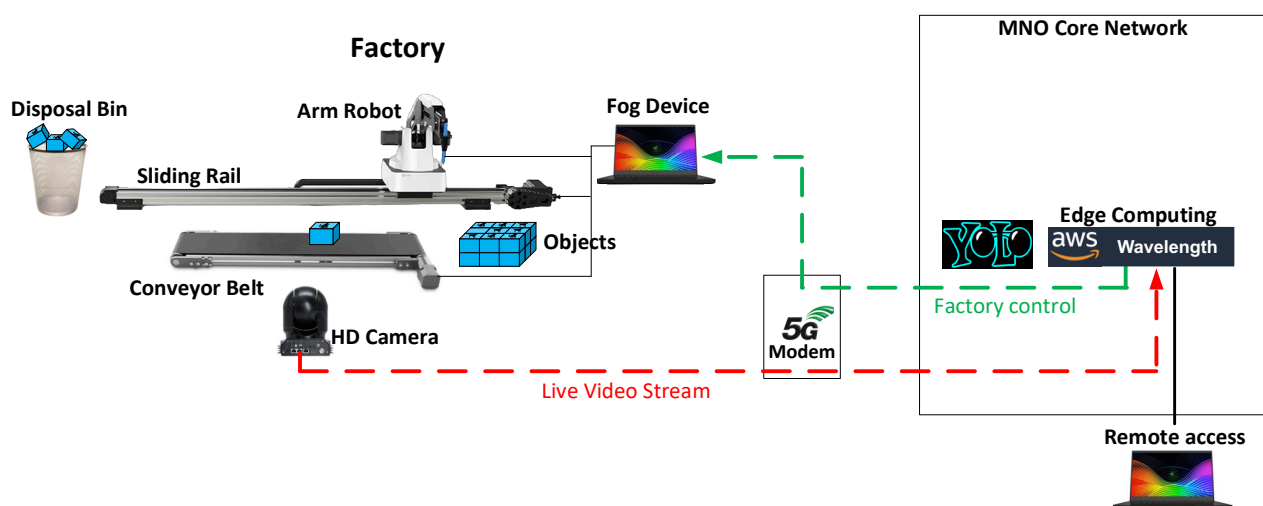


FIGURE 3-12 ZDM SETUP WITH AWS WAVELENGTH

The presented setup is similar to what has been presented in D2.1 [1] for the ZDM use case, with the difference that the Edge node has been replaced by computing resources located in the AWS Wavelength zone, that is located inside a MNO's domain. The new object detection engine has been trained using YOLOv3 and deployed within these computing resources.

The newly trained defect detection engine and its deployment at AWS Wavelength complete the latest ZDM use case setup that has been integrated with the DEEP Platform components. The next subsections describe the workflows of this setup with the DEEP Platform components.

3.2.2. Mapping to the DEEP Platform

In this section, a mapping of the Zero Defect Manufacturing use case to the DEEP platform is presented, along with the workflows of its integration with the BASS and DASS.

3.2.2.1. BASS Service Instantiation for ZDM

Figure 3-13 shows the mapping of the BASS component in the ZDM use case, with the workflow steps for initiating the service and checking its status during the execution.

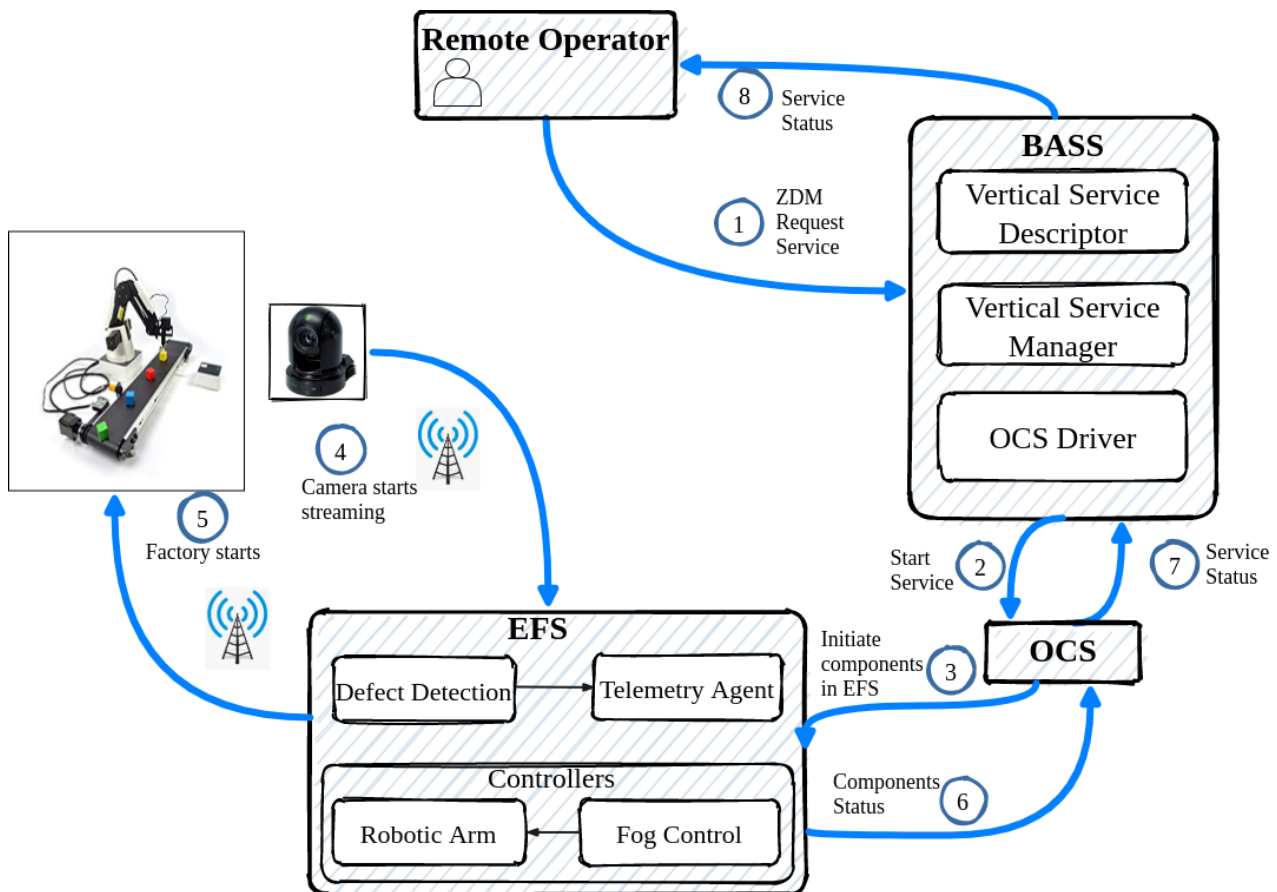


FIGURE 3-13 BASS SERVICE INSTANTIATION

The service initiation process is started by the operator (step 1), which can fill the parameters of the Vertical Service Blueprint in the GUI console of the BASS. The Vertical Service Descriptor holds configuration parameters of the service which are specific to the components needed for running the application. After the vertical service descriptor is defined the BASS uses the orchestrator driver (step 2) to command the orchestrator to start the components in the EFS (step 3). These components are the defect detection application, the fog device control software, the driver controlling the robotic arm, and the telemetry agent. Once the components are deployed the camera can start streaming (step 4) and the factory service can start (step 5). The EFS deployed components report their status to the orchestrator (step 6) which updates the service status in the BASS (step 7). This way the service cycle is complete and the operator (step 8) monitors the result of the vertical started.

3.2.2.2. DASS Enabled Telemetry Data Collection

In the ZDM use case, various forms of telemetry data are collected, as reported in D2.1 [1]. Fig. 3-13 shows the mapping of the BASS component in the ZDM use case, with the workflow steps for initiating the data collection, posting, getting and subscribing for the collected telemetry data.

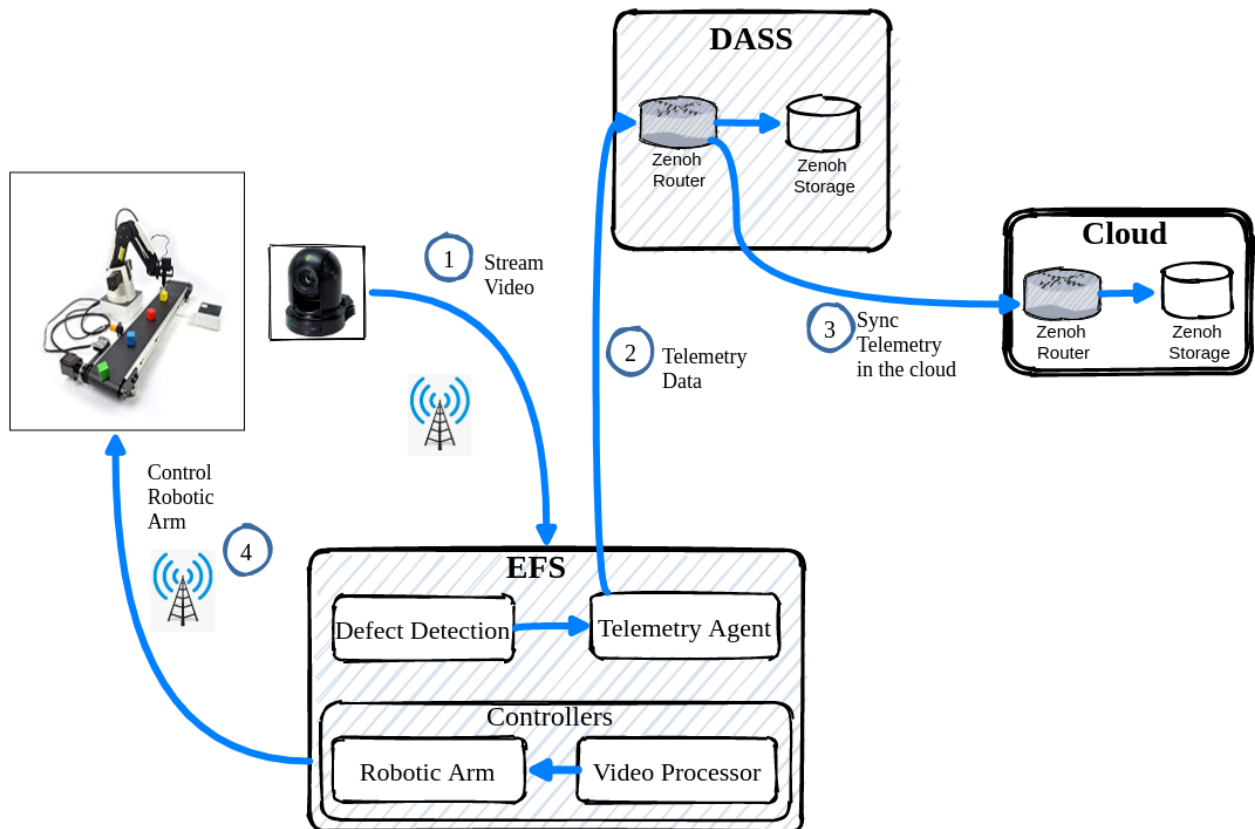


FIGURE 3-14 DASS-ENABLED TELEMETRY DATA COLLECTION

The service flows for telemetry data collection start in step 1, when the factory for the ZDM use case starts working. The camera starts streaming the video towards the Edge node and the factory starts working, with the cubes being placed on the running conveyor belt. The telemetry agent in the EFS uses the Zenoh protocol to communicate with the Zenoh router located at the DASS core (step 2). Next, the Zenoh router within the DASS core synchronizes with the Zenoh router located at a Cloud Services Provider. This synchronization is followed by a link establishment between the routers that uses the Zenoh protocol (step 3) to correctly direct the telemetry data into a Zenoh storage unit, located in the Cloud.

3.3. 14.0 Use Case 3: Massive MTC

In this section, we present the key module design of the current massive MTC system and how it maps to the DEEP platform. In Section 3.3.1, we explore the updated system design with Kubernetes integration and then present the RF fingerprinting module. In Section 3.3.2, we show the mapping of mMTC use case to the DEEP platform by elaborating the workflow of the use case.

3.3.1. Key Module Design

In D2.1 [1] we incorporated cloud native design methodologies to the development of our virtualized IoT network stacks, i.e., LoRa and IEEE 802.15.4. The baseband functions were offloaded from the radio heads to virtualized software functions in the edge. In this section, we first present some updates on the modules developed in D2.1 [1]. Then we elaborate on the current system design with Kubernetes to further explore the orchestration features and to integrate with BASS.

To enhance the mMTC system, we updated several modules designed in first release and also added two new modules. For LoRa emulation testbed, *i)* we updated the packet generator module which was used to emulate cell traffic. The updated packet generator module aims to reduce the complexity for deploying large LoRa networks. In the first release, one packet generator block was developed to emulate one cell. With the new module, we can use one packet generator block to emulate multi-cell traffic by configuring the number of cells and traffic pattern. *ii)* We adopted ZMQ PUB/SUB sockets instead of packet aggregation function for packet reassembling. The system achieves better performance regarding throughput as well as latency. Measurement results show that the maximum supported full-traffic cells for one communication stack increases significantly from 9 to 57. *iii)* We integrated the system to the Kubernetes framework for orchestration and automation of the Docker containers. *iiii)* We also use open-source database and visualization tools, i.e., Telegraf, InfluxDB and Grafana to visualize and record the system metrics such as resource utilization, latency, and throughput. Further, For IEEE 802.15.4 testbed, *i)* an improved RF fingerprinting algorithm was designed with improved performance and scalability. *ii)* A software module for simulating 802.15.4 devices were developed to facilitate system testing and scalability study. In the following, we will focus on the two major updates for the mMTC use case, i.e. Kubernetes integration and intelligent application for RF fingerprinting.

3.3.1.1. System Design with Kubernetes

In D2.1 [1] we explored a vRAN architecture for IoT networks where IoT baseband functions are virtualized in Docker containers. The Docker containers are deployed on a single mini PC with a command line interface (CLI). However, when it comes to running containers in real cloud native networks, service providers can end up with many many containers. These containers need to be deployed, managed, connected, monitored and updated, and this is where a container orchestration tool, e.g., Kubernetes, comes to play an important role. In this section, we will address the concepts of Kubernetes and how it is implemented in mMTC use case.

Kubernetes is an open-source orchestration software that provides an API to control how and where the containers will run [40]. In Kubernetes, application software resides in containers as a service and runs in pods which are the smallest deployable units created by service providers. A desired state of the application/service is described in a Kubernetes Deployment file. The deployment could be scaled vertically and horizontally, and pods could be replicated as configured to provide redundancy. Furthermore, to enable network access to the services, a Kubernetes Service is defined to expose the service with IP addresses and ports. Bringing Kubernetes into our work benefits us in several aspects:

- **Availability:** Instead of deploying IoT communication stacks on one node, Kubernetes uses a multiple cluster consisting multiple nodes for running containerized applications. A cluster contains at least one master node and several worker nodes, providing fault tolerance and high availability. A master node manages the worker nodes and scheduling of the applications. A worker node hosts the pods that run the component of the application workload.
- **Networking:** Kubernetes allows cluster components to communicate with each other (internal) and with other applications outside the cluster (external). There are typically four types of networking for a Kubernetes cluster, i.e., container-to-container communication, pod-to-pod communication, pod-to-service communication, and external-to-service communication. For each type of communication, Kubernetes offers ways to handle the traffic routing automatically.
- **Elasticity/scalability:** With billions of devices connect to the network, the data is created at an unprecedented rate and is also hard to predict. All of these require the system's ability to handle elastic demand and shifting workloads. Kubernetes offers an infrastructure that can scale horizontally/vertically which scales the system resources according to the workload, to meet the end user demand. Furthermore, the scaling of the services can be easily done across network clusters without any impact on the services.

For a better understanding of how Kubernetes manages the containers in our use case, we show in Figure 3-15 the mMTC deployment diagram where a three-node Kubernetes cluster is deployed. Three services, i.e., LoRa decoding function, IEEE 802.15.4 decoding function and RF fingerprinting are running on each worker node. To receive/send data from/to outside the cluster, we expose LoRa and IEEE 802.15.4 services with external IP addresses and ports. For example, in uplink, traffic data from IoT devices is received by the radio head and then published to the worker nodes using ZMQ. Note that we use MetalLB [41] as our load balancer. Kubernetes is able to load balance and distribute the network traffic across the nodes according to the policy applied. In the rest of this section, we present a key part of the Kubernetes configuration, i.e., communication between the radio head and pods as well as how services are connected. Then the scripts for LoRa Kubernetes deployment are provided as an deployment example. Details regarding the integration with the DEEP are addressed in Section 3.3.2.1..

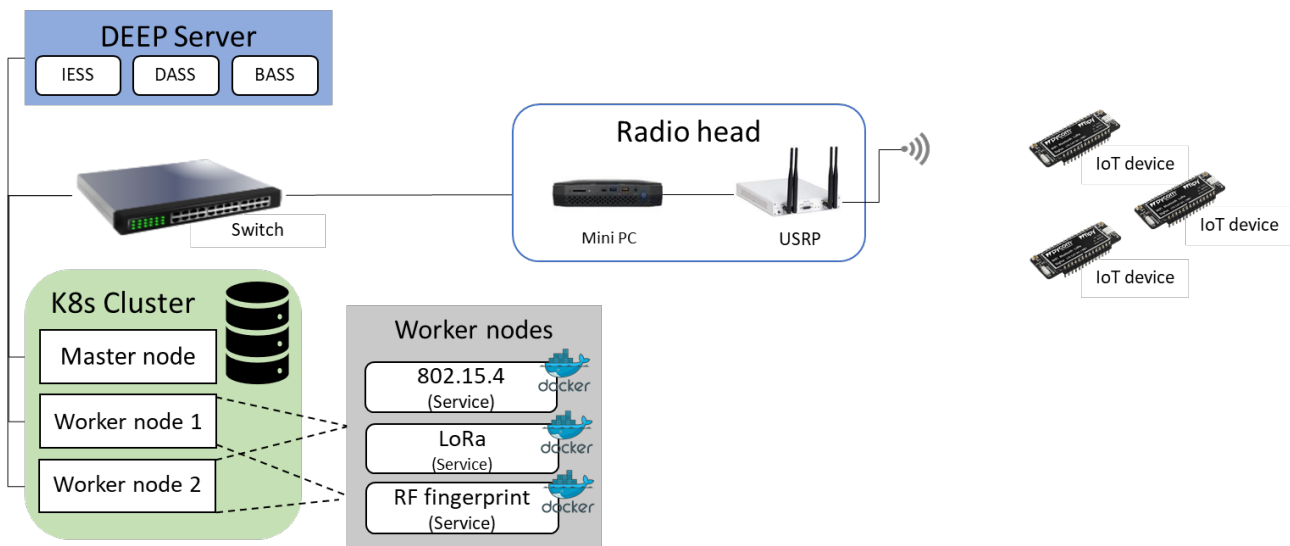


FIGURE 3-15 MMTC DEPLOYMENT DIAGRAM

Normally, Kubernetes pods are created and destroyed to match the state of the cluster. However, each pod gets its own IP address and the set of pods running in one moment in time could be different from the set of pods running that application a moment later. To handle this, we expose the deployment as a Service. In Kubernetes, a Service is an abstraction which defines a logical set of pods and the policy defining the access rules. After the services, i.e. LoRa and 802.15.4, in Kubernetes clusters are exposed, we use two ZMQ PUB/SUB patterns for the connection between the radio head and the Kubernetes services as shown in Figure 3-16. For downlink signals, edge containers (pods) work as publishers and the radio head works as a subscriber. Edge containers (pods) will bind the corresponding service internal IP address and port, while the radio head will connect to the service external IP address and port. A Kubernetes Service is used to map the two sets of IP addresses and ports so that traffic can be load balanced across the pods. For the uplink signal, on the contrary, the radio head works as a publisher and edge containers work as subscribers. In this case, the radio head publishes the received IoT data to the edge by connecting to service external IP address and port. Meanwhile, edge containers (pods) bind to the corresponding internal IP address and port to receive data. We utilize these two patterns in different directions since we want one service can support multiple radio heads (cells). Otherwise, with an increasing number of cells connected to the system, pods in the edge need to connect to hundreds or thousands of IP addresses and ports to receive data from radio heads. The system would be much more complex and difficult to deploy.

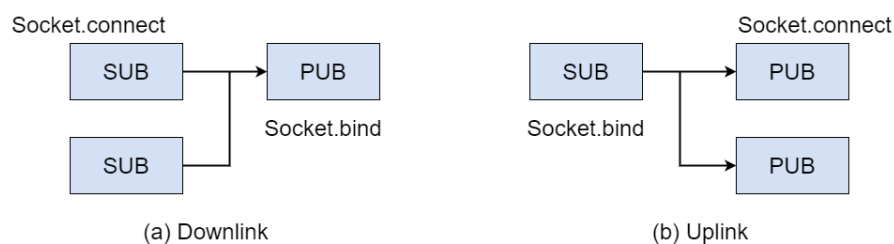


FIGURE 3-16 ZMQ PUB/SUB PATTERNS FOR DOWNLINK AND UPLINK SIGNALS

Further, we take LoRa deployment as an example to show how we expose the service with an external IP address and port. Figure 3-17 illustrates the LoRa service configuration scripts of the *.yaml* file which specifies the configuration of the Kubernetes Service deployment. From the scripts, we observe that we create a new Service object named 'lora-service-edge', which targets TCP port 8088 on any pod with the 'app=lora-edge' label. In addition, Kubernetes assigns this Service an IP address which can be used to communicate with the devices outside the cluster. It is worth mentioning that we herein expose the Service externally using MetalLB, which is a load-balancer implementation using standard routing protocols [41]. Once the Service is deployed, we can publish data to the edge by connecting to the external IP address and the exposed port. The data then will be distributed to the specific pods using predefined load balancing policies.

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: lora-service-edge
  name: lora-service
spec:
  selector:
    app: lora-edge
  type: LoadBalancer
  externalTrafficPolicy: Cluster
  externalIPs:
  - 172.18.255.200
  ports:
  - protocol: TCP
    port: 8088
    targetPort: 8088
```

FIGURE 3-17 AN EXAMPLE OF EXPOSING LORA APPLICATION AS A KUBERNETES SERVICE

Figure 3-18 presents the Kubernetes deployment scripts for LoRa Service. From the figure, we see that a deployment named 'lora-deployment-edge' is created, indicated by the *metadata.name* field. The deployment creates two replicated pods labelled as 'lora-edge'. Having two pods running the same instance in the system adds the redundancy, such that one can take over the traffic in case the other fails. Each pod runs one container, i.e., edge, which runs the Docker Hub image 'eabsics/5g-dive:edge_v2.1.8' to decode LoRa packets. The pods listen on port 8088 using TCP by default. Note that in this deployment, we set environment variables for the container IP address and port that run in the pod. Thus, when the pod restarts, the application will automatically connect to the newly assigned IP address and receive data from radio heads.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: lora-deployment-edge
  labels:
    app: lora-edge
spec:
  replicas: 2
  selector:
    matchLabels:
      app: lora-edge
  template:
    metadata:
      labels:
        app: lora-edge
    spec:
      containers:
        - env:
            - name: podIP
              valueFrom:
                fieldRef:
                  fieldPath: status.podIP
            - name: PORT
              value: "8088"
          image: eabsics/5g-dive:edge_v2.1.8
          name: edge
          command: ["/bin/bash", "-c", "--"]
          args: ["source /home/user/pybombs-packages/setup_env.sh && python2 Rx1.py"]
          ports:
            - containerPort: 8088

```

FIGURE 3-18 AN EXAMPLE OF KUBERNETES DEPLOYMENT YAML FILE FOR LORA

3.3.1.2. Intelligent Application of RF Fingerprinting

Internet of Things (IoT) devices are becoming pervasive in closed-loop control of different environments such as Industry, Home, Cities, etc. We need to ensure the secure connectivity of such devices to maintain the integrity of the sensor data and prevent undesirable data leaks. However, the low-power nature of the sensors limits the use of complex cryptographic functions for secure authentication of the devices. As an added security mechanism for these devices, RF fingerprinting can be used to verify the source of the received signal without extra energy and computing overhead on the IoT devices. RF fingerprinting uses the minute differences in the received signal caused by hardware impairments in the analog component of the particular device to identify that device.

Traditionally RF fingerprinting is performed for fixed networks, where the devices in the network is known apriori, which limits the scalability of networks. In D2.1 [1], we showed the feasibility of using RF fingerprinting with fixed IoT networks. However, this method limits the adaptability of RF fingerprinting in dynamic networks, where new IoT devices can join the network by performing a registration handshake. So, we propose an alternative design for RF fingerprinting using supervised contrastive learning to improve the adaptability of RF fingerprinting in dynamic IoT networks. As we need access to the radio signals, we consider single-hop networks with our virtualized IoT gateways.

We store the network association packet from the devices in our IoT network as a reference packet for that device. We use a deep neural network to learn the mapping from radio signals to unique set of features using supervised contrastive learning. By comparing the set of features for an incoming test message from a device with the reference packet for that device, we can verify the transmitter of a packet. We use a Siamese network which computes the difference between the sets of features for two inputs to authenticate an incoming packet. Siamese network is contrastive learning based deep neural network architecture. It consists of twin identical subnetworks which share the same parameters and weights. The set of features for radio signals from the same device would be similar. Hence the difference between the set of features should ideally be close to zero as shown in Figure 3-19. Similarly, the set of features for radio signals from different devices would be dissimilar, the difference between the set of features would be large as shown in Figure 3-20. This approach is transferable to any new network with new devices by reusing the same neural network structure with minimal data to adapt for new IoT network.

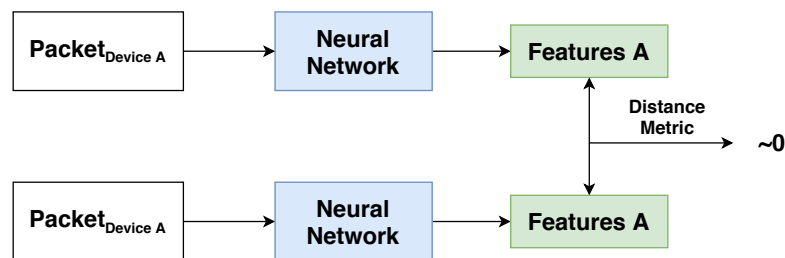


FIGURE 3-19: COMPARING PACKETS FROM THE SAME DEVICE USING SIAMESE NETWORK

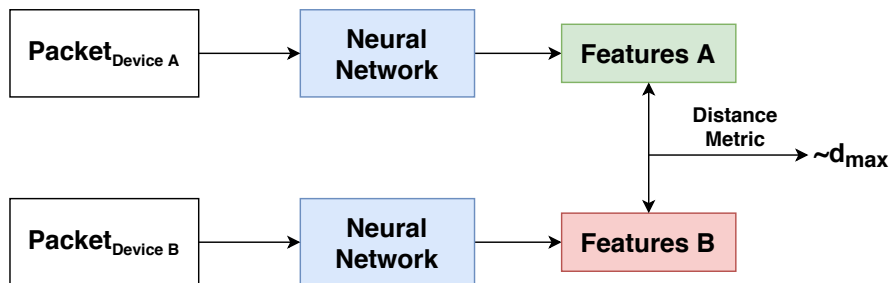


FIGURE 3-20: COMPARING PACKETS FROM THE DIFFERENT DEVICES USING SIAMESE NETWORK

Offline Training

Initially, we deploy M IoT sensors in a single hop network with 1 gateway in a laboratory environment. We collect radio signals corresponding to N packets from each of these M devices at the gateway. The gateway sends these radio signals to the cloud where we train the deep neural network to extract the set of features from these radio signals. Each packet is sliced up into smaller slices, with a sliding window operation for making the learned features shift-invariant. We term this collection of $M*N$ signals as our dataset.

We randomly sample a pair of slices from the training dataset and tag an ideal output for the pair. For generating the ideal output, we check if the random slices originate from the same device. For slices originating from the same device, we label the output for the pair of input slices as one, otherwise it is tagged as zero.

We structure the deep neural network as a stack of feature extraction layers followed by a sequential representation layer as shown in Figure 3-21. The feature extraction layers extract the local temporal features embedded in the slices. The sequential representation layer learns the sequential characteristics of the local features extracted by the feature extraction layers over a whole slice. The sequential representation layer outputs the vector of extracted features for each input slice. We compute a L1 norm of the output vectors for both the inputs and compare it with the labelled output using a contrastive loss function. Contrastive loss is best suited for our model as it minimizes the difference for similar sets of features and maximizes the difference for dissimilar sets of features. The calculated loss is backpropagated to the layers of the neural network to adjust their parameters (weights and biases) with respect to the loss. Please note, in Figure 3-21, we update and propagate the loss to only one neural network as the same neural network is instantiated twice for two inputs. This training process continues until the training loss becomes stable.

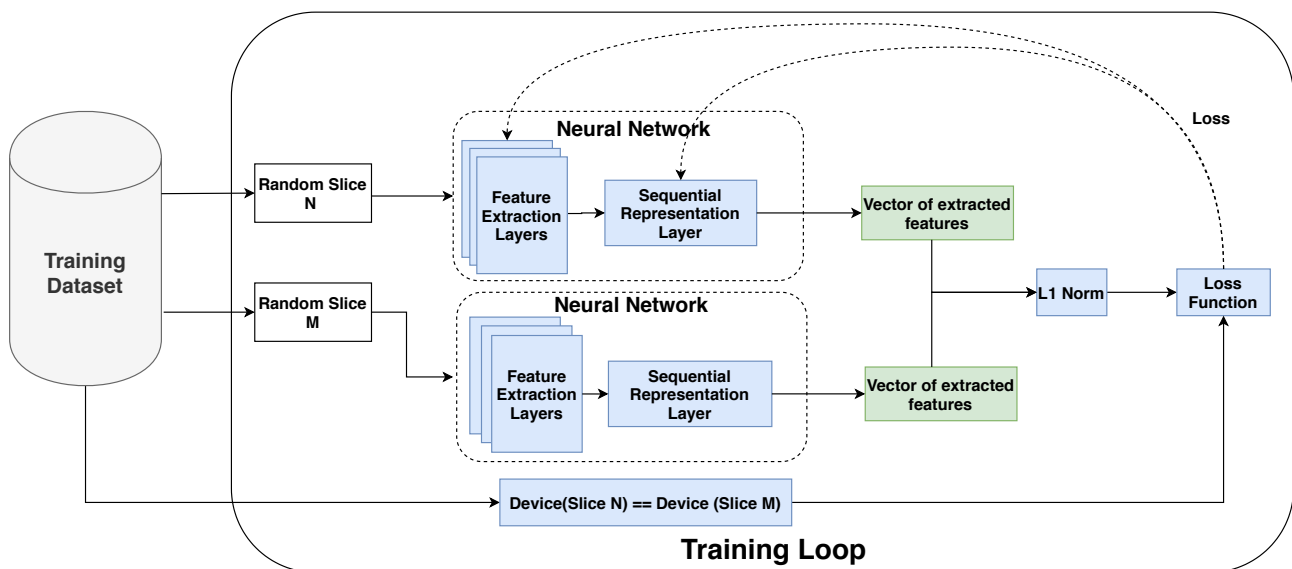


FIGURE 3-21: TRAINING PROCESS

Device Authentication

Once the model has been trained, it can be deployed at the IoT gateway. We use the authentication framework shown in Figure 3-22.

We store the reference packet from all the devices in the IoT network in the reference sample database, which is indexed by the MAC ID of the device. Note here the MAC ID is just an example. Any unique ID e.g., digital certificate keys, sim card info etc, can be used to index the reference packet of a device in the database. We first process an incoming packet to obtain the MAC ID of the transmitter. Next, we randomly sample an ensemble of n slices from that node's reference packet and the incoming packet. We compare the output sets of features across the n slices using our Siamese network (neural network followed by L1 norm). If the L1 norm is less than a predefined threshold, we verify that the transmission is from the same node from which the reference packet is extracted. Hence the packet is now

authenticated and can be processed by the protocol stack. If the packet has a higher L1 norm than the threshold, we can discard the packet and send warning to the management system.

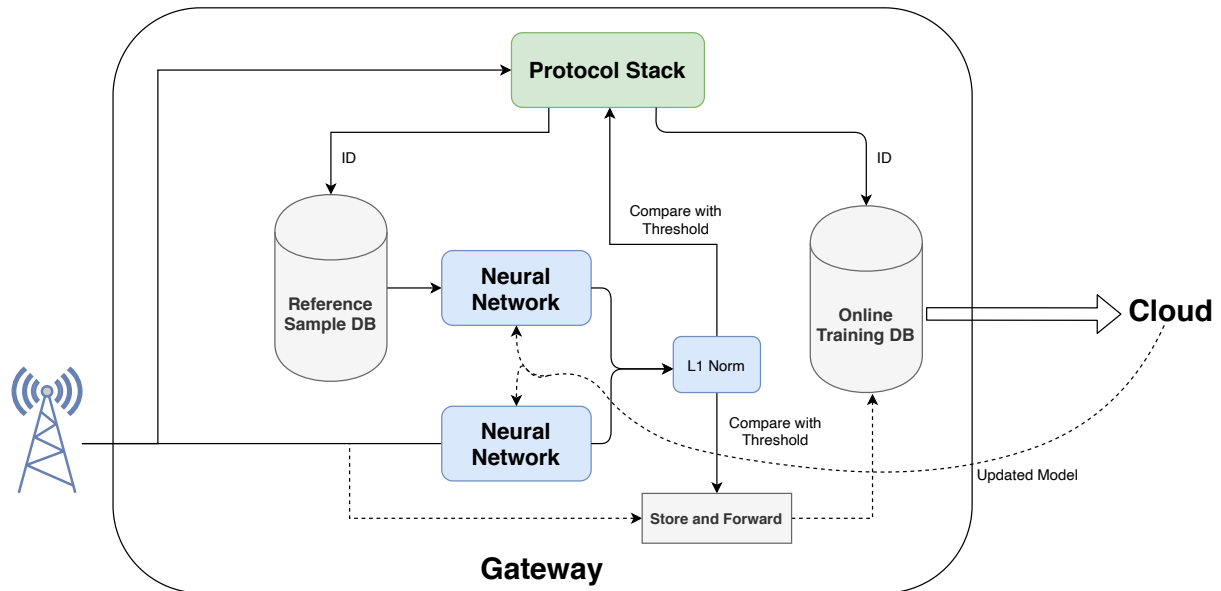


FIGURE 3-22: AUTHENTICATION AND PERIODIC UPDATE FRAMEWORK

We store all incoming authentic packets in our online training database. Each packet in the database is indexed by their ID. We periodically transfer the database to the cloud to retrain and update the deep neural network

3.3.2. Mapping to the DEEP Platform

This section presents mapping of mMTC use case to the DEEP platform, focusing on BASS service instantiation and IESS automation. In Section 3.3.2.1, we show the workflow for deployments of LoRa service and IEEE 802.15.4 service using BASS for service deployment, instantiation and management. In Section 3.3.2.2, we present the workflow for mapping RF fingerprinting to the IESS for model training.

3.3.2.1. BASS Service Instantiation for mMTC

Figure 3-23 illustrates the mapping of mMTC use case to the BASS. In the mMTC use case, the BASS is used on the edge to deploy and manage mMTC system which comprises a business translator, a vertical service manager and an orchestrator driver. During runtime, remote administrator can interact with the BASS and request mMTC service deployment provided by VSB as discussed in Section 2.2.2.4 (step 1). To integrate the BASS and the OCS, a translation from the VSD to Kubernetes deployment is done via a Kubernetes driver. Then, the BASS instantiates the mMTC service with the customized configuration (step 2). The instantiation request is received by the VIM and the OCS deploys the mMTC service in the EFS (step 3). For IoT devices, the status of the devices (e.g. on/off status) is reported to the OCS (step 4). Besides, communication stack information (e.g. deployment running status, error, etc.) is

also reported to the OCS (step 4). The BASS collects the status information and transmit the data to the remote administrator (step 5 and 6).

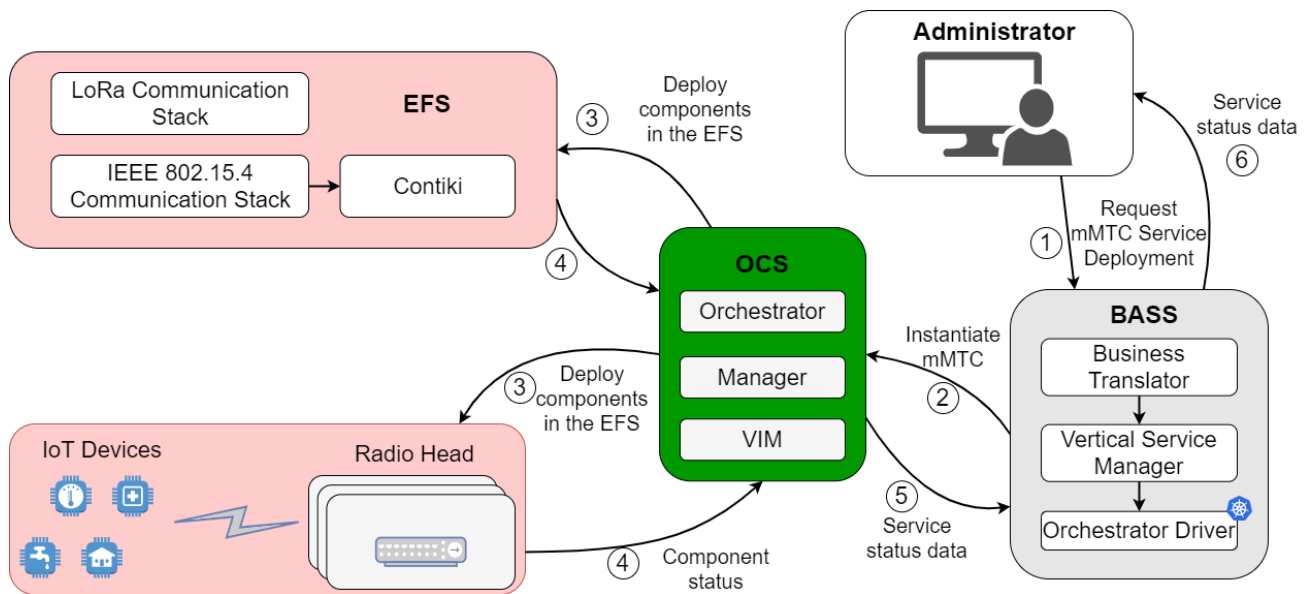


FIGURE 3-23 MMTC BASS DEPLOYMENT

3.3.2.2. IESS Automation for RF Fingerprinting Module

We show the IESS mapping of the RF Fingerprinting module in Figure 3-24. All IoT devices send their packets to their local radiohead (gateway). The radio signals corresponding to these packets are sent to the IoT communication stack to be processed (step 1). The IoT communication stack processes the radio signals and provides the processed data to the IoT application. Consequently, it also stores the radio signals in the DASS data storage (step 2). The stored radio signals are used for IESS model training using the process of offline training described previously (step 3). The trained model is stored in the model catalogue (step 3) and is sent to the BASS to be loaded and included in the mMTC service (step 4). The BASS interacts with the OCS through the Orchestration Driver, requesting the instantiation of the RF Fingerprinting module (step 5). The OCS deploys the RF Fingerprinting module at the EFS (step 6) with the status of the operation sent back to the OCS (step 7). The OCS updates the RF Fingerprinting Module status to the BASS (step 8).

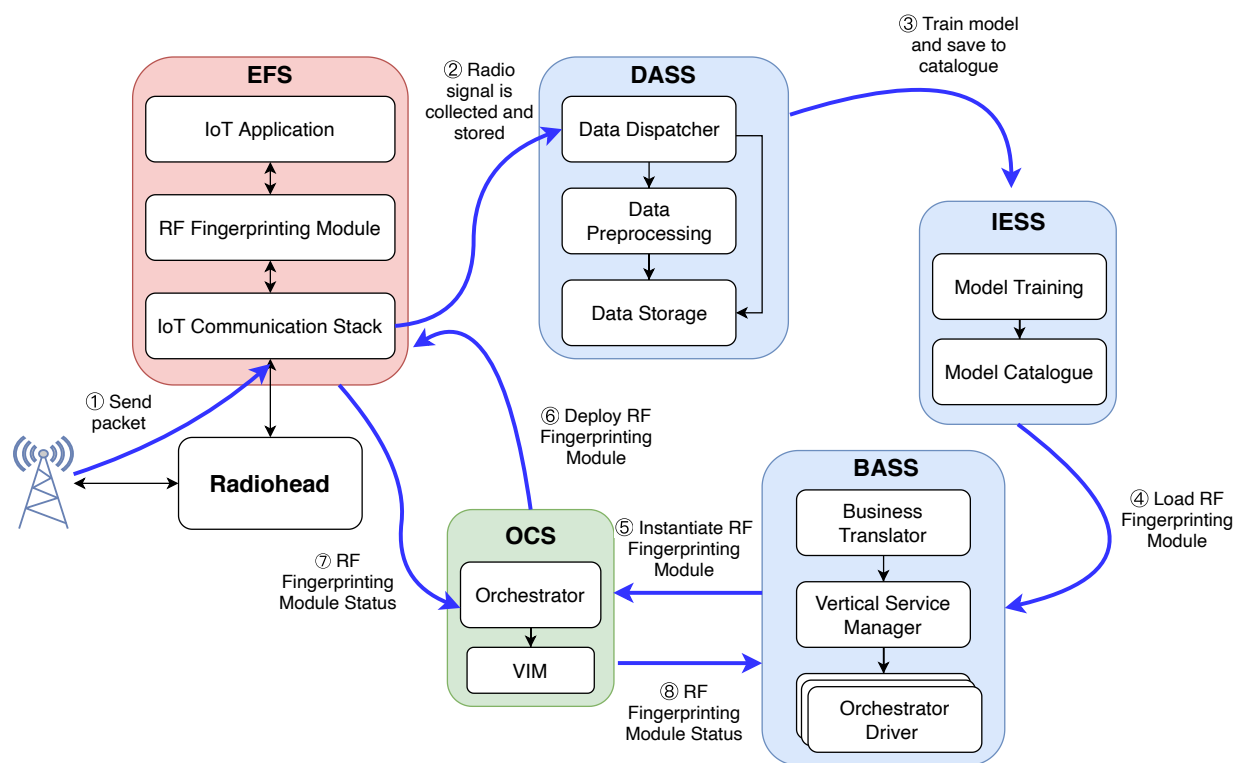


FIGURE 3-24: RF FINGERPRINTING IESS MAPPING

4. 5G-DIVE Solution for Disaster Relief Using Autonomous Drone

This section provides the refined and final key modules design for Disaster Relief Using Autonomous Drone Scouts (Figure 4-1). This will include updates and refinements on the modules already introduced in D2.1 [1], as well as the addition of new modules in both ADS, Use Case 1 Drones Fleet Navigation, and ADS Use Case 2 Intelligent Image Processing for Drones. Details on Use Case 1 will be described in Section 4.1. Details on Use Case 2 will be described in Section 4.2. And finally, yet importantly, the mapping of ADS Use Case 1, and ADS Use Case 2 to the DEEP platform will be presented in Section 4.3.

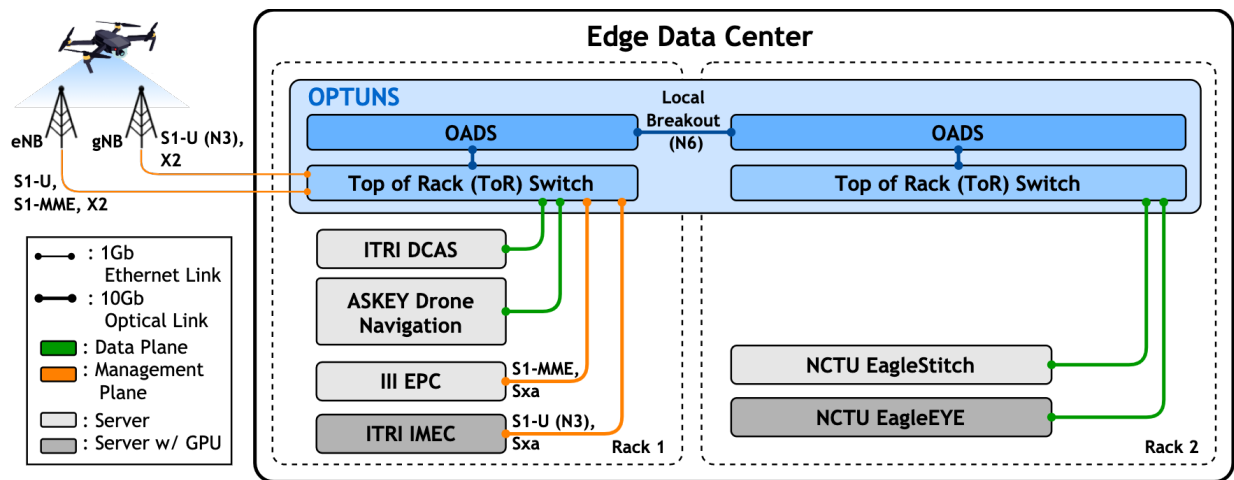


FIGURE 4-1 5G NSA AND EDGE SYSTEM BLOCK DIAGRAM FOR ADS

4.1. ADS Use Case 1: Drone Fleet Navigation

Drone fleet navigation is important functionality needed during a disaster relief mission. In this part, we will introduce the enhanced feature which will allow a smooth drone flight and resource management.

4.1.1. Key Module Design

During the disaster relief mission of the drone fleet, the drone navigation server and drone collision avoidance system (DCAS) function is applied to support drones for executing missions. However, the initial design of DCAS handles the computing in the fog. In this deliverable, we considered the DCAS to be adopted at the edge. This will be part of improving the drone navigation server software which can view and control multiple drones at the same console simultaneously. The new drone navigation software will facilitate the adoption of DCAS at the edge. On the other side, the benefits of iDrOS (Internet Drone Operating System) to support drone fleet navigation functionalities will be quite important. The iDrOS will enable the modules to run on the drone itself or at the edge. Consequently, this section focuses on two systems namely DCAS and iDrOS.

4.1.1.1. DCAS

During the disaster relief mission of the drone fleet, the drone navigation server at the edge and DCAS at the fog (i.e. drone) are applied to support drones for executing missions. The design of DCAS will remain the same as introduced in D2.1 [1]. In this deliverable, DCAS will be adopted in the edge as elaborated earlier. In particular, we will take advantage of the 5G-Connectivity to provide low latency to adopt drone avoidance functionality at the edge. Also, this means utilizing the computing capabilities at the edge. Basically, the drones will transmit the GPS information and drone ID back to the navigation server at the edge over the 5G network. The new design of the navigation server is capable of monitoring several drones at the same time and detect collision. The current design for DCAS at the edge will only hoover the drone and send the request for the mission operator to handle this situation. In near future, the DCAS at the edge will react automatically and fly drones in a different pattern to avoid the collision. One of the candidate designs is to fly in a swapping pattern similar to the model adopted in DCAS.

4.1.1.2. iDrOS

iDrOS (Internet Drone Operating System) is a system support layer simplifying the development of drone applications implemented as a pipeline of data processing components. A component here is to be considered as an individual functionality in charge of a specific step in the data processing pipeline; for example, performing filtering of input image frames or relaying the results of object detection functionality to the backend. iDrOS facilitates implementing these applications by equipping programmers with an actor-based programming model. Single components are mapped to software actors. Actors are loosely coupled and interact via a data bus layer, shown in Figure 4-2 iDros architecture, accessed via pattern matching.

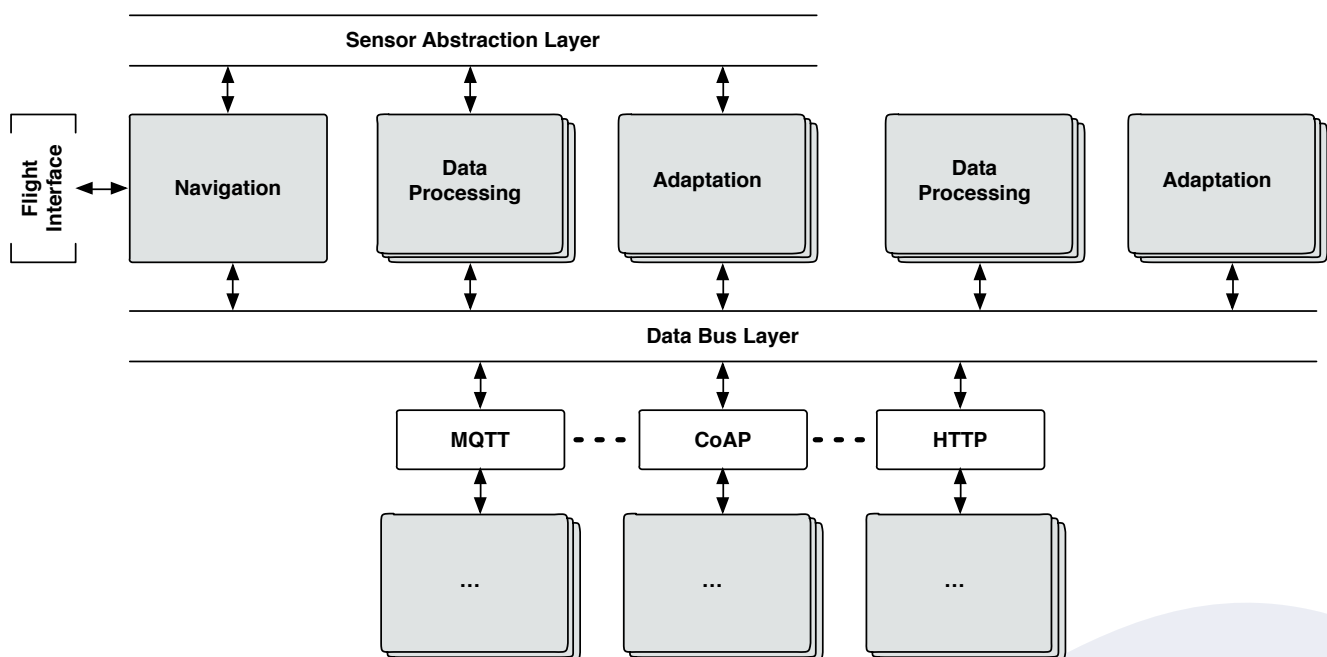


FIGURE 4-2 IDROS ARCHITECTURE

The distinctive feature of iDrOS components is the programmers' ability to migrate individual functionality across different devices. Programmers can tag part of a component state as non-volatile, which causes iDrOS to migrate the state along with the code corresponding to the functionality to be migrated. The original implementation of iDrOS, however, assumed the availability of a previously configured iDrOS instance whenever the functionality was to be migrated.

Within 5G-DIVE, we furthered both the programming model and the underlying implementation of iDrOS to cater to high resource mobility and volatility. We have, in particular, added the ability to migrate entire iDrOS instances in addition to the existing functionality to migrate individual component functions. We leverage fog05 to this end and implement a custom orchestrator that, based on a given objective function that represents the desired performance, dynamically monitors the network conditions and accordingly adjusts the deployment configuration.

Orchestrator: Requirements

The design of the orchestrator has been shaped around three objectives that it must achieve. First, the orchestrator must capture and store data about the current network conditions. This data includes the number of nodes in the network, network performance, etc. Application developers who develop applications that are to be deployed using the orchestrator should be able to add the custom data they need to be recorded. Second, the orchestrator must utilize this data to make decisions on whether and how to modify the deployment configuration. Those decisions take the form of actions to instantiate a new iDrOS instance on a particular node, stop an instance that is currently running on a node, or migrate an instance from one node to another.

The orchestrator must generate those actions to convert the current deployment configuration to an optimized one. That is generated from analysing the current data about the network and also application-specific data that has been added by the application developers. Third, the orchestrator must be able to perform the conversion of the current deployment to the optimized deployment that has been previously generated. This means that the orchestrator must be able to interface with the edge deployment platform, which is Fog05 in this case, to modify the current edge deployment to match the intended deployment.

Orchestrator: Architecture

As shown in Figure 4-3 iDrOS Orchestrator architecture, the iDrOS orchestrator architecture is based on three modules: Surveying, Analysis, and Execution.

- The Surveying Module is responsible for gathering data from nodes in the network.
- The Analysis Module is responsible for utilizing that data to scrutinize the current deployment configuration and generate an optimized deployment configuration.

- The Execution Module is responsible for turning the current deployment configuration into the optimized one.

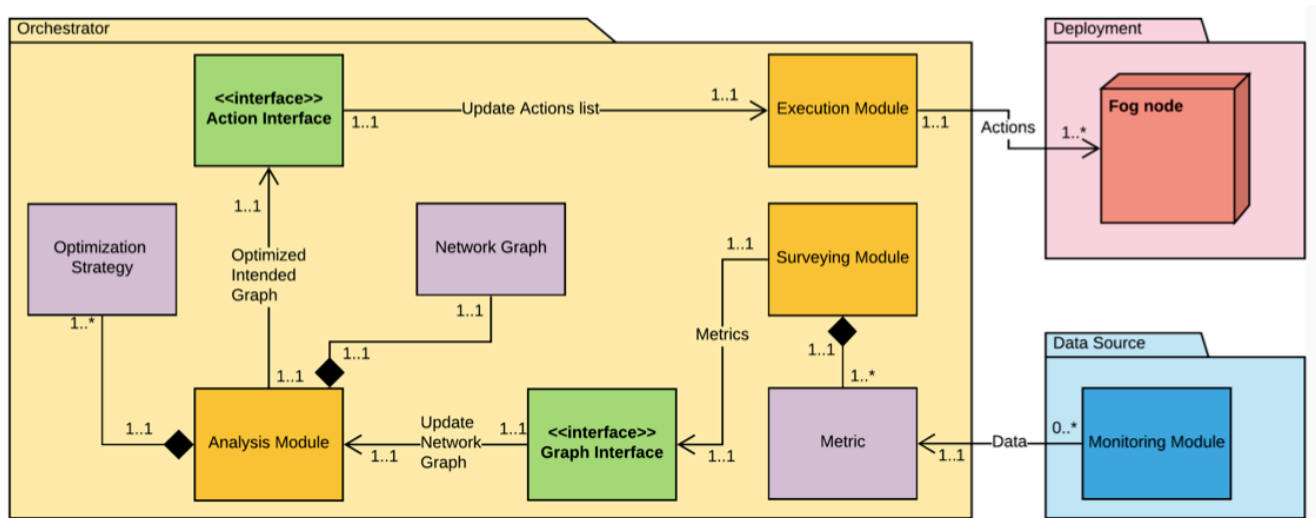


FIGURE 4-3 IDROS ORCHESTRATOR ARCHITECTURE

The deployment configuration refers to a graph of all the nodes in the network, which includes aerial and edge nodes. Edges in this graph represent the distance between the nodes, based on a conceptual notion of distance that is generally application-specific. The 3 modules are each running periodically so that updated information is constantly being factored into the orchestration. It is important to mention that the Monitoring Module is an additional module that is not strictly part of the orchestrator but rather runs on each node in the network to gather important monitoring data that will be sent to the orchestrator.

The architecture of the orchestrator was based on maintaining a continuous flow of data. The flow of data goes from all the Monitoring Modules to the Metric object. Those Metric objects are collected by the Surveying Module and passed to the Graph Interface. The Graph Interface uses those Metrics to update the Network Graph object stored in the Analysis Module. The Analysis Module passes the Network Graph object to the highest priority Optimization Strategy that meets its execution conditions. The Optimization Strategy passes back an optimized intended deployment graph to the Analysis Module, which forwards along with the current deployment graph to the Action Interface. The Action Interface generates a list of Actions necessary to convert the Network Graph to the optimized intended graph. This list of Actions is passed to the Execution Module which executes those actions on the nodes that require any changes.

4.2. ADS Use Case 2: Intelligent Image Processing

Intelligent Image Processing provides the capability to locate PiH in a disaster-impacted area in real-time based on aerial drone video surveillance. The detection and localization of PiH will be done by the EagleEYE system. For the final design, we updated a couple of modules in EagleEYE system. Namely Data Offloader module, Dual Object Detection module, and Visualizer module. In addition, we are also adding the EagleStitch system and the Drone Data Processor system.

Details on the additions are as follow:

1. We added the EagleStitch image stitching system which gives us the capability to perform panorama stitching of a disaster-impacted area. This will help the rescue team in assessing the disaster impacted area during the rescue mission.
2. We added the Drone Data Processor system which gives us the capability to inject metadata information to a drone stream. This is crucial as it allows us to differentiate between multiple drone source inputs. In addition, the Drone Data Processor system uses Zenoh [7] as the DASS platform which will take care of data exchange efficiently and has several capabilities such as data storage and data pre-processing.

4.2.1. Key Module Design

For the final design of intelligent image processing system, we have EagleEYE system for performing PiH detection and localization, EagleStitch system for performing 2D stitching of an area, and a Drone Data Processor system for drone data pre-processing. Overview of the whole system can be seen in Figure 4-4.

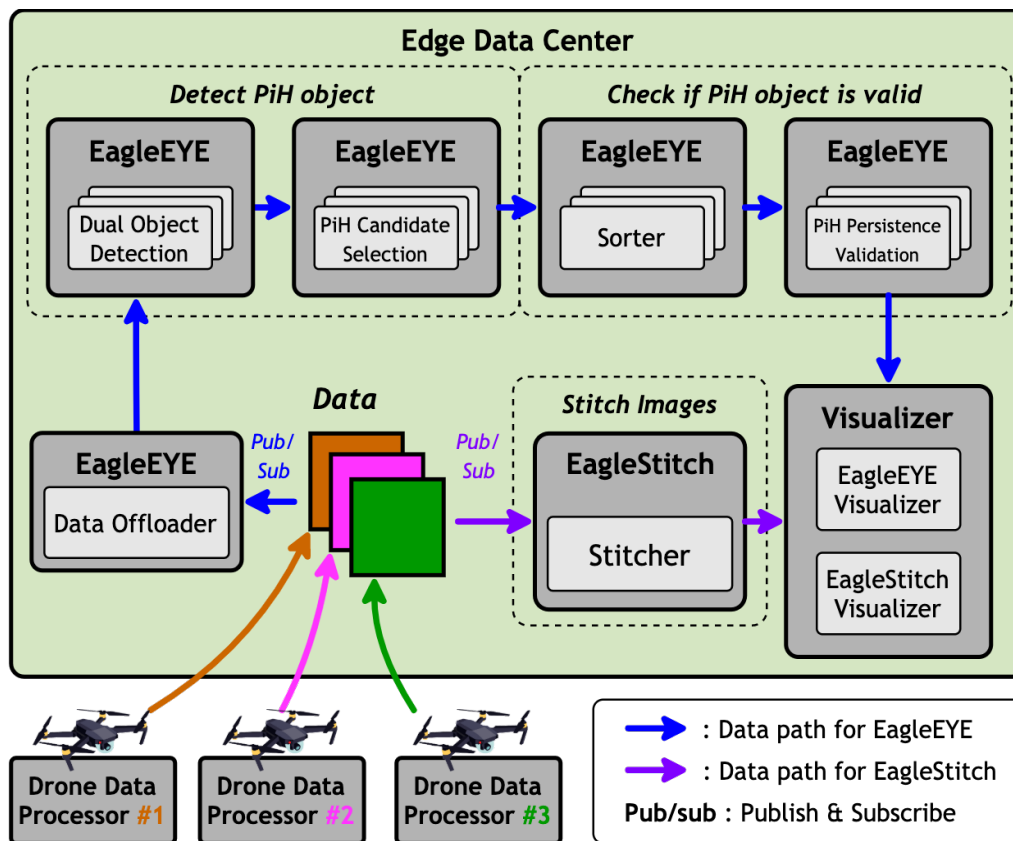


FIGURE 4-4 ADS USE CASE 2 SYSTEM OVERVIEW

4.2.1.1. Data Offloader (EagleEYE system)

In the first release, traditional round-robin technique is utilized to offload data (in this case image frame) to an available dual object detection worker. However, in our testing, we find out that the

traditional round-robin technique is not very efficient. It will stall frame offloading to a busy worker until it is available. This causes a lot of delays and makes available worker utilization low. For the final solution, we updated the round robin technique to be a little bit more dynamic. With this, the offloader will be able to offload frames to any available dual object detection worker (e.g.: worker-1 → worker-3 → worker-2 → worker-1). Compare this to the traditional round-robin technique that can only offload frame to available dual object detection worker that is in order (e.g.: worker-1 → worker-2 → worker-3 → worker-1).

4.2.1.2. Dual Object Detection (EagleEYE system)

In the first release, the dual object detection module consists of 3 workers to handle all of the detection tasks coming from a single drone. In the final solution, we update the dual-object-detection module to contain more workers to handle all of the detection tasks coming from multiple drones. The number of workers can be set according to the available GPU resources as well as the number of drones currently under operation. Ideally, the number of workers will be scaled up/down automatically to handle the processing load.

4.2.1.3. Visualizer System

In the first release, RTSP server is utilized to visualize the output of EagleEYE PiH detection. However, we find out that using RTSP to visualize the output can incur extra latency. This extra latency comes from the extra processes that happen inside RTSP as it is aimed more for media streaming. For the final solution, we will visualize the output directly on a frame-by-frame basis to display the output of both EagleEYE and EagleStitch system.

4.2.1.4. Sorter (EagleEYE system)

The sorter module is a new module placed after the dual object detection and PiH candidate selection module. The sorter module inputs are jumbled data coming from previous modules. Sorter is required so that the other modules after it (PiH persistence validation, and visualizer module) can function properly as they rely on a sorted data. Sorter will sort data according to frame sequence and drone ID. The sorting itself is based on the sorting network technique [42]. Then, the sorter will output the sorted data to the PiH persistence validation module.

4.2.1.5. EagleStitch System

EagleStitch system is a new addition for ADS Use Case 2. EagleStitch system itself consists of a

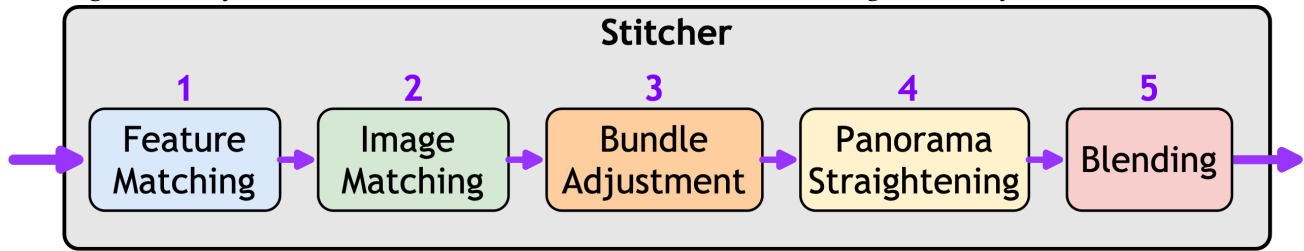


FIGURE 4-5 EAGLESTITCH SYSTEM STITCHER MODULE PIPELINE

single Stitcher module. The EagleStitch system will be installed and run on the edge. The stitcher module input is images of the trial site's surrounding area. The stitcher module performs 2D-Stitching on those images. In our design, we are using the stitching algorithm proposed in [43]. The number of images to be stitched will depend on the target area, as well as the flying characteristic of the drone. The processing pipeline of the stitching algorithm can be seen in Figure 4-5. A brief overview of the pipeline is the following:

1. **Feature Matching**
To detect features in an image (e.g.: corner, curves).
2. **Image Matching**
To match for images that have the same features.
3. **Bundle Adjustment**
To bundle all images with the same features.
4. **Panorama Straightening**
To align the bundled images so that they are not slanted or rotated.
5. **Blending**
To adjust and correct the gain (brightness) of the images being stitched as well as to remove the seams (edges) in the stitched images.

4.2.1.6. Drone Data Processor System

The drone data processing system is a new addition to ADS Use Case 2. This system will be installed and run on a fog device onboard the drone. The drone data processor is used to inject

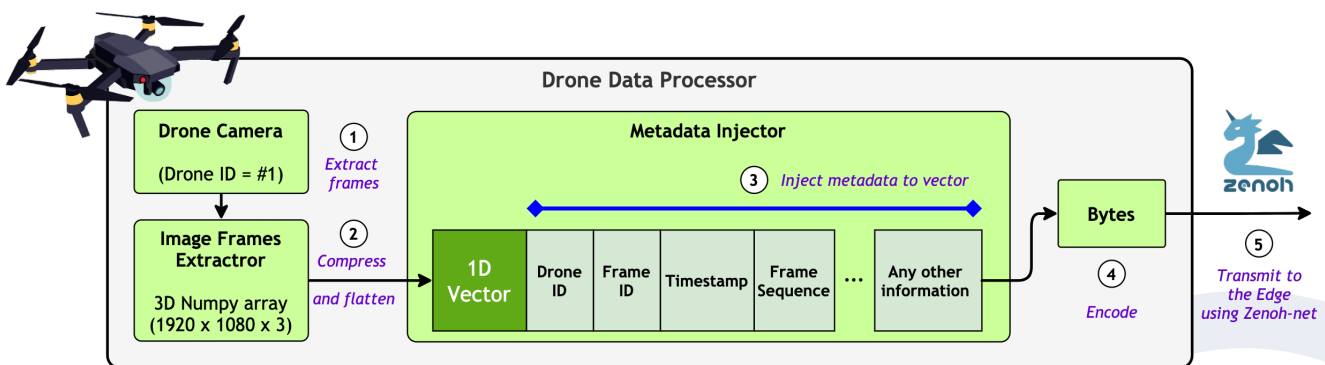


FIGURE 4-6 DRONE DATA PROCESSOR SYSTEM

metadata onto the captured images before sending them to the edge. This metadata information offers a simple way to differentiate the data that is coming from different drone sources. The workflow of the drone data processor can be seen in Figure 4-6.

The drone data processing workflow is as follows:

1. The drone starts to capture Full HD video using the onboard drone camera. From the video, raw image frames are then extracted. In our case, we will extract 30 image frames per second. These raw image frames are then converted into a 3D Numpy array. At this step, the images are still in their original resolution.
2. The 3D Numpy array of the image frames are then compressed and flattened into a 1D Numpy array. The compression is meant to reduce the size of the image frame and to save network bandwidth during transmission. For the compression, we are using lossy JPEG compression.
3. The 1D Numpy array is then injected with information such as Drone ID, timestamp, frame sequence, and any other relevant information. The information will be injected at the end of the 1D Numpy array.
4. The 1D Numpy array is then encoded into bytes for transmission.
5. The bytes are then published to the Edge using Zenoh-net.

4.3. ADS Mapping to the DEEP Platform

Figure 4-7 shows the complete mapping of ADS Use Cases to the DEEP platform. In ADS Use Case 1, the DASS is used for the fog device. In ADS Use Case 2, the DASS is used on both the fog device as well as the edge. DASS is used to perform data pre-processing and data storage tasks. The BASS is used on the edge for the deployment and management of EagleEYE and EagleStitch system. Apart from deployment and management, the BASS is also used for active monitoring. This active monitoring is especially useful in the management of key modules that benefit greatly from scaling, such as EagleEYE's dual object detection module. Finally, the IESS is used on the edge for the automatic training and storing of the trained model for EagleEYE's dual object detection module. Details on the object detection algorithm, object of interest for detection (e.g.: 'person', 'flag'), as well as the desired precision level will be input to the IESS for an automated training process. The different trained models can also be stored in IESS for future use.

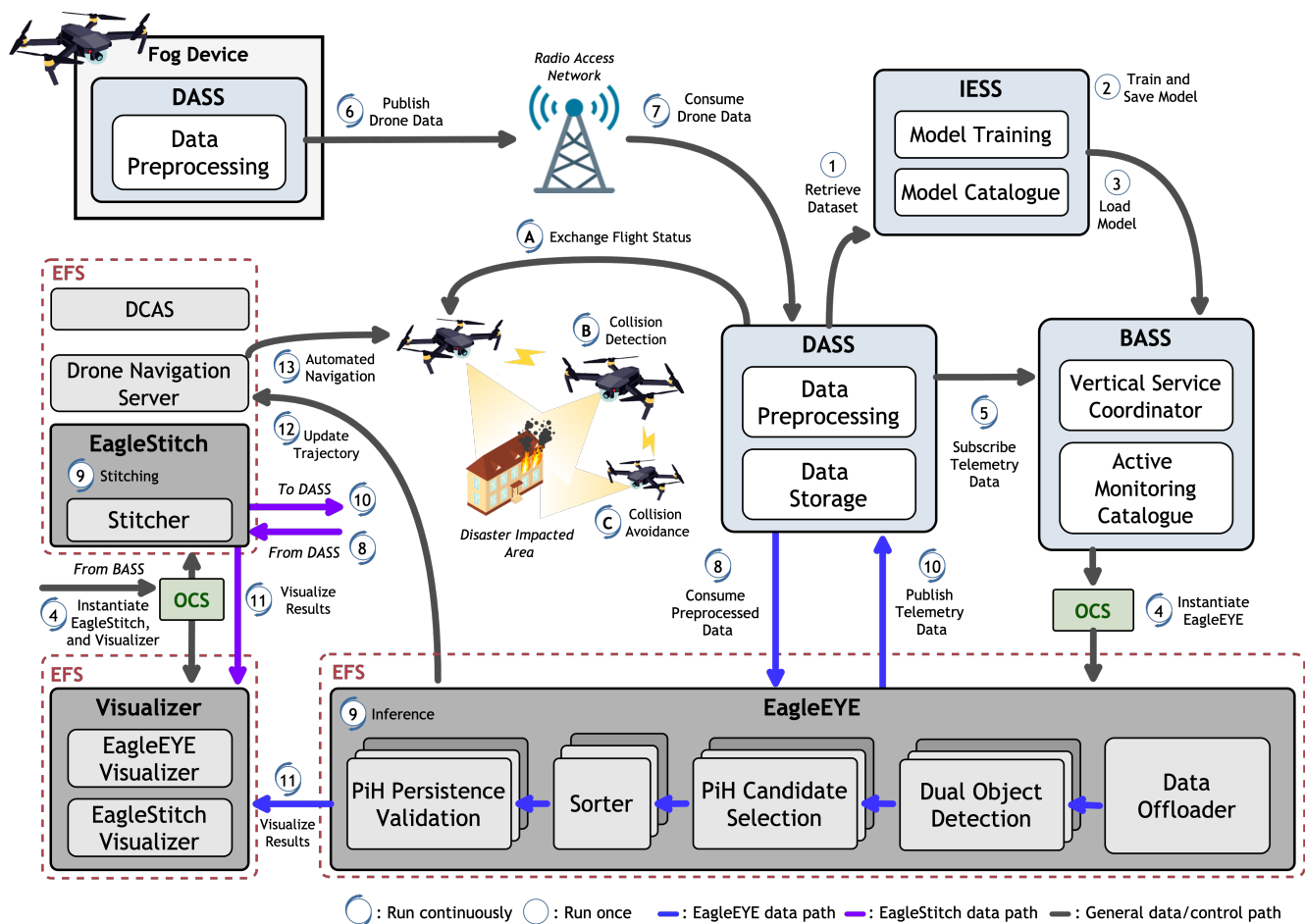


FIGURE 4-7 ADS USE CASE MAPPING TO THE DEEP PLATFORM

The flow for ADS Use Case 1 is from Step A to Step C, while the flow for ADS Use Case 2 is from Step 1 to Step 15.

The complete workflow of the ADS Use Case 1 are as follows:

- A. Exchange flight status: based on Zenoh, the EFS, DASS stores the drone data such as Drone ID and GPS. Then, each drone broadcasts the trajectory repetitively.
- B. Collision detection: Each drone location is used by drone fleet navigation software to decide the drone mission at the edge. Where the done automation path is sent to the edge, Based on the DCAS detection mechanism [1].
- C. Collision avoidance: DCAS will use the automation path and the stored data to trace any risk and then change the drone path if collision risk is detected in the previous step.

The complete workflow of the ADS Use Case 2 are as follows:

1. The IESS first retrieves the dataset from the DASS for training. This dataset will be a collection of images that contain person and flag objects. The dataset can be a custom dataset or sourced from the publicly available repository.
2. The IESS then trains object detection models based on the dataset provided in Step 1. After training, the trained models are stored in the IESS model catalogue for future use and reference. After being stored in the IESS model catalogue, the trained model will also be available for other verticals to use.
3. The BASS then reads the VSB and VSD to prepare for the instantiation of EagleEYE and EagleStitch. The VSB and VSD will contain all of the necessary parameters for the instantiation. In the case of EagleEYE instantiation, the BASS will load the previously trained model from the IESS. After reading the VSB and VSD, the BASS performs the actual instantiation through the help of the OCS.
4. The OCS instantiates EagleEYE and EagleStitch with the instruction provided by the BASS. Both EagleEYE and EagleStitch will be instantiated as an EFS component on the edge. A monitoring probe will also be installed in both EagleEYE and EagleStitch for telemetry data collection.
5. The BASS subscribes to the telemetry data stored at the DASS. Based on this telemetry data, the BASS will be able to perform management activities such as scaling up/down system deployment.
6. The drone then starts publishing data to the edge through the Radio Access Network (RAN). The published data is a pre-processed data by the drone data processor.
7. At the edge, the DASS consumes the published data and store them.
8. EagleEYE and EagleStitch consume the pre-processed drone data stored in the DASS simultaneously.
9. EagleEYE and EagleStitch perform computation. EagleEYE will perform PiH detection, while EagleStitch will perform 2D-Stitching.
10. EagleEYE and EagleStitch publish the collected telemetry data to the DASS. For EagleEYE, an example of the telemetry data is per frame inference time latency, the number of image frames

processed, worker utilization. For EagleStitch, an example of the telemetry data is stitching latency and stitching status.

11. Both EagleEYE and EagleStitch results are visualized for the operator. In the case of EagleEYE, it will be a video that is marked with the bounding boxes of PiH detection, as well as PiH GPS location. For EagleStitch, it will be a stitched image of the target area.
12. Based on the PiH detection result, EagleEYE will send the PiH GPS location information to the drone navigation for drone trajectory update.
13. The Drone navigation calculates waypoints for drone automatic navigation based on the PiH GPS location information received in the previous step. These waypoints are then sent to the drone through the RAN.

5. Conclusion

This deliverable presents the final specification of the 5G-DIVE solution for the use cases targeted in the I4.0 and ADS vertical pilots.

Section 2 presented the final design framework for the solution targeted in 5G-DIVE. It first describes the 5G connectivity solution used to support the verticals. Next, 5G-DIVE DEEP platform was presented. Describing in details the update and improvement made compared to the previous design reported in D2.1 [1].

Section 3 describe in details the final system design for each use cases. Building on top of the design framework in Section 2. Elaboration on how each use case maps to the DEEP platform, as well as how each use cases interacts with the DASS, BASS, and IESS are also presented.

The final specification described in this deliverable served as a basis for the implementations. The achievement for this deliverable are as follows. 5G Connectivity solution of each use cases, final design of the DEEP platform, as well as a complete solution tailored for each use cases that utilizes the DEEP platform. Evaluation on the implementations in each use cases will be reported in WP3.

6. References

- [1] "D2.1," 2020. [Online]. Available: https://5g-dive.eu/wp-content/uploads/2021/01/D2.1-5G-DIVE-innovations-specification_v1.0_compressed.pdf. [Accessed 22 April 2021].
- [2] Torbjörn Cagenius et al., "Simplifying the 5G-ecosystem by reducing architecture options," *Ericsson Technology Review*, November 2018.
- [3] Yuang, M., Tien, P.L., Ruan, W.Z., Lin, T.C., Wen, S.C., Tseng, P.J., Lin, C.C., Chen, C.N., Chen, C.T., Luo, Y.A. and Tsai, M.R., "OPTUNS: optical intra-data center network architecture and prototype testbed for a 5G edge cloud," *Journal of Optical Communications and Networking*, vol. 12, pp. A28--A37, 2020.
- [4] "D1.3," 06 2021. [Online]. Available: <https://5g-dive.eu/wp-content/uploads/2021/06/D1.3-Final.pdf>.
- [5] Zhang, Lixia, Alexander Afanasyev, Jeffrey Burke, Van Jacobson, K. C. Claffy, Patrick Crowley, Christos Papadopoulos, Lan Wang, and Beichuan Zhang, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66-73, 2014.
- [6] "Minica," 14 November 2019. [Online]. Available: <https://github.com/jsha/minica>. [Accessed 31 May 2021].
- [7] "Zenoh," ADLINK, [Online]. Available: <https://www.adlinktech.com/en/Zenoh.aspx>. [Accessed 22 April 2021].
- [8] "Eclipse fog05," [Online]. Available: <https://fog05.io/>. [Accessed 26 05 2021].
- [9] "Bootstrap," [Online]. Available: <https://getbootstrap.com/>. [Accessed 09 June 2021].
- [10] "Spring Security," [Online]. Available: <https://spring.io/projects/spring-security>. [Accessed 26 05 2021].
- [11] IETF, "The 'Basic' HTTP Authentication Scheme," 09 2015. [Online]. Available: <https://tools.ietf.org/html/rfc7617>. [Accessed 26 05 2021].
- [12] IETF, "JSON Web Token (JWT)," 05 2015. [Online]. Available: <https://tools.ietf.org/html/rfc7519>. [Accessed 26 05 2021].
- [13] IETF, "JSRs: Java Specification Requests," [Online]. Available: <https://jcp.org/en/jsr/detail?id=250>. [Accessed 26 05 2021].
- [14] "Ansible," Red Hat, [Online]. Available: <https://www.ansible.com/>. [Accessed 31 May 2021].
- [15] "Jinja," [Online]. Available: <https://palletsprojects.com/p/jinja/>. [Accessed 31 May 2021].

- [16] "Jsonnet," [Online]. Available: <https://jsonnet.org/>. [Accessed 31 May 2021].
- [17] "YAML," [Online]. Available: <https://yaml.org/>. [Accessed 31 May 2021].
- [18] Sobel, Jonathan M., and Daniel P. Friedman., "An introduction to reflection-oriented programming," in *Proceedings of reflection*, 1996.
- [19] "Hibernate Validator," [Online]. Available: <https://hibernate.org/validator/>. [Accessed 31 May 2021].
- [20] "Jakarta Bean Validation specification," 2019. [Online]. Available: <https://beanvalidation.org/2.0/spec/>. [Accessed 31 May 2021].
- [21] "Prometheus Client Libraries," [Online]. Available: <https://prometheus.io/docs/instrumenting/clientlibs/>. [Accessed 31 May 2021].
- [22] "Vector," [Online]. Available: <https://vector.dev/>. [Accessed 31 May 2021].
- [23] "Telegraf," [Online]. Available: <https://www.influxdata.com/time-series-platform/telegraf/>. [Accessed 31 May 2021].
- [24] "Influx DB," [Online]. Available: <https://docs.influxdata.com/influxdb/v2.0/get-started/>. [Accessed 31 May 2021].
- [25] B. Burns, *Designing Distributed Systems*, O'Reilly Media, Inc., 2018.
- [26] "Telegraf Operator," [Online]. Available: <https://github.com/influxdata/telegraf-operator>. [Accessed 31 May 2021].
- [27] "Swagger Open API Specification," [Online]. Available: <https://swagger.io/specification/>. [Accessed 31 May 2021].
- [28] "Swagger," [Online]. Available: <https://swagger.io/>. [Accessed 31 May 2021].
- [29] "H2O AutoML: Automatic Machine Learning," [Online]. Available: <https://docs.h2o.ai/h2o/latest-stable/h2o-docs/automl.html>. [Accessed 01 June 2021].
- [30] "YOLOv3 in PyTorch," [Online]. Available: <https://github.com/ultralytics/yolov3>. [Accessed 01 June 2021].
- [31] "TensorFlow Core The Sequential Model," [Online]. Available: https://www.tensorflow.org/guide/keras/sequential_model. [Accessed 01 June 2021].
- [32] MongoDB Inc., "MongoDB – a cross-platform document-oriented database program," [Online]. Available: <https://www.mongodb.com/>. [Accessed 01 June 2021].
- [33] "MinIO," [Online]. Available: <https://min.io/>. [Accessed 01 June 2021].

- [34] “Amazon S2 API Reference,” [Online]. Available: https://docs.aws.amazon.com/AmazonS3/latest/API/Type_API_Reference.html. [Accessed 07 June 2021].
- [35] “Docker Registry,” [Online]. Available: <https://docs.docker.com/registry/>. [Accessed 01 June 2021].
- [36] “BentoML,” [Online]. Available: <https://www.bentoml.ai/>. [Accessed 01 June 2021].
- [37] “Jib,” [Online]. Available: <https://github.com/GoogleContainerTools/jib>. [Accessed 01 June 2021].
- [38] F. & U. R. & G. D. & M. J. Bulnes, “An efficient method for defect detection during the manufacturing of web materials,” *Journal of Intelligent Manufacturing*, 2014.
- [39] J. a. A. F. Redmon, “YOLOv3: An Incremental Improvement,” in *ArXiv abs/1804.02767*, 2018.
- [40] “Kubernetes Documentation,” [Online]. Available: <https://kubernetes.io/docs/concepts/>. [Accessed 31 May 2021].
- [41] “MetalLB Documentation,” [Online]. Available: <https://metallb.universe.tf/concepts/>. [Accessed 31 May 2021].
- [42] D. E. Knuth, in *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison–Wesley, 1997, p. 219–247.
- [43] M. a. L. D. G. Brown, “Automatic panoramic image stitching using invariant features,” *International journal of computer vision*, vol. 74, no. 1, pp. 59-73, 2007.
- [44] 5G-CORAL Project, “5G-CORAL,” May 2019. [Online]. Available: <http://5g-coral.eu/wp-content/uploads/2019/06/D3.2.pdf>. [Accessed 31 May 2021].
- [45] J. Baranda et al., “Realizing the network service federation vision: Enabling automated multidomain orchestration of network services,” *IEEE Vehicular Technology Magazine*, vol. 15, no. 2, pp. 48-57, 2020.
- [46] J. B. et al., “Nfv service federation: enabling multi-provider ehealth emergency services,” in *Proceedings of the International Conference on Computer Communications (INFOCOM'20)*, 2020.
- [47] “5G-CORAL,” August 2019. [Online]. Available: http://5g-coral.eu/wp-content/uploads/2019/09/D4.2_FINAL.pdf. [Accessed 31 May 2021].
- [48] K. Antevski, M. Groshev, G. Baldoni and C. J. Bernardos, “DLT federation for Edge robotics,” in *2020 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, Leganes, 2020.

- [49] "D1.1," 2020. [Online]. Available: https://5g-dive.eu/wp-content/uploads/2021/01/D1.1_Final.pdf. [Accessed 18 May 2021].

7. Appendix

In this section feasibility study of DLT-based federation, as well as workflow details on the integration of BASS and OCS are provided respectively in Section 7.1, and Section 7.2.

7.1. DLT-Based Federation Support

This section presents the applicability of Distributed Ledger Technology (DLT) as a mechanism of the *external federation support* element of the BASS. The goal of the DLT-based federation support is to improve the orchestration and control processes by automating the service federation across multiple administrative domains (ADs). Federation has been described as a concept for integrating multiple ADs at a different granularity into a unified open platform where the federated resources can trust each other at a certain degree. The **federation of resources** between ADs was introduced in 5G Coral Deliverable 3.2 [44] here we analysed the profit-maximized federations and advanced resource provisioning. In this section, we will focus on the **federation of services** between ADs, where network services deployment is extended over the infrastructure of an external domain. First, we will describe the service federation procedures. Then, we will elaborate on how the *external federation support* element of the BASS applies DLT for service federation. Finally, we will show experimental validation based on Edge robotics use case and summarize the obtained results.

Service Federation

Service federation is a concept where a **consumer domain** through its orchestrator requests an extension of a service (or part of a service) to be deployed over a **provider domain**. The orchestrator of the provider domain monitors the complete deployment process of the service extension. In order to successfully complete a service federation [45] [46], there are several steps that are executed in sequence:

- **Registration:** initial step in which the ADs that are involved in the service federation establish a peer-to-peer interconnectivity or register to a central entity. The registration step defines the type of federation, which can be open or closed. As an open service federation can be considered when external new domains are more easily establish the interconnectivity. The closed federation includes pre-defined participants with strict policies and rules that are set and defined by the ADs.
- **Discovery:** in this step the involved ADs exchange information on their computing and capabilities to provide services or resources. Each AD holds and periodically updates a global view of the available services at the external ADs.
- **Announcement:** the consumer domain initiates this step once it has been decided the need to federate a service (or part of a service) in an external domain. An announcement is broadcasted to all the potential provider ADs. The announcement is composed of the requirements for a given services.
- **Negotiation:** all the potential provider ADs receive the announced offer and sends back an answer including the pricing of the service.

- **Acceptance and deployment:** The consumer AD collects all the responses from the potential provider ADs and selects a single offer that is most suitable for him. The selection process is entirely dependent on the consumer AD internal policies and preferences. The consumer AD sends back an acceptance reply and starts the deployment of the requested federated service.
- **Usage & Charging:** once the service is deployed in the providers domain, the provider notifies the consumer AD and sends all the necessary information for the consumer AD to include the federated service as part of its end-to-end service chain. From that moment, the provider AD starts charging the federated service during its lifecycle, until it is terminated.

We would like to stress out that security, privacy, and trust among the participating ADs in the service federation is crucial in all the previous steps. Due to competitive reasons, different ADs would not reveal much information regarding the underlying infrastructure or the full capabilities for service deployment.

Applying DLT for Federation

The sequential execution of the service federation steps can take from more than a minute to over an hour depending on how they are implemented. In a fog environment that is dynamic and heterogenous, the underlying infrastructure of each AD is continuously changing, and the state of a resource can change in order of seconds. To improve the federation process in a secure manner, the BASS through the *external federation support* element offers the service federation process to run over DLT. More specifically, the federation procedures to be executed on a Federation smart-contract (SC) which is running on top of a permissioned blockchain. The focus of the SC design is to maintain neutrality and privacy while overseeing the federation procedures that involve all ADs.

Each AD sets up a single node as part of the peer-to-peer blockchain network. The distributed nature of blockchain allows scalability while maintaining the security. The ADs communicate with the Federation SC through transactions and every transaction is recorded in the blocks. The generation of blocks depends on the consensus protocol. The choice of the consensus protocol would determine the speed and the security level of the federation process. For example, the Proof-of-Authority consensus increases the speed, while the Proof-of-Work mechanism increases the security of the blockchain.

Each AD that wants to join establishes connectivity with at least a single node in the blockchain network using a new and locally deployed node. It registers to the Federation SC with a single transaction using its unique blockchain address. In the registration transaction the Federation SC records the relevant information of the registering AD and its service footprint. This process is equivalent to the **registration step** explained before and it is relatively simple to be realized. Once the registration is completed, the AD is ready to consume or provide federated services.

Figure 7-1 shows the interactions of the consumer and provider domains Orchestrators with the BASS Federation SC for a single service federation process. When a consumer AD needs a federated service, it creates a federation announcement (step 1). The **announcement** is sent as a transaction to the Federation SC which records the announcement as a new auction process on the blockchain (step 2). Then, the Federation SC broadcasts the announcement to all registered ADs (step 3). Please note that the **discovery** step is omitted in the design of the Federation SC because the privacy and security of the

ADs are protected by hiding their address in the broadcast announcement. Once the broadcasted announcement is received, the potential providers analyse the requirements and place a bid offer to the Federation SC (step 4 & 5). Each offer is recorder by the Federation SC (step 6). In our vision the consumer domain oversees the **negotiation** and **acceptance** steps. In that way, the consumer AD has full control and freedom to apply any selection policies. Consequently, the consumer AD is notified for any new bidding offer and polls the Federation SC to obtain information of each bidding offer (step 7,8 & 9). Once the consumer AD selects a winning provider AD, it closes the auction in the Federation SC (step 10 & 11). The winning provider is recorded in the Federation SC and a message is broadcasted to all the participating ADs that the auction has finished, and a winner is chosen (step 12 & 13). Each of the participating ADs attempts to find out if he is the chosen winner in order to deploy the service. As shown in Figure 7-1 only the winning provider AD is granted access to the information (step 14 & 15). At this point the negotiation and acceptance steps are completed and the **deployment** of the federated service has started (step 16). Once the deployment is finished, the provider AD confirms the operation by sending transaction to the Federation SC (step 17). The Federation SC records the successful deployment and starts **charging** for the federated service (step 18). Finally, the Federation SC notifies the consumer AD of successful federated service deployment (step 19 & 20) so the consumer AD can start **using** it.

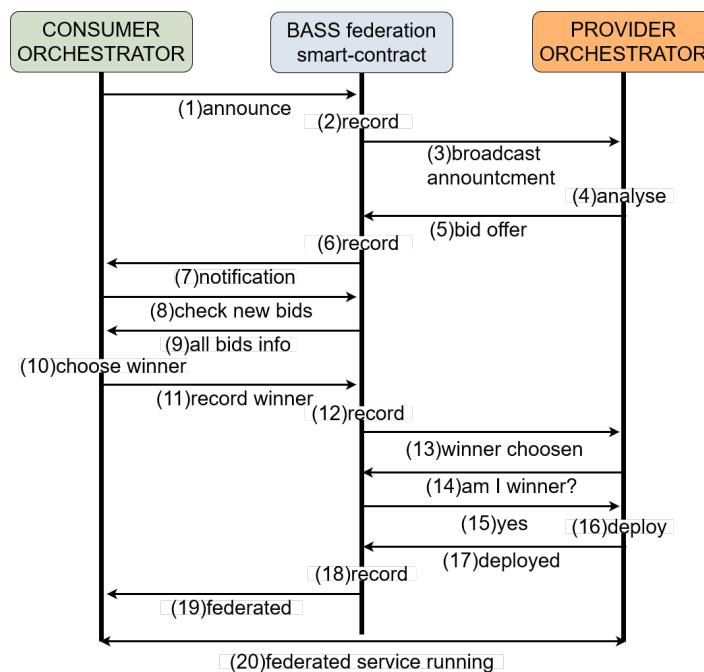


FIGURE 7-1 SEQUENCE MESSAGE DIAGRAM FOR BASS FEDERATION SMART-CONTRACT AND ADMINISTRATIVE DOMAINS DURING FEDERATION

Experimental Validation

To prove the feasibility of the DLT service federation concept we deployed a trusty and untrusty experimental scenario where we performed federation over an Edge robotics use case. The presented DLT-based federation can be useful in Edge robotics scenarios [47] here highly mobile robots demand frequent change of point service in the access network which is currently feasible within single AD.

Often, and Edge robotics service require fast and short-lasting expansion of the access point service footprint over multiple administrative domains.

The consumer AD infrastructure in our testbed consists of a host that runs LXD virtualization on top. The host is orchestrated by the consumer orchestrator which is a simple custom developed orchestrator that uses fog05 as distributed Virtual Infrastructure Manager (VIM) to deploy virtual Access Points (vAPs). The provider AD is isolated from the customer domain. Contains a single host and a Provider orchestrator orchestrates the virtualized LXD infrastructure through a new isolated instance of fog05. The BASS external federation support is implemented as two instances of Ethereum blockchain. The instances are deployed over virtual machine on a server. Both instances contain the Federation SC described before. The first instance is running Proof-of Authority (PoA) consensus for trusty communication, and the second instance Proof-of-Work (PoW) for untrusty communication.

The experimental scenario is mimicking a real use-case where mobile robot is instructed to deliver goods in an area. In order to finalize the task, the robot needs to drive from the consumer domain covered area to the area of coverage of the provider domain. Based on the real-time robot location the consumer orchestrator knows when the robot is about to leave the coverage area and triggers the federation procedure. After the triggering, the consumer orchestrator proceeds with the federation procedure as described. The provider domain is selected as winner, establishes an overlay inter-domain link to the consumer domain and deploys the federated vAP. After the deployment of the federated vAP has finished, the provider orchestrator confirms the deployment to the Federation SC by storing the BSSID of the deployed AP. The consumer orchestrator will use this information to perform handover to the federated AP.

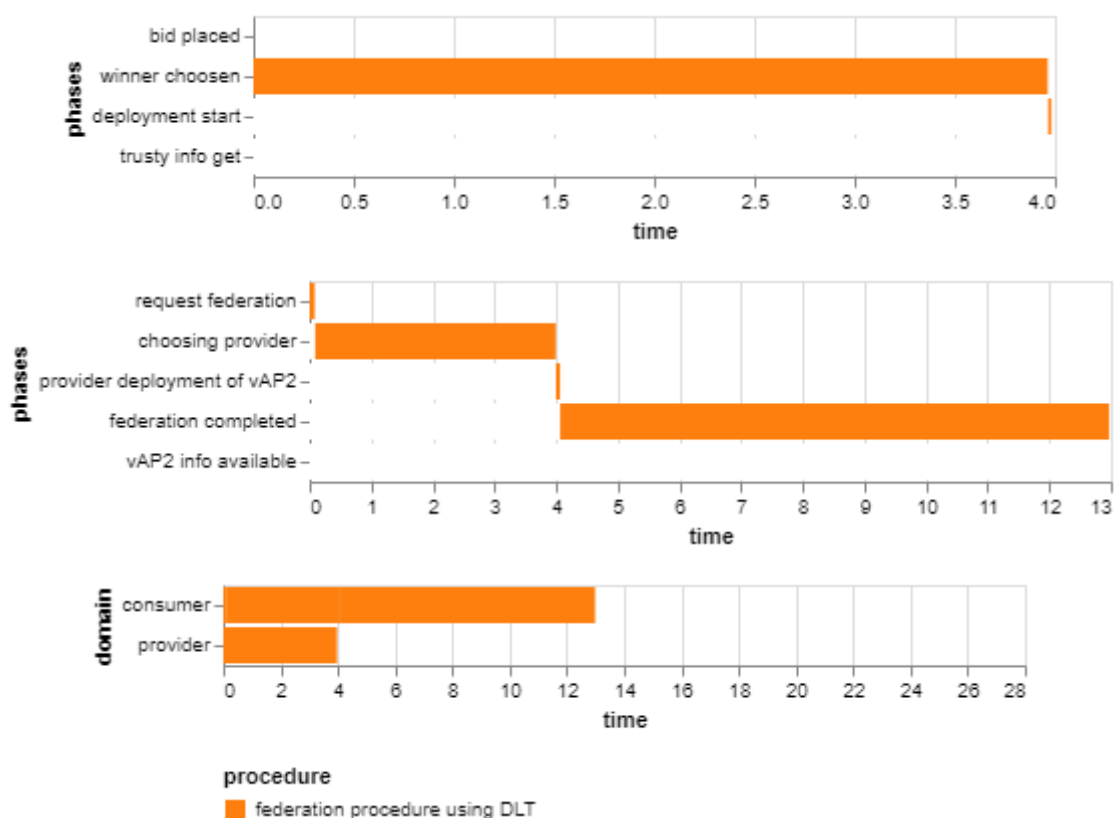


FIGURE 7-2: FEDERATION USING POA CONSENSUS: (TOP) SUMMARIZED PHASE; (MIDDLE) CONSUMER AD; (BOTTOM) PROVIDED AD; [48]

We evaluated the time performance of the Edge robotics federation for each of the PoA-based and PoW-based scenarios. To that end the bottom graph on Figure 7-2 presents the accumulated times for the federation procedures in both consumer and provider domain using PoA consensus. The average federation time is 12.97 seconds for the consumer domain and 3.98 seconds for the provider domain. Figure 7-2 in the middle breaks down all the phases in the consumer domain that occur within the previously mentioned 12.97 seconds and are needed for the consumer domain to retrieve the BSSID of the federated vAP in the provider domain.

Figure 7-2 on the top breaks down all the phases in the provider domain where we can see that the negotiation and bidding process until the provider domain is elected as winning provider takes 3.98 seconds. More specifically, it takes 3.98 seconds from the time the provider receives the broadcast announcement until the deployment is ready to start.

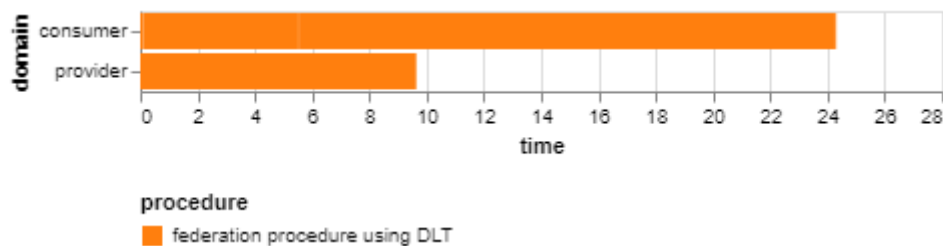


FIGURE 7-3: FEDERATION USING POW CONSENSUS: SUMMARIZED TIMES [48]

The results of the PoW-based scenario and untrusted communication are shown in Figure 7-3. The graph shows only the accumulated times for both domains. Compared to the PoA-based solution, the PoW-based solution takes significantly more time to negotiate and complete the federation process using the DLT. Due to the PoW consensus mechanism the federation completed phase is completed within 24.3 seconds, nearly double the time of the PoA-based solution.

7.2. BASS and OCS Integration Workflow

Figure 7-4 presents the BASS and OCS integration workflow steps. In the first step the Vertical service Coordinator (a.k.a BASS Controller) trigger the functionality of creating a vertical service on the Fog05Driver that was developed for this purpose. Then the Fog05Driver is in charge of translating the generic Vertical service descriptor file to a specific Fog05 descriptor that is called FDU (Fog05 Deployment Unit) descriptor. If it's necessary a specific Fog05DriveConfig class contains configuration options for the Fog05 Region/Driver. One example would be connection details, credentials, etc. Then the Fog05Driver issues a "on-board" POST request to the fog05-rest-server by passing in the body the FDU in json format. The fog05-rest-server will then validate it and on-board the FDU in Fog05 server. The response is an OK message and the populated FDU including the UUID of the newly created descriptor. Step two refers the instantiation of the onboarded FDU, this is achieved by sending an "instantiate" POST request to the fog05-rest-server including the created UUID. The response is a descriptor's instance UUID and a OK status.

Step three of the workflow refers to triggering the "delete" vertical service, the Fog05Driver will send a DEL request to the fog05-rest-server by passing the UUID of the FDU descriptor's instance. The response will be an OK status. Finally, step four of the workflow deals with removing "off-loading" the vertical service descriptor from the Fog05 Server. The Fog05Driver will send a DEL request to the fog05-rest-server by passing the UUID of the FDU descriptor. The response will be an OK status.

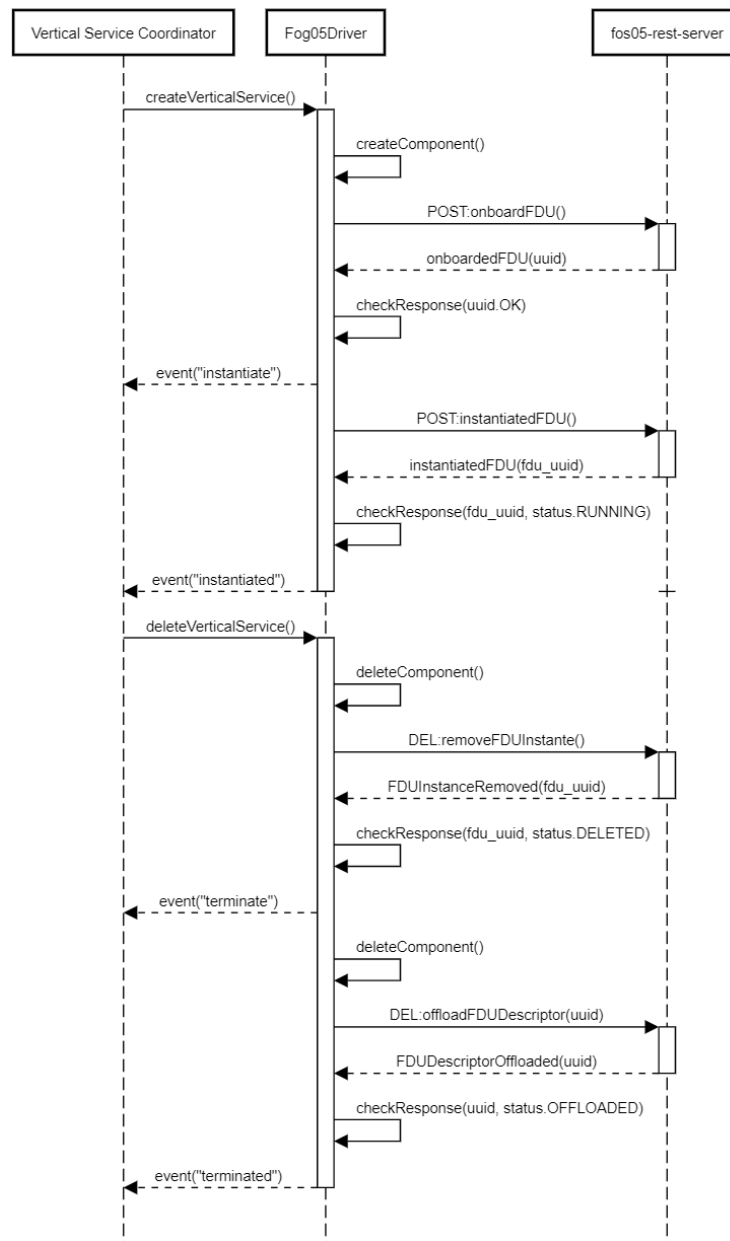


FIGURE 7-4 BASS AND OCS INTEGRATION WORKFLOW

7.3. Data driven RAN Intelligence

The 5G-DIVE project relies on Edge computing resources to assist the different use cases in improving their performance on different aspects, the majority of improvements being use case specific, i.e., at the application layer. However, an Edge computing fabric or, in the case of Open-Radio Access Network (O-RAN) standards, an O-RAN Radio Intelligent Controller (RIC), can also serve the 5G network to provide improvements at the Network Layer. The ways to achieve this are to either deliver an intelligent engine to an edge fabric node or to a RIC node. The intelligent engine will run as an Over-the-Top application (OTT), and will be able to improve network layer functions.

The ZDM use case relies on a video stream to detect defective objects. Preliminary tests on the 5G connectivity have shown that a commercial 5G deployment may have difficulties sustaining a continuously reliable video stream. This is an important finding that becomes even more relevant if the required stream quality for detection is a higher resolution setting than the currently used camera, that is HD, such as 2K, 4K or 8K (keeping sustained quality). It becomes therefore important for the ZDM use case to address throughput improvement strategies in order to improve the QoE of the video being streamed. This is done in the following couple of sections. Next section provides an introduction to the O-RAN architecture and the current 3GPP standardization work for 3GPP networks Edge access. The following section describes the concept of Intelligent engine that will be used in the ZDM use case to achieve higher available throughput.

7.3.1. O-RAN architecture

Some of the latest mobile network developments of the past years have included work on aspects around an open and virtualized RAN. In virtue of control and user plane separation (CUPS) in 5G service-based architecture (SBA), the functions that derive policies can be located apart from main 5G core network functions, opening the door for more virtualized functionalities.

The O-RAN Alliance has been driving the standardization efforts to achieve this vision. The concept of an open RAN translates into an open hardware and cloud platform, that telecom manufacturers, suppliers and operators can use to deploy their networks. The goal of such open platforms is to reduce the current number of proprietary product architectures and vendor specific Operations and Management (O&M), with the goal of increasing efficiency of both deployments and operations. To deploy and operate on open platforms, virtualization of network functions is a key aspect.

Besides the increased efficiency of both deployments and operations, virtualized network deployments in open platforms provide an easier infrastructure for embedded AI-enabled RAN control. Figure 7-5 Open RAN architecture depicts the basic architecture for Open RAN.

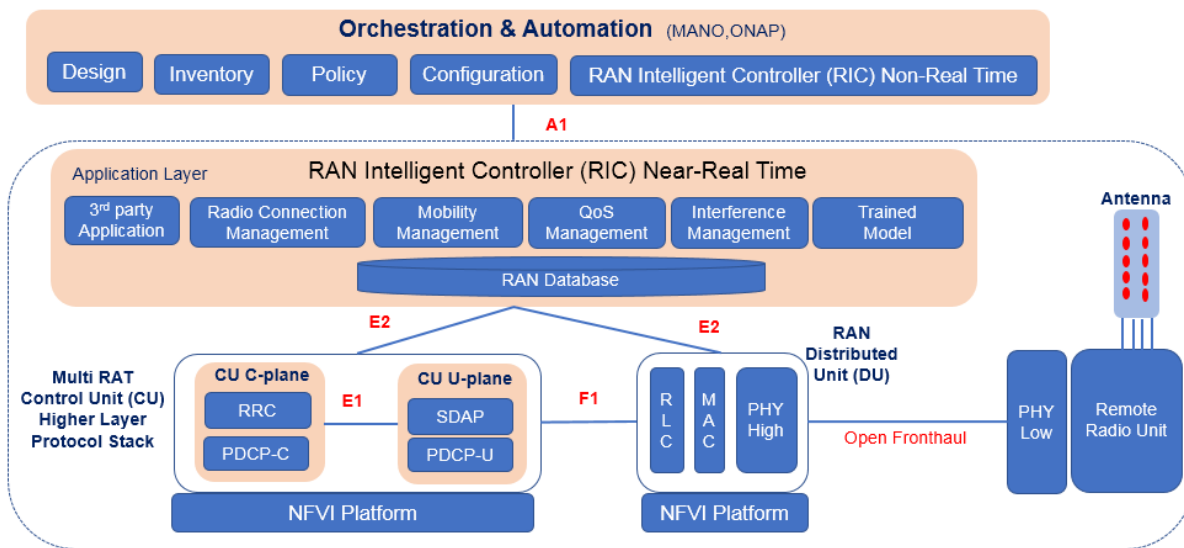


FIGURE 7-5 OPEN RAN ARCHITECTURE

Besides the separation of control and user planes by introducing new interfaces and functional base station components, the figure depicts two RAN Intelligent Controllers (RICs), one for near-real time, and another for non-real time intelligence. As depicted as well in the figure, there are a number of RAN functions that are controlled at the RICs. It is important to note that, because these functions pertain to the application layer, whether they relate to mobility, QoS, interference management, or any other function, they are OTT. As the architecture is compliant and complementary to 3GPP (and other bodies) standards, the RIC can be seen as a deployment node for any kind of RAN intelligence, especially via AI/ML. Key enablers for data driven intelligence are databases that keep storing useful telemetry data to serve a specific intelligent application purpose.

7.3.2. 3GPP Edge fabric standardization efforts

RAN deployments that follow a pure 3GPP architecture can also have many of their functions optimized via Edge nodes. Current standardization efforts in 3GPP include works from almost all Service and System Aspects (SA) working groups, namely from SA2-SA6. SA2 is covering core network enhancements. A mapping between the 3GPP CN architecture including the enhancements specified in SA2 and the 5G-DIVES solution was presented in D1.3 [zz]. SA3 is covering security aspects while SA4 conducts works on media processing, and SA5 is responsible for general management aspects.

In SA6, the working group responsible for the application layer architecture, normative specification work has been initiated for enabling Edge Applications. The objective of this work is to define an enabling layer to facilitate communication between the Application Clients (AC) running on the UE and the Edge Application Servers (EAS) deployed on the Edge Data Network.

The support of interworking between the Edge fabric and 3GPP networks is therefore a very active effort, and this effort will pave the way towards a global adoption of Edge Computing fabric and pervasive deployments of Edge Networks that can serve both the end consumers and industry verticals.

Both the RICs defined in the O-RAN architecture and the Edge fabric supporting pure 3GPP networks can therefore be seen as hosts for OTT applications that can control and improve a number of aspects and functions in the RAN.

Particularly in the case of RIC deployments, intelligent control applications are named xApps or rApps, depending on whether they are deployed at the near-real time RIC or non-real time RIC. xApps can be deployed at the Edge in private premises or environments. The benefit of deploying xApps in private networks is twofold. Firstly, the telemetry data is stored within the private network, not leaving a public domain, enhancing security aspects. Secondly, the latency associated with an Edge node or RIC controller deployed in-premise is necessarily lower.

One example of an OTT application that can be deployed at any RIC or at the Edge fabric is for Access Traffic Steering, Splitting and Switching (ATSSS), and its conceptual functionality is detailed in the next section.

7.3.3. ATSSS xApp

An ATSSS xApp can be utilized in access traffic steering decisioning. If such an xApp is deployed at the edge, the latency associated with pushing access traffic steering rules is lower when compared to having SMF/PCF inside the mobile network operator. An AI/ML model trained with the available telemetry at the host node can be incorporated in the traffic steering decisioning. This can be considered as an enhancement to 3GPP Rel-17 ATSSS framework which provides flexibility to both the UE and the UPF on the traffic splitting control over 3GPP and non-3GPP access networks in order to maximize the bandwidth/throughput. It is worth to note that in Rel-17 ATSSS framework, the link performance measurements provided by Performance Measurement Function (PMF) is used. However, in the proposed ATSSS xApp, in addition to link performance measurements, access network telemetry can be used. Therefore, any AI/ML-based prediction on the access link status, user mobility, gNB/AP load status can be used to enhance the access traffic steering decisioning.

The main motivation of the considered ATSSS xApp is to react to sudden/predicted changes on the link and/or network status in order to efficiently use 3GPP and non-3GPP access networks. The ATSSS xApp gathers access network telemetry including 3GPP and non-3GPP accesses, link performance measurements from PMF, and modifies the access traffic steering rules, in other words access traffic weight factors for 3GPP and non-3GPP access networks. As an example, a steering rule with access traffic weight factors of 30% onto 3GPP and 70% onto non-3GPP can be set by the network operator. When the 3GPP access network gets congested, assigning 30% of the ongoing traffic for a UE may not achieve the throughput requirements. In this case, ATSSS xApp makes use of RAN telemetry to understand/predict load status of 3GPP and non-3GPP access networks and modifies the weight factors i.e., 10% onto 3GPP and 90% onto non-3GPP to maximize the achievable throughput.