IoTwins
BIG DATA OPTIMIZE INDUSTRY AND SERVICES

Grant Agreement N°857191

Distributed Digital Twins for industrial SMEs: a big-data platform

DELIVERABLE 2.3 - ARCHITECTURE, CORE DATA AND VALUE-ADDED SERVICES BUNDLES SPECIFICATIONS (II)

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement № 857191

Document Identification

Project	IoTwins
Project Full Title	Distributed Digital Twins for industrial SMEs: a big-data platform
Project Number	857191
Starting Date	September 1st, 2019
Duration	3 years
H2020 Programme	H2020-EU.2.1.1. - INDUSTRIAL LEADERSHIP - Leadership in enabling and industrial technologies - Information and Communication Technologies (ICT)
Topic	ICT-11-2018-2019 - HPC and Big Data enabled Large-scale Test-beds and Applications
Call for proposal	H2020-ICT-2018-3
Type of Action	IA-Innovation Action
Website	iotwins.eu
Work Package	WP2 - IoT-Edge-Cloud infrastructure and big data services for SMEs
WP Leader	FOKUS
Responsible Partner	ESI
Contributing Partner(s)	FOKUS, BSC, INFN, SAG, SAGOE, THALES, UNIBO
Author(s)	C. Ahouangonou (ESI)
Contributor(s)	C. Garcia (BSC), D. Cesini (INFN), S. Rossi Tisbeni (INFN), T. Preclik (SAG), T. Schüle (SAG), S. Busch (SAG), F. Kintzler (SAGOE), V. Thouvenot (THALES), K. Kapusta (THALES), G. Di Modica (UNIBO), A.Willner (TUB), C. David (ESI)
Reviewer(s)	C. Arlandini (CINECA), D. Nehls (FOKUS)
File Name	D2.3 – Architecture, Core Data and Value-Added Services Bundles Specifications (II)
Contractual delivery date	M22 – 30 June 2021
Actual delivery date	M27 – 03 November 2021
Version	1.1
Status	Final
Type	R: Document, report
Dissemination level	PU: Public
Contact details of the coordinator	Francesco Millo, francesco.millo@bonfiglioli.com

Document log

Version	Date	Description of change
V0.1	25/05/2021	Initial Document structure.
V0.2	10/06/2021	Added implementation details to Section 4.3.1 (TU Berlin / Fraunhofer FOKUS).
V0.3	22/06/2021	Merge of contributions made in different versions of the file. Accept the comments and tracked modifications.
V0.4	28/07/2021	Fix content after the first review, resolve comments and accept tracked modifications.
V0.5	07/08/2021	Additional first review feedbacks implementation and document formatting.
V1.0	04/10/2021	Pre-final version sent to the consortium for a final check.
V1.1	03/11/2021	Final version.

Executive summary

This deliverable reports on the actions, considerations and design decisions that have been taken so far, with regards to the IoTwins architecture and related data services. From a comprehensive set of interviews conducted with the Testbeds, by WP2 and WP3 technical partners, it appears that the IoT and edge implementations are already mature but significant efforts are still needed by the more complex implementation at the cloud level. Moreover, some testbeds requested support for their machine learning deployments, which are mainly based on pytorch, and tensorflow and keras for the WP3.

To provide simulation services, that may run at the cloud level, the reference architecture has been updated (for instance see the Testbed 5). The modification we brought to the platform opened access to new developers. Such modification consists of introduction of new components such as, e.g., License management, Private Repository and Simulation functional blocks.

Passing through the platform API's high-level definitions, involving data management services and orchestrators, this deliverable also tackles the overview instantiation and implementation details with respect to the IoTwins data services, that will be implemented by the partners, relying on state-of-the art and broadly-known software, such as *MongoDB* or *InfluxDB* for storage or *Keycloack* for security.

Last, this deliverable also provides practical examples of the deployment of a simulation service.

Table of Contents

Executive summary 4
1 Introduction 6
2 References and Abbreviations 6
2.1 References 6
2.2 Abbreviations 7
3 Feedback from testbeds 7
3.1 Testbed Description 8
3.2 Interview Feedbacks 9
3.2.1 Testbed 1 10
3.2.2 Testbed 2 11
3.2.3 Testbed 3 11
3.2.4 Testbed 4 12
3.2.5 Testbed 5 12
3.2.6 Testbed 6 13
3.2.7 Testbed 7 14
3.2.8 Table AS_IS vs TO_BE 15
4 IoTwins platform enhancement 20
4.1 Revised reference architecture 20
4.2 Platform API 20
4.2.1 Orchestrator INDIGO 21
4.2.2 Orchestrator Kubernetes 25
4.2.3 S3 storage 30
4.3 Platform-Level Data Services 31
4.3.1 Implementation Details 32
4.4 Private Docker repository (Harbor) 36
5 Conclusion 37

1 Introduction

This document extends the deliverable D2.2 “Implementation of the IoTwins platform (I)”. Its purpose is to supplement and refine D2.2 according to the recommendations from the individual meetings organized with each testbed. These meetings took place between WP2 and WP3 partners on one hand, and the partners of each testbed on the other. The objective of these meetings was to carry out the technical review of the testbed to better understand its needs and to study the approaches facilitating the handling by the testbed of the elements of the platform. Ways of improvement were also explored together. Finally, during the meetings we identified the support needs of the testbed, assigned the necessary WP2-WP3 resources accordingly, and implemented the feedback into the platform.

In the following chapters, we will successively present:

- Feedback from the various meetings organized with each testbed.
- A table presenting the synthesis of the components.
- Improvements made to the architecture such as functional blocks relating to simulation as well as optimization.
- Changes in the architecture of the IoTwins platform. Several aspects will be discussed and illustrated; particular care has been taken to describe the data as well as services attached to them.
- A chapter dedicated to the specification of APIs.
- A few examples illustrating some usage of the platform capabilities.

2 References and Abbreviations

2.1 References

D2.1 Architecture, APIs, Specifications and Technical and User Requirements, March 2020

D2.2 Implementation of the IoTwins platform (I), v07, November 2020

D3.1 Requirements from Large-scale Industrial Production and Facility Digital Twins, version 3, February 2020

D3.1 First version of the simulation and AI services for digital twins, version 2.1, September 2020

D4.1 Enhanced data collection and integration for the manufacturing testbeds, version 4, August 2020

D4.2 First Digital Twin Version Delivery for the Manufacturing Testbeds (WP4), version 1, July 2021

D4.3 Feedback On First Version of Digital Twins to Technical WPs (WP4), version 1, July 2021

D5.1 Enhanced data collection and integration for facility management test-beds, November 2021

D5.2	First Digital Twin version delivery for the facility management test-beds, version 1, May 2021
D5.3	Feedback on first version of digital twins to technical WPs (WP5), version 1, July 2021

2.2 Abbreviations

AMQP	Advanced Message Queuing Protocol | AMQP
MQTT	Message Queuing Telemetry Transport | MQTT
ELK-stack	Elasticsearch, Logstash, Kibana | Elastic
HPC	High Performance Computing | Wikipedia
IaaS	Infrastructure as a Service | NIST SP 800-145
PaaS	Platform as a Service | NIST SP 800-145
IAM	Identity and Access Management | IBM
ML	Machine Learning | Wikipedia
REST	REpresentational State Transfer | Wikipedia
YAML	Yet Another Markup Language | Wikipedia

3 Feedback from testbeds

At first, we proceeded to elaborate the architecture and to identify tools and components applicable to the implementations of the various testbeds. The results of this activity were reported in the deliverable D2.2. Based on this deliverable, one-to-one interviews have been conducted between the WP2 and WP3 teams on one side, and each of the testbeds team on the other side.

The collaboration process started with the building of a template document made by the WP2 and WP3 partners. Then, we asked each testbed member to fill this template in order to stimulate testbed owners to provide a more detailed specification of the testbed itself and to identify the platform's missing features. Finally, the interviews were made as follows:

1. A one-to-one interview was organized with each testbed;
2. Each meeting was led by representatives from technical work packages (WP2/WP3) and from each Testbed owner. The prepared document was used as support to guide the interview, to pose questions and to keep track of the answers;
3. Answers were elaborated and structured.

The following paragraphs describe, first, the testbeds involved in the interview cycle and then present the feedbacks of these interviews.

3.1 Testbed Description

The testbeds of manufacturing work package (WP4) and the Facility Management Sector (WP5) are the ones involved in this first round.

The following table (extracted from D2.2) lists the different testbeds involved in the current implementation.

Testbed	Description
Manufacturing Testbeds	
TB1: Wind Turbine Predictive Maintenance	The testbed is aimed at creating a digital twin of a wind farm by aggregating simulation and Machine Learning (ML) models of single turbines for predictive maintenance.
TB2: Machine Tool Spindle Predictive Behaviour	The testbed is aimed at creating multiple target-oriented digital twins of machine tools for the production of automotive components.
TB3: Predictive Maintenance for a Crankshaft Manufacturing System	In this testbed data is collected from a high throughput crankshaft manufacturing system to enable e.g. predictive maintenance for the production line.
TB4: Predictive Maintenance and Production Optimization for Closure Manufacturing	The objective of this testbed is to establish an overall management optimization including specific goals in the area of predictive maintenance.
Facility Management Sector	
TB5: Sport Facility Management and Maintenance	This pilot focuses on the management of facilities involving the flow of large crowds both during normal operation and during maintenance and upgrade phases involving construction.
TB6: Holistic Supercomputer Facility Management	The testbed focuses on data-driven prescriptive maintenance and optimization of large computing facilities.
TB7: Smart Grid Facility Management for Power Quality Monitoring	This testbed focuses on smart grid KPI computation as close to the data sources as possible, with input of higher-levels info, that cannot be accessed locally.
Replicability	
TB8: Patterns for Smart Manufacturing for SMEs	This use case is devoted to the definition of a general and replicable methodology for SMEs based on the convergence of data analytics, AI, IoT, and physics-based simulation
TB9: Standardization/Homogenization of Manufacturing Performance	The main objective of this use case is the extension of the models investigated and assessed in testbed N.6 to a wider series of machinery and other plants around the EU.
TB10: Examon Replication to INFN/BSC Datacentres	The Examon IoTwins-enabled management of testbed N.6 will be replicated on two datacentres of INFN and BSC. The purpose of this use case is to define a methodology for the monitoring infrastructure reuse and deployment in new and different contexts.
TB11: Replicability towards Smaller Scale Sport Facilities	This use case will demonstrate the replicability and scalability of testbed N.5 at other FCB sport facilities.

Testbed	Description
Business Oriented	
TB12: Innovative Business Models for IoTwins PaaS in Manufacturing	The testbed aims to validate innovative business-models that bring resources available in the cloud accessible to applications running on manufacturing machines related to the machine monitoring business.

3.2 Interview Feedbacks

The current section aims at reporting on the actions listed from the different Testbed meetings organized between the WP2 and 3. These interviews took place in November-December 2020.

During the user requirements collection (see D2.1) period we identified that testbeds were equipped at IOT level and were looking support more at edge and cloud levels. The following diagrams (described in the deliverable D2.2) were used to support the discussions with the Testbed partners. Proposed Components for implementing Edge and Cloud levels are as illustrated:

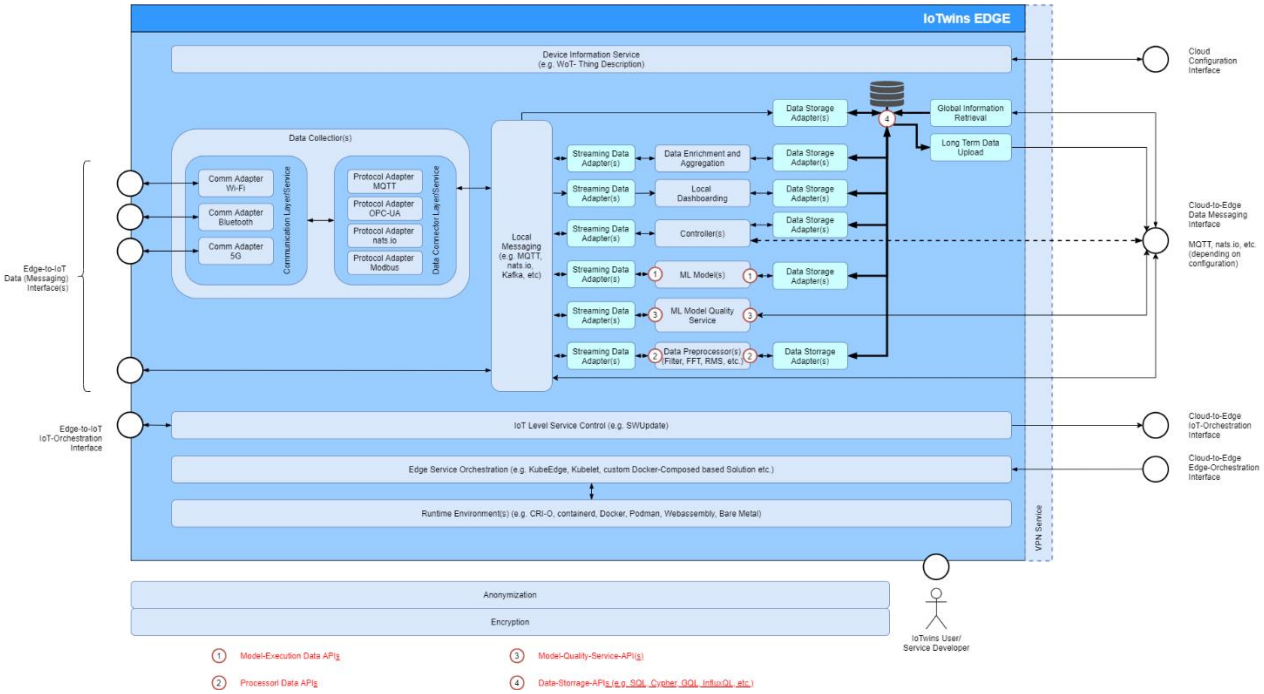

Figure 1: IoTwins Architecture Implementation Edge Level

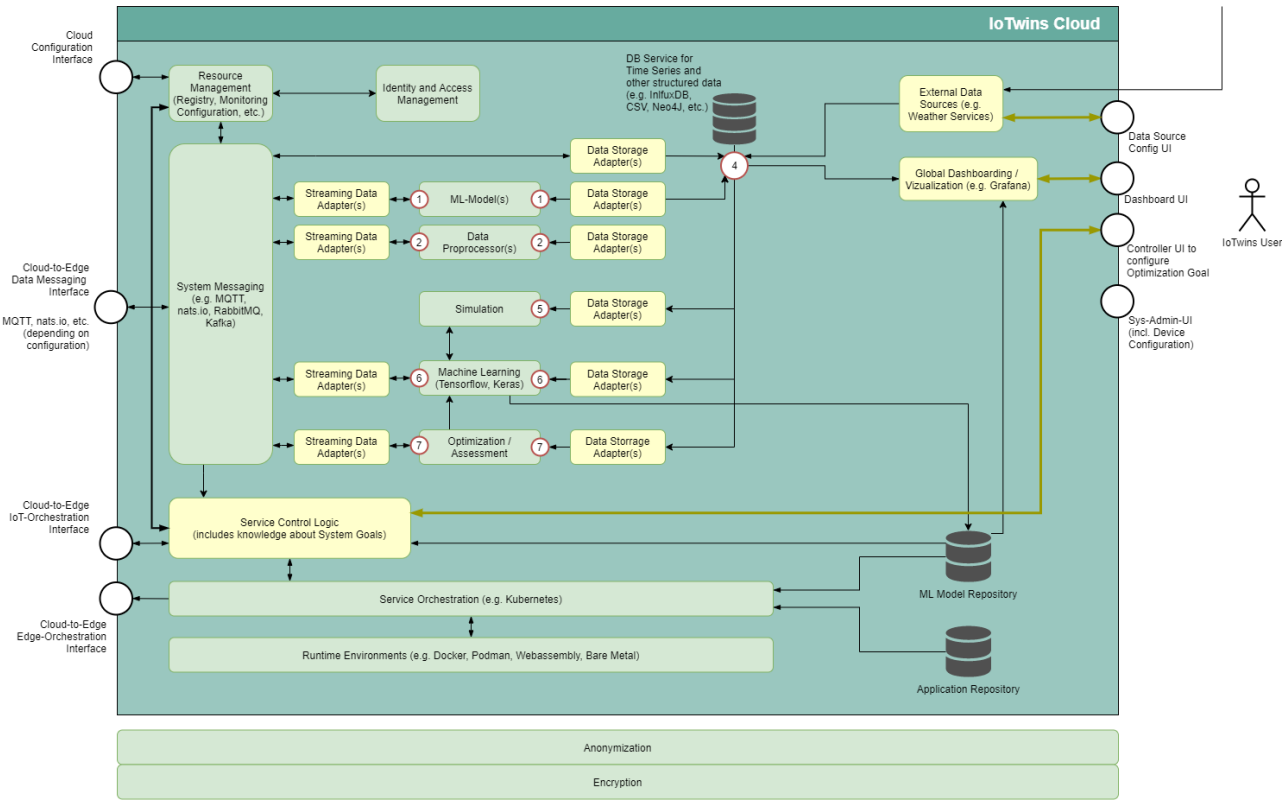

Figure 2: IoTwins Architecture Implementation Cloud Level

3.2.1 Testbed 1

The IoT level and Edge level are well defined today, and implementations are ongoing. Data is sent to the edge machine which, after some pre-processing, sends the data to the cloud. In the current prototype, HTTP protocol is used. According to the constraints of the selected IoTwins cloud described in the D2.2, the testbed 1 is willing to adapt to the protocol required by the cloud. After discussion it is agreed that HTTP will be supported.

Testbed 1 intends to use services provided by three different infrastructures: storage from INFN provider, cloud for storage, heavy computations from ESI and some private front-end software from on-premise machines.

Table 1: TB1 action list

#	Description	Who
1	INFN will investigate how to make the 3 cloud collaborate	BRI / INFN / ESI
2	Offer the possibility to use InfluxDB on cloud	INFN

3.2.2 Testbed 2

The layers of IoT and edge are well managed today and implemented. Historical data is already collected and stored on the edge machines using InfluxDB. Meta data and analysis data are stored in a MongoDB database. All of these are located on the Edge machine.

The following technologies are in use:

- Dot net Core,
- Analysis tools in Python,
- Microservices
- Data Brokers (RabbitMQ)

The place where people from TB2 are looking more on the cloud side and IoT-to-cloud side. This may imply orchestrating cloud and edge.

Table 2: TB2 action list

#	Description	Who
1	Evaluate the best way to send the data to the cloud. As an example, it could be an HTTPS request executed every n seconds.	FILL INFN
2	Evaluate the best way for implementing the data base support at the cloud side (Service or local implementations).	FILL INFN
3	Show how to use hyperparameter fine-tuning service (already developed and capable of handling PyTorch models). Understand how to use the services provided by WP3 and do a complete deployment of WP3 services in INFN infrastructure, possibly with a graphical interface.	FILL, WP3-TBA
4	Analyse the managements of simulations; within Docker, with a remote solver management using an HPC configuration and handling license management	FILL INFN

3.2.3 Testbed 3

The testbed expectation is to become mainly cloud-based. Concretely, there is no need for strict real-time stream data analysis to be done by IoTwins, as a health index is calculated on a per second basis and is expected to be shipped to the cloud for further processing. Moreover, the testbed is interested in shipping the raw or semi-processed data while being produced, via MQTT, following a small-or-medium-sized batching fashion. This is because the TB wants to relieve devices from keeping data. Thus, the forecasting is expected

to be done (not necessarily on strict real-time) on Edge devices, which may hold and update the ML model, but the data should be kept in cloud, in order to keep track on the evolution of specific variables.

Currently, only one machine ships data to the cloud, although this will likely change in the future.

Table 3: TB3 action list

#	Description	Who
1	Obtain sample of data from TB03 to have a look at the data and identify best course of action	WP3-TBA
2	Understand if and how validation data can be obtained (simulation with test bench?). Validating data will need to evaluate the performance of ML models.	WP3- TBA, EXCE

3.2.4 Testbed 4

The GCL testbed covers the production of closures. In the current implementations, the IoT and Edge layers are covered. A tentative of implementation for the cloud is ongoing with the support of CINECA.

Edge machines are implemented as virtual machine (VM Ware) deployed on a server. Data is stored in an InfluxDB database. For IoT and edge, applications from PTC are used to perform the operations

The current testbed needs direct support to implement the ML analysis. The goal of the ML analysis is, at this stage, predictive maintenance, that is 1) detecting failure and wrong behaviour of components and 2) predict such failures.

Table 4: TB4 action list

#	Description	Who
1	Continue preliminary analysis on batches of data (currently being performed)	UNIBO
2	Review of the architecture implementation proposed by CINECA Evaluate its compliance to the IoTwins proposed architecture	GCL / CINECA / INFN
3	Explore the IAM proposed in the IoTwins platform	GCL

3.2.5 Testbed 5

This testbed collects data from many sources and is willing to improve by adding new smart devices, such as GPU-integrated camera devices. It plans to use also external data sources (weather, traffic, social media, ...).

Currently there is no available intermediate machine between the IoT and the HPC/cloud. They are planning to set up one (not necessarily on premise), although it might be not strictly necessary. Most of the data is currently obtained via an API from a centralized FCB production machine that offers data to external users, intended to enable early-forecasting but not strict real-time streaming. However, WiFi data is gathered via an ARUBA VM instance, but its data access mechanisms are still not fully defined, and its handling might require further dedicated efforts, as ARUBA is expected to ship up to 800 million reading json documents per day.

Extra difficulties arise, as the testbed is based on people movement tracking, and due to COVID-19 pandemic the visits and matches held in the stadium are greatly reduced or directly suspended.

Table 5: TB5 action list

#	Description	Who
1	Provide WIFI early or real-time data reception. If not possible, offer a way in which to obtain data in batches.	FCB
2	Take final decision regarding Edge machine	FCB

3.2.6 Testbed 6

The testbed is based on the Examon infrastructure, which has been deployed on several CINECA's machines in the last few years. Examon has been functioning without issues in production machines; changing Examon components to accommodate IoTwins platform will be challenging, but luckily there are analogies and connections between the IoTwins and Examon architectures that may be exploited. To demonstrate the integration of Examon within the IoTwins platform, TB06 cloud layer will be improved following IoTwins platform guidelines (e.g., creating pipelines and services to develop and deploy AI models via Spark).

This testbed collects data from the IoT (the computing nodes of the supercomputer) using software modules, gathering data from a variety of physical sensors and service monitors. The data is directly sent to the CINECA cloud, as the IoT layer (the computing nodes) is directly connected to the cloud platform through dedicated interconnection; there is no need for Edge functionalities and services. Data is sent to a central broker which acts as middleware through MQTT protocol. The central broker (actually a couple of brokers for redundancy and load balancing) is situated on the CINECA cloud infrastructure. Along the MQTT broker, there are the databases used to store data: 1) KairosDB for handling time series data and 2) Cassandra for long-term storage.

On the cloud level, AI models are being developed and (tentatively) deployed, for predictive maintenance and optimization tasks. Currently, the models run on bare metal, in the form of Jupyter notebooks and python scripts – as it is the most efficient way to exploit the computational capabilities of a supercomputer. However, improvements are possible, especially in the context of deploying services (and not simply running python scripts) on the cloud – TB06 could benefit from assistance and support from WP2 & WP3 in this area.

Table 6: TB6 action list

#	Description	Who
1	Continue developing AI-based services, tailored to TB-specific needs	CINECA, UNIBO
2	Evaluate the possibility to install INDIGO platform on CINECA's premises (this activity could not be performed between January-June 2021, as the CINECA cloud infrastructure was undergoing renovations – and off-limits for non-critical, not in-production activities)	CINECA, UNIBO, WP2-TBA
3	Tighten integration between Examon cloud layer and IoTwins platform (see also action #3), possibly by replicating some data currently stored in Cassandra DB.	CINECA, UNIBO, WP2-TBA

3.2.7 Testbed 7

This testbed focuses on the computation of smart grid KPIs, performed close to the data sources. Having near real-time requirements, high data volumes, and distribution over a large geographical area, data needs to be pre-processed locally with regard to the needs of the processes which use the data.

In industrial systems, power quality monitoring enables the possibility to identify short current peaks that, otherwise, may lead to system failure. Prescriptive analytics will help to set up a system that can automatically react to issues in the power grid.

This testbed is already active in the framework of the "Aspern Vienna's Urban Lakeside" (https://www.ascr.at/en/), a smart city environment of private apartments, a student home, and a school, as well as a supply area with about 6.000 inhabitants and small business, which IoTwins will be able to exploit for data collection and configuration tuning.

The installed edge devices are based on the CP-8050 energy automation device from Siemens, that allows runtime installation of the applications. IoTwins will extend the existing framework with a more flexible edge platform which can process power quality measurements (including the necessary management functionalities); with a cloud-based digital twin providing system-wide power quality assessments and integrating off-line generated parameters into distributed twins running at configurable edges.

Table 7: TB7 action list

#	Description	Who
1	Organize a meeting with the TB7 owners to study the possibilities of connecting to the cloud (IoTwins). TB7 partners won't be available before January 11th, 2021	INFN
2	Support the TB7 to implement the current TB7 algorithms using the IoTwins services approach and test them in the scope of the TB7	UNIBO
3	Organize a meeting to analyse and assess the ML Algorithms appropriate to the kinds of data collected by the TB7	ESI

3.2.8 Table AS_IS vs TO_BE

In the following, we present a synoptic table of the current status (AS-IS) of testbeds and the future status (TO-BE) that each testbed is expected to reach with the support of the IoTwins platform. In Table 8, we present the functionality offered by the three computing environments that testbeds are already exploiting vs those that they intend to exploit in the near future through the adoption of the IoTwins platform. The table cells depicted in a grey background represent features already possessed by the testbed, while those in white signify that the corresponding functionality is missing. The text appearing in the cell specifies the particular technology testbeds intend to use to acquire the corresponding functionality.

Table 8: IoTwins platform design

		IoT		IoT-edge data transfer		Edge		Edge-cloud data transfer		Cloud/HPC	
		Storage	*Computation*	*IoT-to-edge*	*Edge-to-IoT*	*Storage*	*Computation*	*Edge-to-cloud*	*Cloud-to-edge*	*Storage*	*Computation*
TB1	AS-IS										
	TO-BE					InfluxDB	Docker+Mesos-Chronos	HTTP-based,Message-oriented?		Time-series (InfluxDB), Object-storage (MinIO)	INDIGO + Docker
TB2	AS-IS										
	TO-BE							HTTP-based,		Time-series (InfluxDB), Object-storage (Minnie)	INDIGO + Docker

		IoT		IoT-edge data transfer		Edge		Edge-cloud data transfer		Cloud/HPC	
		Storage	*Computation*	*IoT-to-edge*	*Edge-to-IoT*	*Storage*	*Computation*	*Edge-to-cloud*	*Cloud-to-edge*	*Storage*	*Computation*
TB3	AS-IS										
	TO-BE							HTTP-based			
TB4	AS-IS					InfluxDB; SQL server					
	TO-BE							HTTP-based			
TB5	AS-IS										
	TO-BE			Message-oriented		Time-series DM (No specific solution)	Docker+Mesos-Chronos	HTTP-based			INDIGO + Docker, ...

		IoT		IoT-edge data transfer		Edge		Edge-cloud data transfer		Cloud/HPC	
		Storage	*Computation*	*IoT-to-edge*	*Edge-to-IoT*	*Storage*	*Computation*	*Edge-to-cloud*	*Cloud-to-edge*	*Storage*	*Computation*
TB6	AS-IS		Data sent to centra MQTT broker via MQTT packets	**(IoT to cloud)** MQTT messages through high-speed interconnect	**(Cloud to IoT)** NA	NA (edge is skipped)	NA (edge is skipped)	NA (edge is skipped)	NA (edge is skipped)	Time-series DB (Kairos DB), Cassandra DB	Python scripts, Jupyter notebooks
	TO-BE	NA	Data sent to centra MQTT broker via MQTT packets	**(IoT to cloud)** MQTT messages through high-speed interconnect	**(Cloud to IoT)** Docker-like services cannot be used on the IoT (the computing nodes) no direct cloud-to-IoT communication is expected	NA (edge is skipped)	NA (edge is skipped)	NA (edge is skipped)	NA (edge is skipped)	Time-series DB (Kairos DB), Cassandra DB (potentially Examon could also work with InfluxDB – possibility studied within TB10)	INDIGO + Docker, Spark Jobs, potential "cloud-to-cloud" feedback (e.g., AI-based decisions could influence the job scheduler)

		IoT		IoT-edge data transfer		Edge		Edge-cloud data transfer		Cloud/HPC	
		Storage	*Computation*	*IoT-to-edge*	*Edge-to-IoT*	*Storage*	*Computation*	*Edge-to-cloud*	*Cloud-to-edge*	*Storage*	*Computation*
TB 7	AS-IS			Modbus	-	Time Series (currently proprietary)	OCI Containers (ARM32v7 architecture)	MQTT, nats.io, UPC-UA (measurement data)	MQTT and/or nats.io (for data transfer) Container Deployment via KubeEdge (containing apps and models)	Time Series (InfluxDB), Container Registry (Apps and Models inside containers), Kubernetes Backend (Rancher, AWS) for cloud app and edge node management	OCI Containers (amd64)
	TO-BE										

The information summarized in the table above has guided the (re)design of the functionalities that the IoTwins platform will have to deliver. In particular, the intense collaboration between the technical partners and the testbed owners made it clear that: i) testbeds need a support to implement, deploy and maintain new services on their own; ii) a strong "data service" support is requested by testbeds for what concerns the elaboration and transfer of data among the three computation layers. To this end, we offer two different sets of platform API that developers/users can use to implement/deploy IoTwins services and propose the design of platform-level data services to support data manipulation, storage and transfer.

4 IoTwins platform enhancement

4.1 Revised reference architecture

The IoTwins reference architecture (see Figure 3) was slightly modified to consider that:

- IoTwins will provide *Simulation* software. Basically, simulations are supposed to run at the cloud level. Customer may resort to simulation to get useful insights to optimize their production processes; also, they can exploit these tools to simulate hardly replicable or dangerous events concerning the machining operation (e.g., faults of machines or machine parts);
- *IoTwins Application developers* will access functionalities offered by the Platform Service Layer to code IoTwins applications.

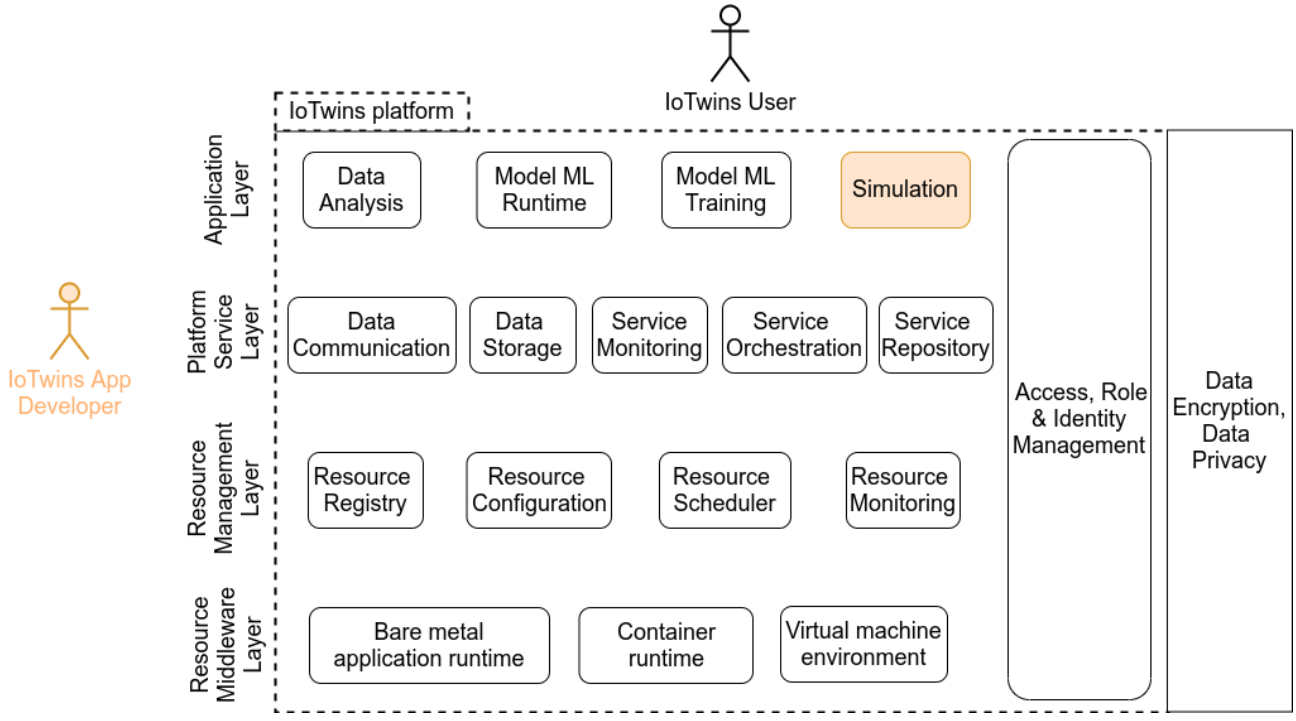

Figure 3: IoTwins reference architecture

4.2 Platform API

In the following sections, we describe two sets of IoTwins API that both developers and users can use to interface with the IoTwins platform functionality. Each API set complies with the IoTwins architecture

requirements discussed in the Deliverable D2.2 “Implementation of the IoTwins platform (I)”, and allows developers/users to interface to either of the two service orchestrators made available within the IoTwins project: INDIGO and Kubernetes.

4.2.1 Orchestrator INDIGO

The PaaS (Platform as a Service) solution adopted in this project allows the users to deploy Virtualised computing infrastructures with complex topologies (such as clusters of virtual machines or applications packaged as Docker containers), by using standardized interfaces based on REST APIs and adopting the TOSCA (Topology and Orchestration Specification for Cloud Applications) templating language. TOSCA is an OASIS open standard that defines the interoperable description of services and applications hosted on the cloud and elsewhere; including their components, relationships, dependencies, requirements, and capabilities, thereby enabling portability and automated management across cloud providers regardless of underlying platform or infrastructure.

The PaaS layer features advanced federation and scheduling capabilities ensuring the transparent access to the different back-ends, including on-premises cloud Management Frameworks such as OpenStack, public cloud providers such as Amazon Web Services and Microsoft Azure and finally, Container Orchestration Platforms such as Apache Mesos and Kubernetes. The selection of the best cloud provider to fulfil the user request is performed considering criteria like the user’s SLAs, the services availability, and the data location.

The API is organized around REST and has predictable resource-oriented URLs, accepts JSON-encoded request bodies, returns JSON-encoded responses, and uses standard HTTP response codes, authentication, and verbs. Authentication is performed via OAuth2 bearer token. The core resource is Deployment that represents a TOSCA template deployment.

The INDIGO-PaaS Orchestrator exposes the following REST APIs for its main functionalities provided to users.

Table 9: INDIGO REST APIs

API Name	Input Parameters	Output Parameters	Note
Authentication			
Authenticate	`OAuth2 bearer token`		
Deployment			
Get Deployment Status	`uuid`	`status`	
Create Deployment	`Template, parameters`	`uuid, status`	Include delayed schedule
Update Deployment	`uuid, parameters`	`202`	
Delete Deployment	`uuid`	`204`	
Get Deployment Extended Info	`uuid`	`template, parameters`	
Get Deployment Log	`uuid`	`Deployment log`	
Get Deployment Events		`Scheduled deployment events`	

API Name	Input Parameters	Output Parameters	Note
Infrastructure			
Get Resources	uuid	Creation time, state, resource	Include list of nodes that require this resource

In order to use this APIs the REST client must authenticate via OAuth2 bearer token (RFC 6750). A sample request to create a deployment could be:

```
POST /deployments HTTP/1.1
Content-Type: application/json
Authorization: Bearer <access token>
Host: localhost:8080
Content-Length: 234

{
  "template" : "template",
  "parameters" : {
    "cpus" : 1
  },
  "callback" : "http://localhost:8080/callback",
  "timeoutMins" : 5,
  "providerTimeoutMins" : 10,
  "maxProvidersRetry" : 1,
  "keepLastAttempt" : false
}
```

With response containing the uuid and status of the created deployment.

```
HTTP/1.1 201 Created
Content-Type: application/json;charset=UTF-8
Content-Length: 662

{
  "uuid" : "7b60c1ca-bd3f-4371-9124-0a629b8f86d6",
  "creationTime" : "2020-12-04T08:57+0000",
  "updateTime" : "2020-12-04T08:57+0000",
  "status" : "CREATE_IN_PROGRESS",
  "outputs" : { },
  "task" : "NONE",
  "callback" : "http://localhost:8080/callback",
  "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/deployments/7b60c1ca-bd3f-4371-9124-
0a629b8f86d6"
  }, {
    "rel" : "resources",
    "href" : "http://localhost:8080/deployments/7b60c1ca-bd3f-4371-9124-
0a629b8f86d6/resources"
  }, {
    "rel" : "template",
    "href" : "http://localhost:8080/deployments/7b60c1ca-bd3f-4371-9124-
0a629b8f86d6/template"
  } ]
}
```

The deployments can be deleted or updated with similar POST requests. A GET request can be used to retrieve the extended information associated to a deployment. This may be VM information for cloud deployments or JOB information.

The arguments required is the uuid of the deployment:

```
GET /deployments/7b60c1ca-bd3f-4371-9124-0a629b8f86d6/extrainfo HTTP/1.1
Authorization: Bearer <access token>
Host: localhost:8080
```

With response:

```
HTTP/1.1 200 OK
Content-Type: text/plain;charset=UTF-8
Content-Length: 1125

{
  "vmProperties" : [ {
    "class" : "network",
    "id" : "pub_network",
    "outbound" : "yes",
    "provider_id" : "external"
  }, {
    "class" : "network",
    "id" : "priv_network",
    "provider_id" : "provider-2099"
  }, {
    "class" : "system",
    "id" : "simple_node1",
    "instance_name" : "simple_node1-158799480931",
    "disk.0.os.flavour" : "ubuntu",
    "disk.0.image.url" : "ost://api.cloud.test.com/f46f7387-a371-44ec-
9a2d-16a8f2a85786",
    "cpu.count" : 1,
    "memory.size" : 2097152000,
    "instance_type" : "m1.small",
    "net_interface.1.connection" : "pub_network",
    "net_interface.0.connection" : "priv_network",
    "cpu.arch" : "x86_64",
    "disk.0.free_size" : 10737418240,
    "disk.0.os.credentials.username" : "cloudadm",
    "provider.type" : "OpenStack",
    "provider.host" : "api.cloud.test.com",
    "provider.port" : 5000,
    "disk.0.os.credentials.private_key" : "",
    "state" : "configured",
    "instance_id" : "11d647dc-97f1-4347-8ede-ec83e2b64976",
    "net_interface.0.ip" : "192.168.1.1",
    "net_interface.1.ip" : "1.2.3.4"
  } ]
}
```

The high-level reference architecture is depicted in Figure 4. The architecture can be broken down into the following main categories of components:

- Core components:
  - API server, providing REST endpoints to submit and handle the deployment requests;
  - Workflow Engine, that manages the deployment workflow;
  - Message Bus, providing a way of integrating services loosely and based on notifications (events).
- Plugins:
  - Cloud connectors, implementing the interfaces with the relevant Cloud Management Frameworks.
  - Container orchestration connectors, implementing the interfaces that abstract the interaction with the relevant container orchestration platforms, e.g. Mesos, Kubernetes.
  - HPC integration connectors, implementing the interfaces to interact with the HPC services; the envisage interaction is based on REST APIs provided by gateway hosted by the HPC site, e.g. using QCG APIs [R1] or SLURM APIs [R2].
  - Storage services connectors, implementing the interfaces to interact with the relevant storage management and orchestration services; the interaction is based on REST APIs provided by the storage services themselves.

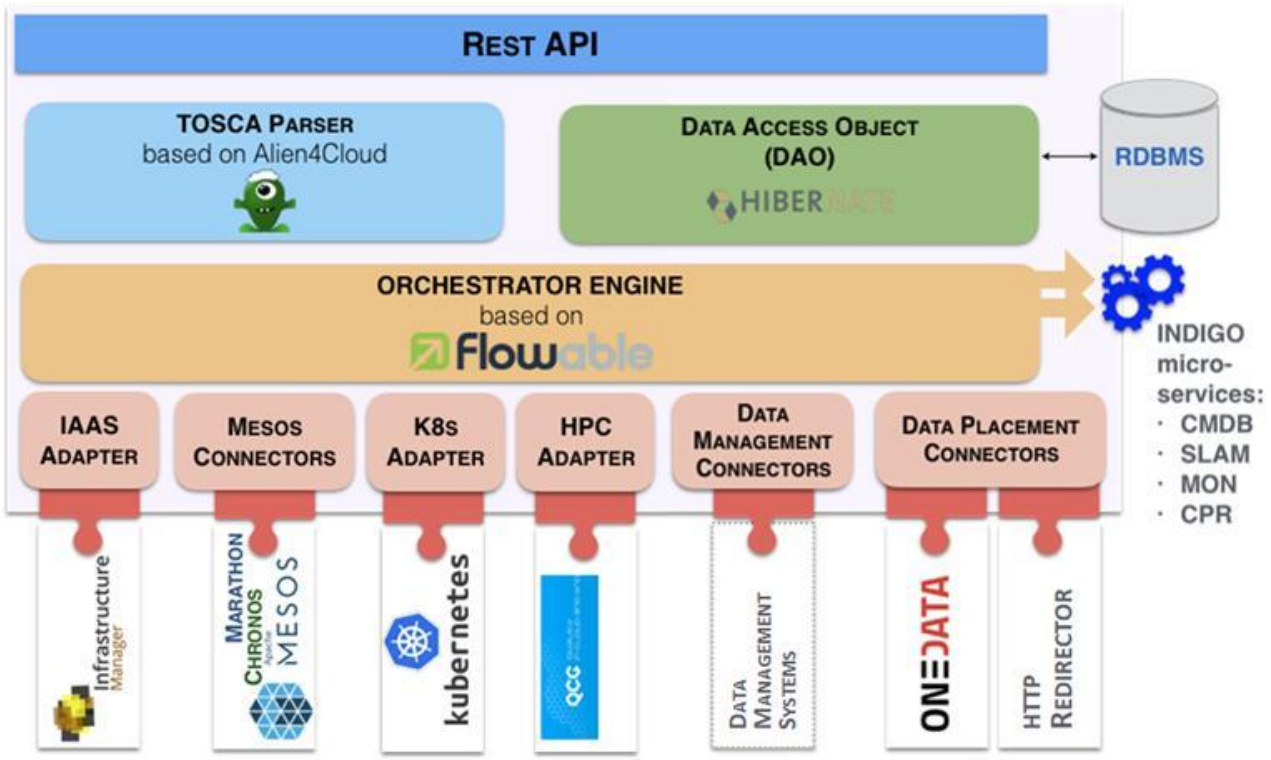

Figure 4: High level architecture for the PaaS orchestration

The INDIGO PaaS Orchestrator has been adopted to the IoTwins TB needs, in particular to deploy containers and VMs into MESOS Clusters on the Cloud and on the Edge layers. A new K8s connector is under development and testing and will be made available to the project in the next Orchestrator release.

The service is offered to IoTwins users through an instance deployed on the INFN-CNAF Cloud.

4.2.2 Orchestrator Kubernetes

The control plane components of the open-source container orchestration system Kubernetes are[1]:

- *API Server*
  This is a component of the Kubernetes control plane that exposes the Kubernetes API.
- *Controller Manager*
  This component runs the controller processes for the different kind of resources (nodes, jobs, endpoints, services).
- *Scheduler*
  This component watches for newly created Pods with no assigned node and selects a worker node for them to run on.
- *Configuration Storage*
  This is a high-available key value store used as Kubernetes' backing store for all cluster data.
  Kubernetes stores the serialized state of objects by writing them into the Configuration Storage.

These components are also running as containerized applications. Thus, it becomes possible to scale the system to the needs of the use-case. As shown in Figure 5 the system can run itself in a container (e.g., on a laptop) or as a high redundancy system in which the core components are themselves manages as redundant services.

All interactions with the system are done via the API Server. Even though there are client libraries for a large set of programming languages[2] and the powerful tool `kubectl`[3] can be used for command-line based interaction with the orchestrator, the API itself is a REST interface[4] which provides different endpoints for managing the different kinds of resources (workloads, services, nodes, etc.)[5] Thus it become possible to easily interact with the cluster from every custom service.

In general Kubernetes supports the imperative management of resources. However, in most of the cases the declarative approach is not only preferred, but also saves a lot of manual work and makes the system more resistant to failures and provides the possibility to manage large sets of similar edge setups together (see the general description below).

Using the imperative management approach, the user can explicitly command the system to run a specific container (in this example the container named `anomaly-detection` for CPU architecture arm32v7) in a pod on the specific edge node labelled with label "`weidlingau`":

```
kubectl run --generator=run-pod/v1 anomalyd --image=armv7/anomaly-
detection:1.13 --restart=Always --overrides='{"apiVersion": "v1", "spec":
{"nodeSelector": { "edgeNode": "weidlingau" } } }'
```

A better approach is to create a deployment descriptor or demon descriptor which specifies the circumstances under which a container is to be deployed to a node. Thus, the *Scheduler* and *Controller Manager* can manage the deployment and the cluster user is no longer required to update each pod

1 https://kubernetes.io/docs/concepts/overview/components/
2 https://kubernetes.io/docs/reference/using-api/client-libraries/
3 https://kubernetes.io/docs/tasks/tools/
4 https://kubernetes.io/docs/concepts/overview/kubernetes-api/#api-specification
5 https://kubernetes.io/docs/reference/kubernetes-api/

individually. The cluster user therefore creates a YAML[6] configuration file (JSON is also supported as a data format) which specifies the containers to be run in the pod, configuration parameters, dependencies, and circumstances under which the pods are to be deployed[7]. To make Kubernetes deploy the anomaly detection on each edge node that provides a certain configuration (which is identified by node label `TransformerXXY`) the following example deployment descriptor can be used:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: anomalyd
  labels:
    app: anomaly-detection
spec:
  replicas: 1
  selector:
    matchLabels:
      app: anomaly-detection
  template:
    metadata:
      labels:
        app: anomaly-detection
    spec:
      containers:
      - name: anomaly-detection
        image: anomaly-detection:1.13
        env:
        - name: CUSTOMER_BACKEND
          value: 192.196.1.1
        - name: PROVIDER
          value: SIEMENS AG
  nodeSelector:
    substationConfig: TransformerXXY
    nodeType: EdgeNode
```

The configuration is then applied to the cluster using the following simple command:

```
kubectl apply -f configfile.yaml
```

The *Scheduler* and *Controller Manager* thereafter assign a pod to each matching edge node. Thereby a node can be labelled using the following command:

```
kubectl label nodes <node-name> substationConfig=TransformerXXY
```

Whenever a new node is added and labelled accordingly a new pod will be deployed. Once the labels are removed or changed, the pods would be removed. In case the deployment descriptor is adapted and re-applied to the cluster all matching pods would be updated. Thus, there is no need to manage all pods individually anymore. The environment variables can be used to provide non-security relevant configuration

6 https://en.wikipedia.org/wiki/YAML

7 See for example https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

parameters that are valid for all deployed pods. In case security relevant information is to be provided, like tokens or passwords, Kubernetes secrets are to be used instead[8].

In case the system consists of more than one service and to reduce the effort of managing different version dependencies manually between the micro services, it is recommended to use Helm[9] charts[10] (see also section 4.4)[11].

```
helm create ts-anomaly-detection
```

This command creates a sub-directory that contains the following files:

```
Chart.yaml   charts/      templates/   values.yaml
```

The `Chart.yaml` file provides the information about the contents of the chart. Example:

```
apiVersion: v2
name: ts-anomaly-detection
description: Anomaly detection of transformer sub-stations

# A chart can be either an 'application' or a 'library' chart.
# Application charts are a collection of templates that can be
# packaged into versioned archives to be deployed.
#
# Library charts provide useful utilities or functions for the chart
# developer. They're included as a dependency of application charts to
# inject those utilities and functions into the rendering
# pipeline. Library charts do not define any templates and therefore
# cannot be deployed.
type: application

# This is the chart version. This version number should be incremented
# each time you make changes to the chart and its templates, including
# the app version. Versions are expected to follow Semantic Versioning
# (https://semver.org/)
version: 0.1.0

# This is the version number of the application being deployed.
# This version number should be incremented each time you make changes to
# the application. Versions are not expected to follow Semantic Versioning.
# They should reflect the version the application is using.
# It is recommended to use it with quotes.
appVersion: "1.16.0"
```

To the `charts` sub-directory dependent charts can be added that are needed to deploy the application (empty in this small example).

8 https://kubernetes.io/docs/concepts/configuration/secret/
9 https://helm.sh/
10 https://helm.sh/docs/topics/charts/
11 For an introduction on how to use Helm charts see for example https://opensource.com/article/20/5/helm-charts

The `template` directory contains the Kubernetes descriptor files like deployment- (see above), daemon-set-, service-, or ingress-descriptors. Thereby values to be filled-in like version umbers, environment variable values etc. are only referenced in these templates and provided in the `values.yaml` file (multiple versions of this file can be provided, e.g., for production and test environments).

Once all chart files are set up correctly, the chart can be applied to the cluster using the following command (`helm` uses the cam e configuration as `kubectl` and thus does not need any special configuration to access the cluster):

```
helm install ts-anomaly-detection
```

Even though all the described options provide a high degree for freedom for the cluster manager to adapt the system to his/her needs, it is one of the results of the first IoTwins project phase that the direct management of computing jobs should in the general case be hidden from the end-user.

This can for example be done by providing a custom interface that enables the user to command the system to apply desired functionality to any applicable location in the system (e.g., a graphical user-interface on which the user can select the anomaly detection functionality and the according setup). This system would then be responsible for creating, updating, and removing the deployment and daemon-set descriptors.

Another possibility to hide this complexity from the end-user is to implement the system to derive the needed deployments from the parameters entered by the user and the system state itself. When the user enters a threshold to be monitored for a given transformer via the user interface the system would then derive the necessity to deploy the monitoring and control components to the edge nodes and to deploy the according services in the backend.

Since the Kubernetes orchestration system provides its complete API via one service using REST (see above) the adaptation of the system to provide an automatic update of the descriptors is straight forward (as done for the sensorless control use-case in testbed 7, see IoTwins deliverable 5.2). On the one hand, this hides the complexity of the installed system from the end-user and enables the usage even when the user has no IT department available to manage the system. On the other hand, it requires the implementation of custom control logic components for the implemented use-cases (see component “Service Control Logic” in the cloud level in IoTwins deliverable D2.2).

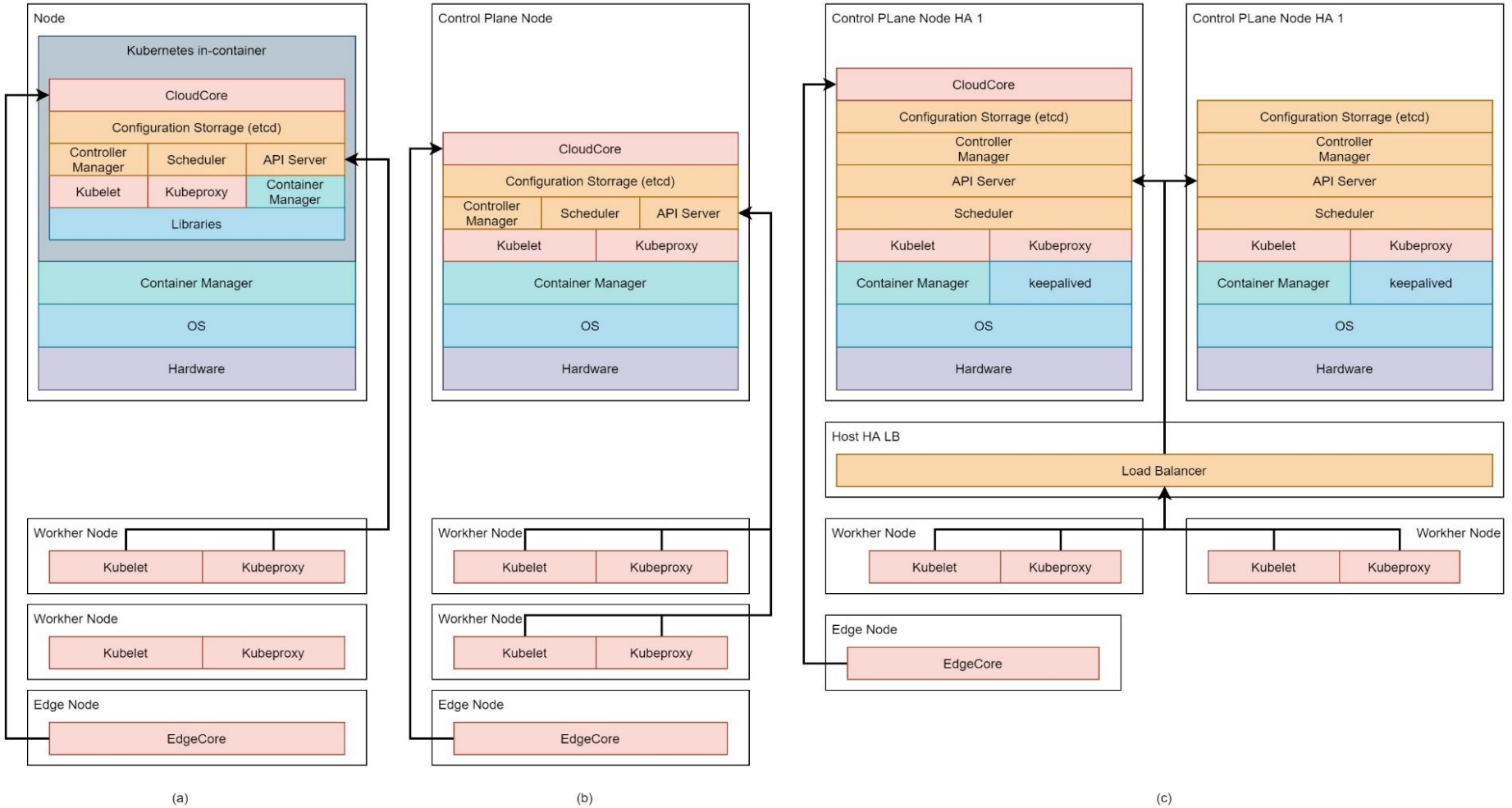

Figure 5: Different instantiation of the Kubernetes software stack providing the same functionality but with different levels of availability (see also IoTwins Deliverable 5.3, section 4.3).

4.2.3 S3 storage

To satisfy the requirements on the Data Storage side of the Platform Service Layer of the architecture, the proposed object storage solution selected for the architecture is based on de facto standard protocols and APIs.

The solution is a cloud native object storage that provides compatibility with Simple Storage Service (S3) protocols (i.e. MinIO). This protocol can be employed to store any type of object, backup and recovery, and hybrid cloud storage. It provides high availability in a scalable solution and low latency.

Requests are authorized using access control systems that can be connected with the platform's Identity Management solution. Objects are organized into buckets, and are identified by a unique, user-assigned key. Bucket names and keys are chosen so that objects are addressable using HTTP URLs.

The object storage is accessible through a web service interface, as well as CLI and RESTful APIs. The most common S3 APIs are reported in the following table.

Table 10: S3 REST APIs

API Name	Input Parameters	Output Parameters	Status code
Buckets			
Create Bucket	Bucket name	Bucket location	
List Buckets	-	Buckets name	
Delete Bucket	Bucket name	-	204
PutBucketACl	Bucket name, ACL	-	
GetBucketACL	Bucket name	ACL	
Objects			
Put Object	Bucket name, Object key, Body	Object location	
Get Object	Bucket name, Object key	Body	
Delete Object	Bucket name, Object key	-	204
List objects	Bucket name	Objects key	
PutObjectACL	Bucket name, Object key, Object ACL	-	
GetObjectACL	Bucket name, Object key	ACL	

A sample PUT requests to create a bucket requiring only the Bucket name and authorization codes is:

```
PUT / HTTP/1.1
Host: new_bucket.iotwins-minio.cloud.cnaf.infn.it:9000
Content-Length: 0
Date: Wed, 01 Mar  2021 12:00:00 GMT
Authorization: authorization string
```

The response would be:

```
HTTP/1.1 200 OK
Date: Wed, 01 Mar  2021 12:00:00 GMT
```

```
Location: /new_bucket
Content-Length: 0
Connection: close
```

To retrieve objects from S3 storage you can use a GET request; you must have READ access to the object. If the READ access is granted to anonymous user, you can return the object without using an authorization header. A S3 bucket has no directory hierarchy such as you would find in a typical file system. One can create a logical hierarchy by using object key names that imply a folder structure. For example, instead of naming an object data.csv, you can name it metrics/2021/February/data.csv.

To get an object from such a logical hierarchy, specify the full key name for the object in the GET operation.

```
GET /Key HTTP/1.1
Host: new_bucket.iotwins-minio.cloud.cnaf.infn.it:9000
Authorization: authorization string
```

The response will contain information on the object and its content.

```
HTTP/1.1 200
Last-Modified: LastModified
Content-Length: ContentLength
Cache-Control: CacheControl
Content-Disposition: ContentDisposition
Content-Encoding: ContentEncoding
Content-Language: ContentLanguage
Content-Range: ContentRange
Content-Type: ContentType
Expires: Expires

Body
```

4.3 Platform-Level Data Services

Data may take a very long journey from the time it is produced by sensors at the shop floor to when it is consumed again at the shop floor in the form of useful information elaborated along the IoT-edge-cloud path. The IoTwins platform will offer services to support the data management along the path.

A typical data “journey” may comprise the following actions. A data stream produced by a sensor installed on the working machine reaches the IoT Gateway. Here, it gets cleaned and polished (optionally, locally saved) before being sent over to the edge. At the edge, a ML-based predictive maintenance software detects potential data anomalies from the data flow. In case anomalies are to be predicted; actions are signalled back to the IoT Gateway that may act upon actuators. Data landed on the edge is also gathered in a long-term storage and periodically sent in bulk to the cloud. Once here, data is fed to a ML model for continuous training.

For another data path, see the sensorless control scenario described in IoTwins Deliverable D5.2 where sensor data and data from a third-party source are combined at the cloud level to create ML-Models that estimate the time delayed sensor data based on measurement data at other grid locations which can be measured in real time. The ML model is afterwards deployed to the edge Device and its performance is continuously monitored to start a retraining when the accuracy is not in the tolerable range.

In the scope of IoTwins, Data services are logically broken down into the following categories:

- *Data storage services:* Services/tools for long-term storage of data. By way of example, the following tools will be provided: Timeseries DB (InfluxDB), NoSQL DB (MongoDB), Relational DB (MySQL, Postgresql), File/Object storage.
- *Data transfer services*: This category refers to services/tools to support the transfer of data among the three computing levels (IoT->edge, edge->cloud, etc.). Examples of data transfer tools to be provided by IoTwins are Message-oriented brokers (compliant with main messaging protocols like, e.g., AMQP, MQTT), bulk data transfer (mainly http-based), and the OPC-UA Pub/Sub library[12].
- *Data elaboration services*: Several services will be offered for elaboration of at any level based on the algorithms that are designed, adapted, and implemented in WP3:
    - Services that train artificial neural networks (and optimize their structure, dee also D3.2 and the hyper parameter tuning algorithm provided by WP3) supervised based on historical labelled data.
    - Services that monitor the performance of deployed ML-Models to invoke a re-training and/or to deactivate the usage of the current ML-model.
- Adapters/connectors: These tools are devised to ensure integration among heterogeneous components/protocols/data formats, this service category will include: Data Publishers (e.g., software agents that fetch data from a given source and deliver them to message brokers), Data Subscribers (software agents that grab data from message brokers and deliver them to a given target entity), File format converters, etc.

The IoTwins project aims at implementing the mentioned services and offering them through the service repository for use by its partners and for future re-use by practitioners. All services will expose public REST API to support full integration with any component/application that needs to make use of them.

In the future, being the IoTwins platform an open framework, more services/tools than the ones already bundled may be developed and published in the service repo. This applies to all categories. For instance, for users that wish to port/implement their application on the IoTwins platform but have got lots of data represented in a proprietary format, a customized file converter to integrate their data with existing elaboration services needs to be developed from scratch.

4.3.1 Implementation Details

For the actual implementation we will use latest state-of-the-art tools to store (e.g. InfluxDB, MongoDB, PostgreSQL, Redis, …), process (e.g. NiFi), analyse (e.g. JSON Rules Engine), visualize (e.g. Grafana), secure (e.g. Keycloak) and manage data and services (e.g. nats, traefik, ELK stack, Docker, Kubernetes, …).

As discussed above, the general functionalities needed for the Data Services, can be clustered in several categories. We've visualized this process in Figure 6 below. The implementation of such a data distribution service, involve the developments of applications (e.g., AI functions or data visualization) that might reside at a centralized cloud instance or within edge nodes, depending on the context. While some of the produced information (e.g., sensor data or analytics results) is being processed locally other is streamed and shared between infrastructures. Important is the translation (e.g. SNMP client or REST library) towards the local system to extract measurements and to trigger actuations. These local systems (e.g., a production machinery)

[12] Made publicly available by partner Siemens AG at: https://github.com/siemens/opc-ua-pubsub-dotnet

are proprietary to the given use case and act the relevant data sources (measurements) and data sinks (actuations).

Such an infrastructure, however, could span multiple hierarchical levels. Depending on the use case, the requirements and the business model, the logical extreme of a network (edge) may vary widely. Further, the data processing does not necessarily have to take place at the topologically outermost extreme. Rather, there is a broad continuum of possible positioning in a network (often denoted as Fog Computing). The location of functionality follows the relevant requirements and does not constitute a binary decision. According metrics decide for the placement of functionality includes aspects related to the network (e.g., latency, jitter, network reliability, backhaul traffic costs, …), the data (e.g., privacy and protection requirements, context awareness, autonomy, …), or the costs (e.g., economy of scale, energy efficiency, …).

Further, we will provide APIs to register and manage edge nodes, to describe and find metrics within the infrastructure, to control the data flow between nodes, and to trigger actions at given assets. For example, for the Data Elaboration Services / Gateways, a promising approach might be to take advantage of Apache NiFi processors to extract information from sensors, to pre-process the data, to buffer and convert information, and to route them between nodes and services.

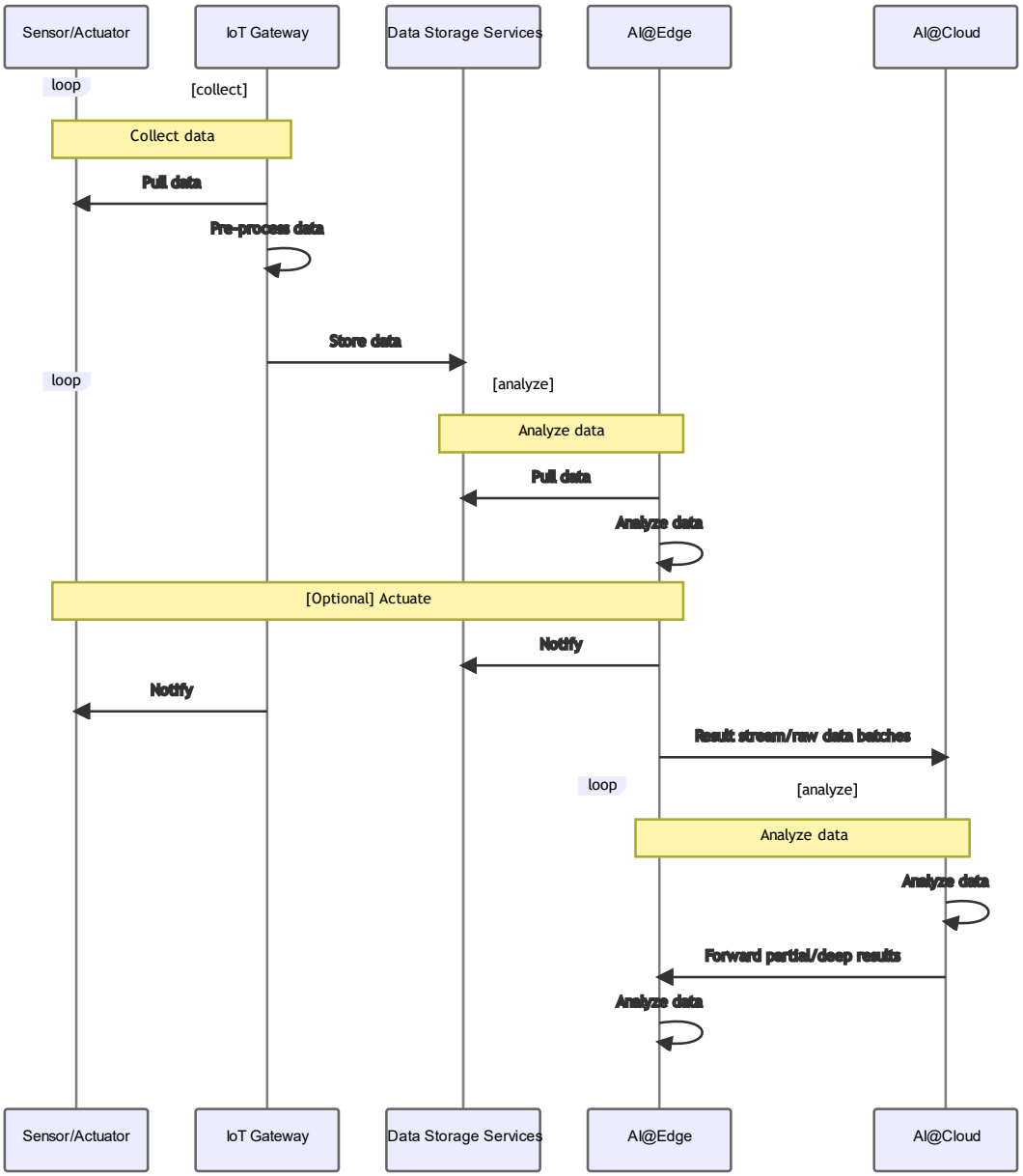

Figure 6: Data Flow and Processing Overview

In Figure 7, the dynamics concerning the building of edge-to-cloud service pipeline are explained. The INDIGO PaaS orchestrator takes care of User requirements and triggers the actions required to deploy data elaboration services on both the edge and the cloud side. In the considered scenario, a Data analytics application and a data storage service are requested to be activated at the cloud end, while the edge node will host a data storage and a data visualization instance (via deployment of InfluxDB and Grafana applications, respectively).

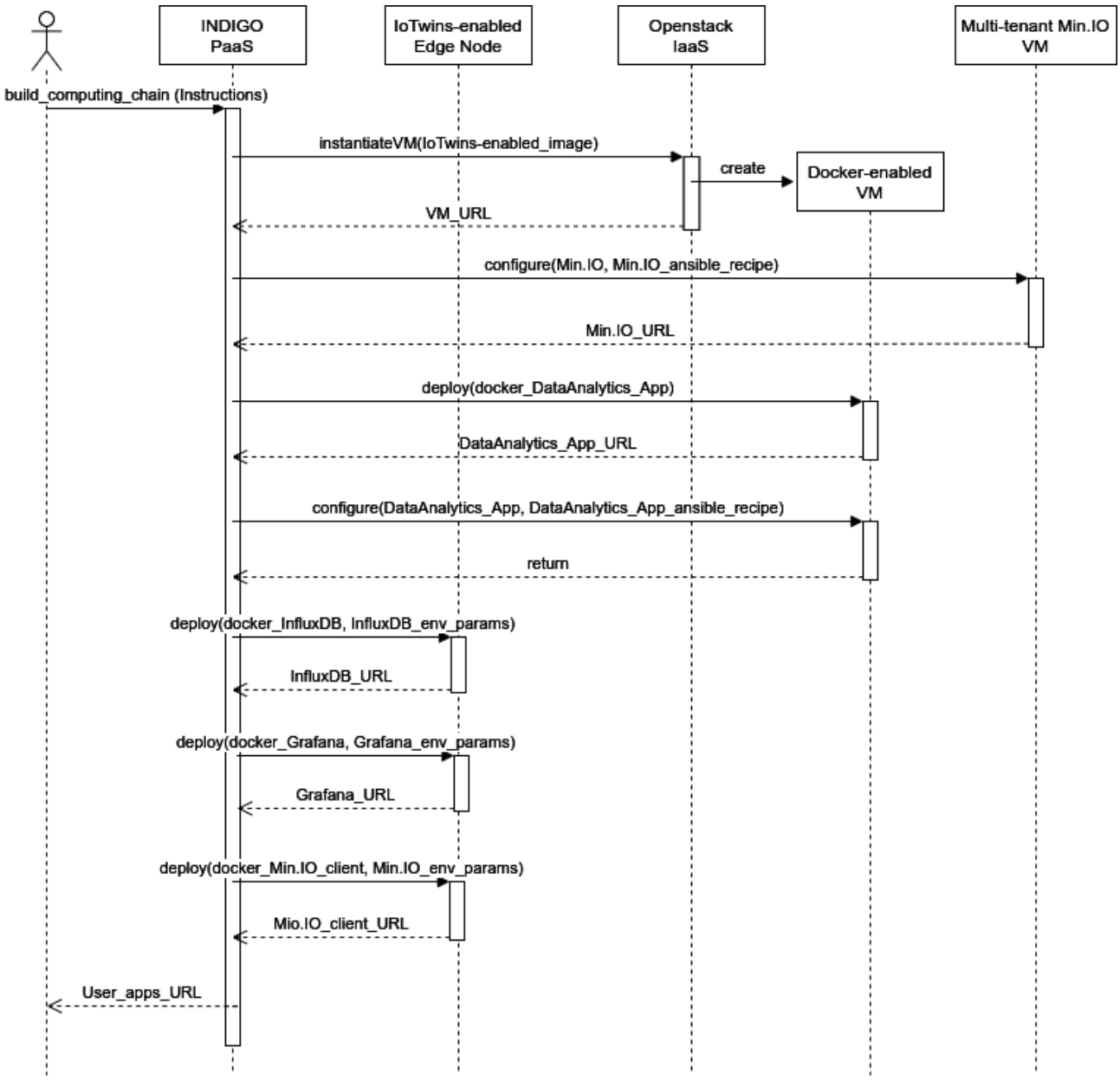

Figure 7: INDIGO PaaS building an edge-cloud computing chain

In the Figure 8, we depicted the building of a computing chain spanning the three levels: IoT, edge and cloud. In the considered scenario, a ML model gets trained in the cloud with historical data and is moved to the edge where it will execute with fresh data coming for IoT. In the sequence diagrams, data services are also deployed that support the application business logic (MQTT, InfluxDB).

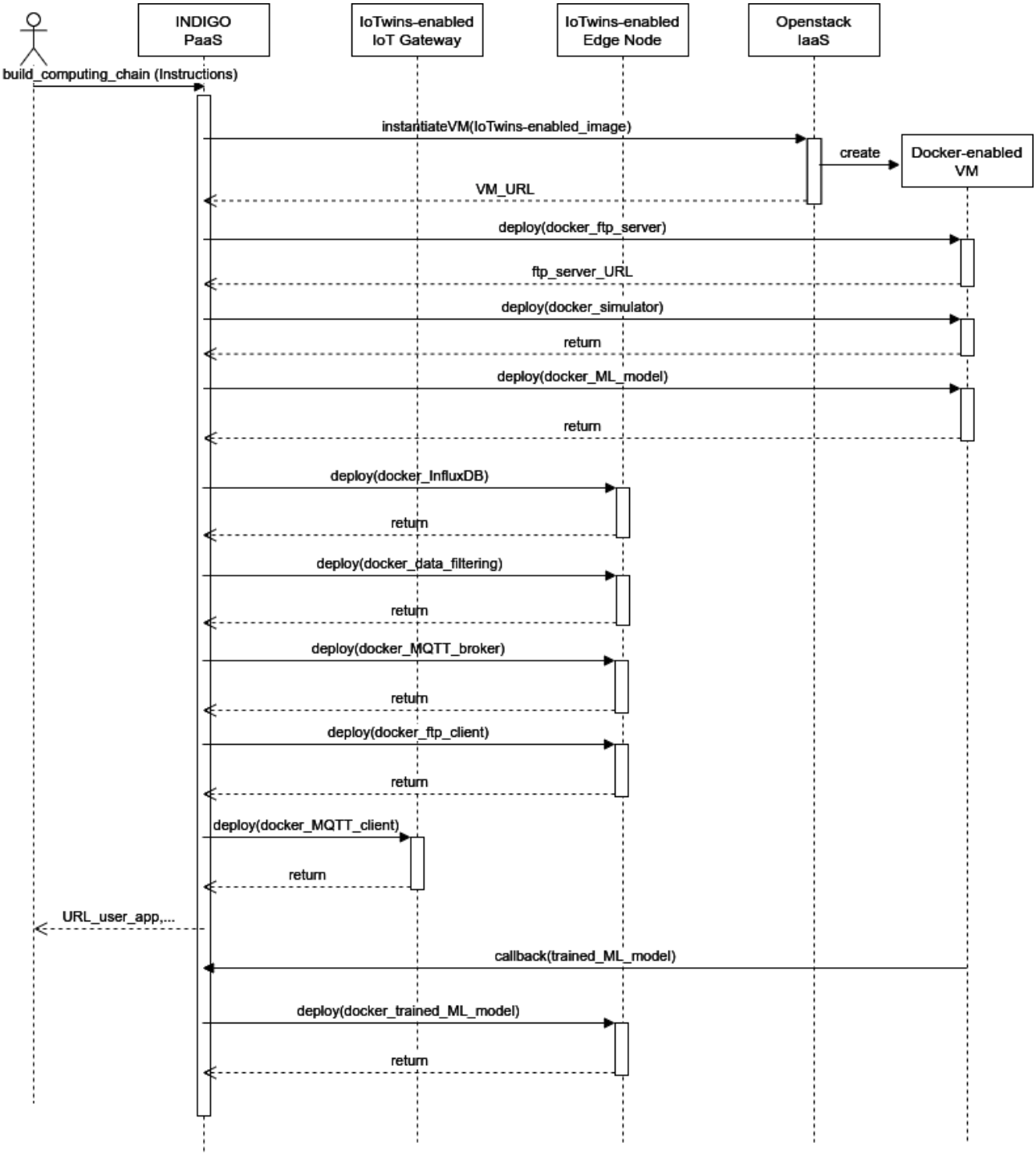

Figure 8: INDIGO PaaS building a IoT-edge-cloud computing chain

4.4 Private Docker repository (Harbor)

Harbor[13] is an open-source registry used to store both container images (such as those used by Docker container framework) and Helm charts (for Kubernetes). It provides an extension to the Docker Distribution by adding security and identity management functionalities to the registry.

13 https://goharbor.io/

The private repository is deployed on INFN cloud resources (managed by Openstack) with extensible volume for storage. The access to the private repository is done via OpenIDConnect authentication provided by the INDIGO Identity and Access Management Service[14] using persistent access tokens.

The Harbor registry is accessible and manageable through a Web UI as well as via Harbor RESTful APIs[15].

The repositories are managed through 'projects' with Role based access control for the users, and support LDAP/AD protocols to inherit groups policies and give users permission to specific projects.

5 Conclusion

This deliverable reports on the actions, considerations and design decisions taken so far, with regards to the IoTwins architecture and related data services. First, it describes the results of the interviews conducted with the Testbeds, by WP2 and WP3 technical partners. Concretely, it summarizes, for each testbed, the collected feedback, and a list of actions to be performed. Some of the most noteworthy insights gathered during this process are, first, that most testbeds already have IoT and edge levels well defined and an ongoing implementation, second, that, generally, testbeds are more demanding than expected, needing further assistance, at the cloud level and the relation between IoT and cloud. Moreover, some testbeds requested support for their machine learning deployments, which are mainly based on PyTorch, TensorFlow and Keras, concretely, some partners use PyTorch while WP3 partners mainly use TensorFlow and Keras.

Regarding the IoTwins platform design, this deliverable provides a slightly updated version of the reference architecture, to take into account that IoTwins is also intended to provide some simulation services, that may run at the cloud level, such as the ones used by Testbed 5, in order to simulate people movement. Also, it is revised to that it enables the development of IoTwins applications by further developers.

Passing through the platform API's high-level definitions, involving data management services and orchestrators, this deliverable also tackles the overview instantiation and implementation details with respect to the IoTwins data services, that will be implemented by the partners, relying on state-of-the art and broadly-known software, such as MongoDB or InfluxDB for storage or Keycloack for security.

The jobs can be performed on either edge or cloud depending on the application logic and a front-end application may be refreshed depending on the results.

To sum up, this deliverable is intended to portray the advancements regarding the IoTwins architecture and data services, paving the way towards the distributed digital twin materialization.

14 https://indigo-iam.github.io/docs/v/current/

15 https://iotwins-harbor.cloud.cnaf.infn.it/devcenter-api-2.0