



Grant Agreement N°857191

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

DELIVERABLE 2.4 - IMPLEMENTATION OF THE IOTWINS PLATFORM (II)



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	FOKUS
Contributing Partner(s)	BSC, INFN, ESI, SAG, SAGOE, THALES, UNIBO
Author(s)	D. Nehls (FOKUS)
Contributor(s)	J. C. Ahouangonou (ESI), F. Kintzler (SAGOE), G. Di Modica (UNIBO), S. Rossi Tisbeni (INFN), C. G. Calatrava (BSC), C. Delamarre (ESI)
Reviewer(s)	Daniele Cesini (INFN), C. G. Calatrava (BSC), Vincent Thouvenot (THALES)
File Name	D2.4 – Implementation of the IoTwinS platform (II)
Contractual delivery date	M26 – 31 October 2021
Actual delivery date	M28 – 16 December 2021
Version	1.0
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
V0.0	09/15/2021	Initial creation of the current document
V0.1	09/30/2021	Structure and preliminary content
V0.2	10/10/2021	Added content for chapter 4
V0.3	10/15/2021	Chapter 3
V0.4	10/25/2021	Chapter 1, 5, 6
V0.5	10/29/2021	Version under review
V0.6	11/19/2021	Consortium review
V1.0	12/16/2021	Final Version

Executive summary

This deliverable is the second in a series of three deliverables and reports on the status of the implementation of the IoTwinS platform.

Whereas the first of the series had its focus on the theoretical and architectural foundation of the platform, the emphasis of this document lies more on the daily work for and with the implementation.

This document includes a detailed description of the GitLab workflow for the platform itself and, especially, the platform's and AI services. The GitLab pipeline establishes a thorough testing of software artifacts and increases overall software quality.

Chapter 3 provides an extensive example of the IoTwinS service deployment lifecycle, describing two different use cases. The first use case exemplifies the integration of edge node deployments in cloud-based services with the help of the INDIGO Orchestrator. The second use case shows an alternative approach based on KubeEdge.

Both ways follow the same architectural patterns introduced in D2.2, the first iteration on the "Implementation of the IoTwinS platform".

Finally, chapter 4 showcases different approaches of platform adaption on selected test-beds.

Table of Contents

Executive summary.....	4
1 Introduction.....	6
2 References and Abbreviations.....	6
2.1 References.....	6
2.2 Abbreviations.....	7
3 Gitlab Integration	7
4 Service deployment examples.....	9
4.1 Demonstration on edge and cloud.....	9
4.2 Integrated cloud and edge Level Service Management	16
5 Summary of Test-bed implementation	21
5.1 Categories of platform implementation.....	21
5.1.1 Fully Integrated.....	22
5.1.2 Partially Integrated	28
5.1.3 Autonomic implementation	30
6 Conclusion	38

1 Introduction

This deliverable reports on the state of implementation of the IoTwinS platform. It is a successor to D2.2 - Implementation of the IoTwinS platform (I), which introduced the IoTwinS reference architecture and gave an overview of the required functional blocks at IoT, Edge and Cloud each level (Figure 1).

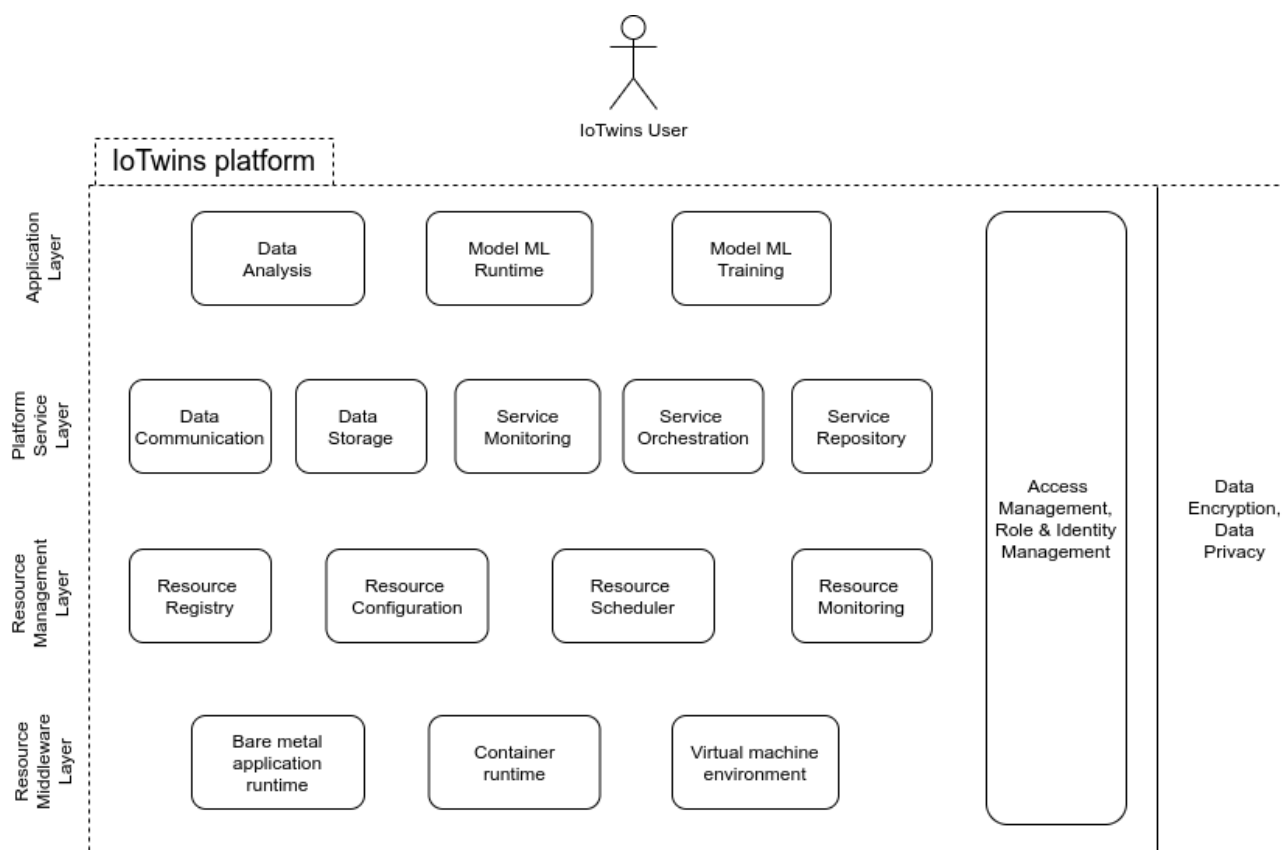


Figure 1: IoTwinS Functional Blocks

The evolution of the architecture was described in D2.3 – Architecture, Core Data and value-added Services Bundles Specifications (II). It reported on the feedback gained from the test-beds in several interviews and deferred a slightly updated reference architecture from this. Whereas concentrating on architecture and specification by design produces a deliverable focused more on theory, this deliverable at hand in contrast will focus more on the daily work, as it describes the established Gitlab development pipeline, provides description of how to deploy a service on the platform and summarizes the adoption of the platform at the different test-bed sites.

The different approaches at test-bed level and especially the two service deployment implementations highlight the flexibility of the IoTwinS reference architecture.

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

D2.3 Architecture, Core Data and Value-Added Services Bundles Specifications (II), v1.0, July 2021

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, July 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 Gitlab Integration

Task 2.5 - Technical Verification and Integration Testing, as one of the main contributors of D2.4, will follow all software development activities and will employ a common software verification and testing framework to be used on all software outputs and to check software integrity. An integration plan is prepared to guide the integration of the developed backbone infrastructure with the various services and components. The components of the platform will be covered by functional and integrated tests wherever possible. To keep the quality of IoTwinS User Interface components, manual test scenarios will be created based on the input received and become automated when possible. Appropriate testing frameworks and tools will be identified that meet the requirements of the components' testing developed within this project. Execution of both automated and manual test are expected to be performed on a regular basis, synchronized with release schedules.

Gitlab¹ has been selected as the technology for storing the code repositories for the IoTwinS project. The Gitlab repositories are hosted at CINECA.

Gitlab offers a convenient continuous integration and testing technology, Gitlab runners, that will be used as the official verification and testing framework for the platform.

The specific software technology that will be used is GitLab Runner². Runners are the agents that run the CI/CD jobs that come from GitLab and are executed when events at the repository launch them -- even if they are not run at the repository itself but in a container, in a Kubernetes cluster, or in auto-scaled instances in the cloud. The testing framework will require Docker³ or Singularity⁴ to provide local code testing environments. Furthermore, continuous code quality testing can be integrated through the sonarqube⁵ platform, which allows automated code review and integrates well with Gitlab pull reviews.

Code documentation is expected to be done either Natural Docs⁶, Sphinx⁷, or Doxygen⁸ (starting from the requirement that the documentation must work for a multi-language project and be as little intrusive as possible).

For Data testing, documentation, and profiling Great Expectations is recommended⁹. It would be optimal if data is stored using some form of version control such as DVC which would allow to keep different working versions of data and processing pipelines.

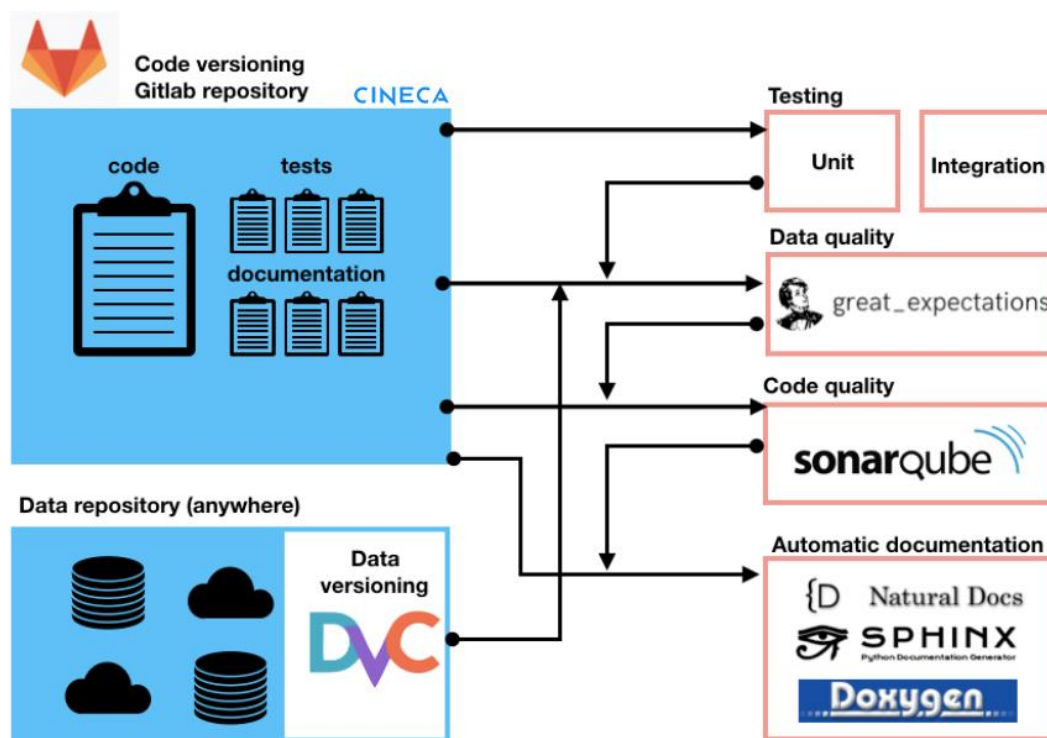


Figure 2: GitLab Integration

¹ <https://www.gitlab.com>

² <https://docs.gitlab.com/runner/>

³ <https://www.docker.com/>

⁴ <https://sylabs.io/>

⁵ <https://www.sonarqube.org/>

⁶ <https://www.naturaldocs.org/>

⁷ <https://www.sphinx-doc.org/en/master/>

⁸ <https://www.doxygen.nl/index.html>

⁹ <https://greatexpectations.io/>

4 Service deployment examples

This section reports on two different implementations of a service deployment scenario. The intention thereby is to show the flexibility of the architecture defined in D2.2 and to validate the congruency of the different approaches for the given use-case.

4.1 Demonstration on edge and cloud

This use-case implements the edge and cloud specification within an IoTwinS platform that leverages component standardization to let the end-user choose between execution on the edge or the cloud. In terms of software, the cloud level service is using Indigo Orchest10 and Mesos11 whereas the edge level service relies on Marathon12/Chronos13 on top of Mesos.

There are two personas involved: a Developer and a User who interacts with the platform across the Internet. A Developer develops and provides a simple web app requiring an execution engine, he also makes the web app available on the web. User accesses the web app, defines inputs, launches some computation, and gets a result.

In this example, and for the sake of reproducibility, the User is free to select either edge or cloud engine within the web app, but in practice this selection may be hidden to the User and done by the web app depending on inputs parameters, pre-computed data presence, user/data locality, etc. The computationally-intensive execution is scheduled via proper usage of orchestrators to allow scalability as well as resource allocation on edge and cloud. Dashboards are also available to check execution status and track resource usage as multiple computations can be launched on either engine. The deployment of the web app is also done once by the Developer through a command passed to the cloud orchestrator to expose the web app to the Internet.

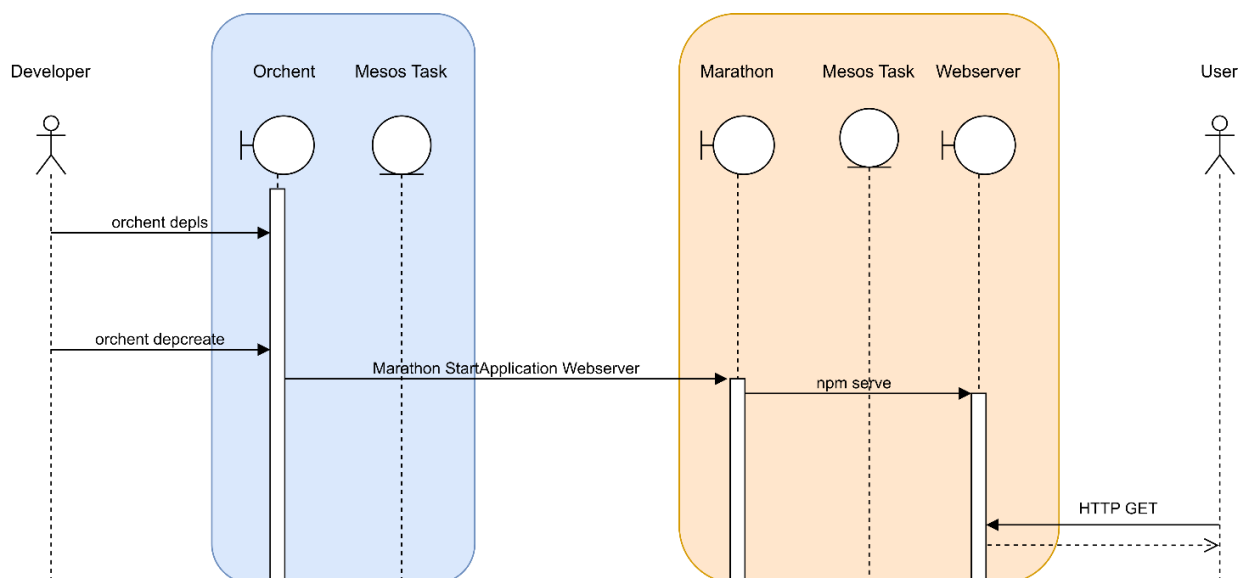


Figure 3: setup frontend on the cloud through Marathon

¹⁰ <https://github.com/indigo-dc/orchent>

¹¹ <http://mesos.apache.org/>

¹² <https://mesosphere.github.io/marathon/>

¹³ <https://mesos.github.io/chronos/>

The initial setup of the web app is described as a sequence diagram between various services. In this use case, an INDIGO orchent command-line tool is installed on the Developer machine and a TOSCA deployment file is written. The web app is already built and available within its associated container. The authentication between various services is pre-configured.

Before deploying, the Developer can check the status of the deployment using `orchent depls`. This command lists the already deployed services and can be used, throughout the use-case, at any time, to list the deployed services. The command talks to the INDIGO Orchent service deployed in the cloud to be widely available and to relax network constraints for the Developer.

To initiate deployment of the web app on the cloud and edge, the command `orchent decreate` with a deployment file argument allocates the needed containers and communication channels. This file corresponds to the configuration of the web app in terms of resources on cloud and edge sides, as well as data exchange between the resources. It follows the Oasis TOSCA specification serialized in a YAML file. Once the configuration is transmitted to the Orchestrator, the resources are created on edge using the Marathon `StartApplication` command. A Webserver (`npm serve` in this use case) is started and a URL to connect to it is sent back to the orchestrator.

The web app is exposed to the web on the edge side and any User can start to access it through the provided URLs, it behaves like a classical application on a web page with a dedicated user interface. For this use-case, the User can define parameter fields, select a platform to execute on (either edge or cloud), and click on a Launch computation button that will trigger more actions on the orchestrated services depending on the selected platform. As written in the introduction content, this is a simple web app that does not automatically select the platform depending on edge or cloud state (availability, load, data presence, etc.) nor web app parameters value (range, corner cases, computational complexity, etc.) but let the user select the node by himself.

Note that the Developer was not be involved at all during the application life cycle as in any other web application.

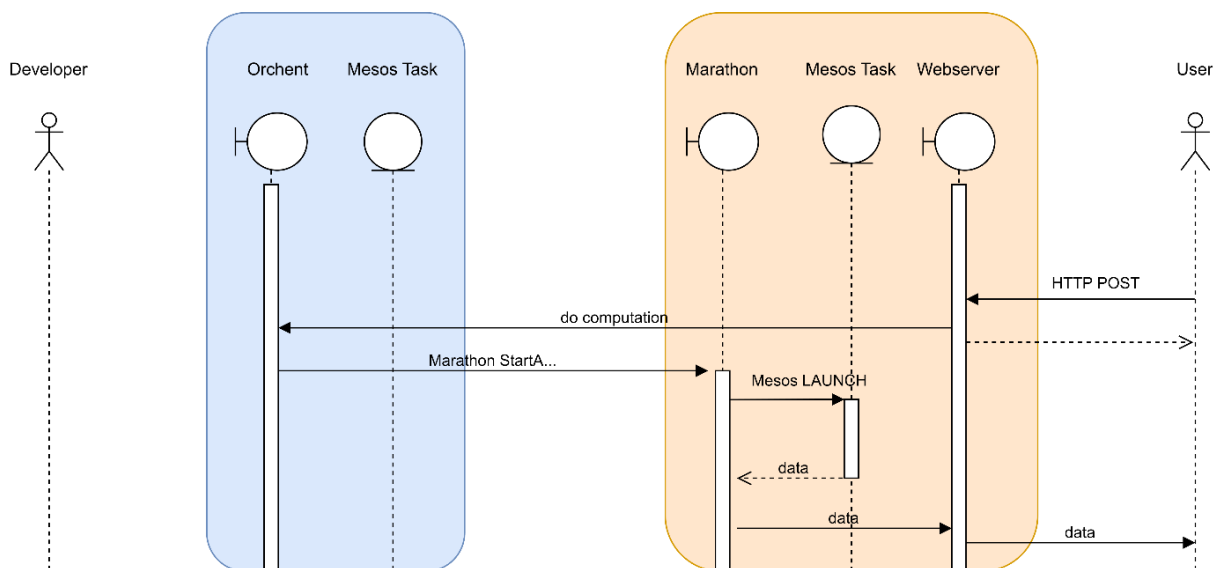


Figure 4: compute on the cloud

When the User selected edge as the execution platform and clicked on the Launch computation button, a new orchestration is performed. The User is at the origin of the action thus pushed some information to the

Webserver. A reverse connection from the Webserver to the Orchestrator asked it to allocate and start a computation on the edge side through the Orchestrator REST API.

A command `Marathon StartApplication` was then emitted from the Orchestrator to Marathon to create the corresponding `MesosTask` with `Mesos LAUNCH` operation. During this process, web app parameters were mapped from HTML values to JSON values and passed around to ensure that values previously filled by User are available to `MesosTask`. In the meantime, the web app was notified about job creation and it alerted the user that an operation is in progress. This behavior allows the web app to be reactive and display information even if no data is available yet.

After the computation was completed, the computed data were passed to the web app via HTTP PATCH verb to refresh the initially provided content with newly computed data from the `MesosTask` (or any other mean such as `socket.io`). Using this edge side computation does not round-trip data over the cloud which might reduce the transport cost of big data.

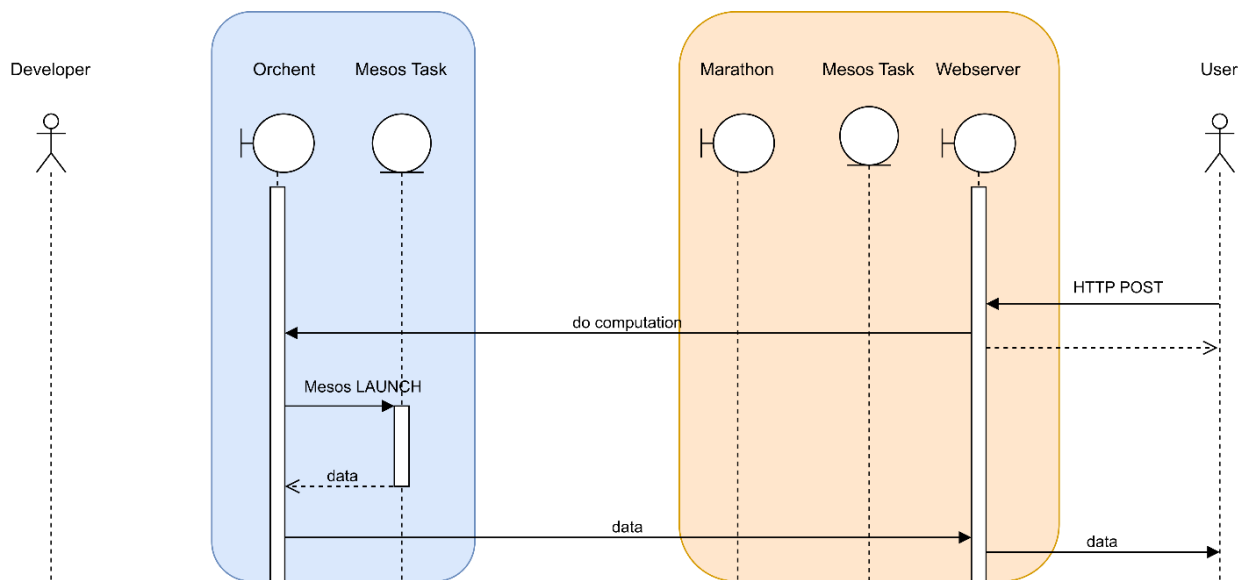


Figure 5: compute on the edge

When the User selected cloud as the platform and clicked on the Launch computation button, a different orchestration is performed, and another sequence diagram let us describe how previous objects interact. The User is still at the origin of the sequence via the click on the button and a reverse connection from the Webserver to the cloud asked for execution on the cloud side via the Orchestrator REST API. Like in the edge side execution case, the web app was notified early that a task allocation is in progress and informed the User that execution is in progress.

With the cloud execution platform selected, the Orchestrator used a cloud `MesosTask` to execute the computation with parameters selected by User and mapped parameters the same way as in the edge execution platform. However, in this case, Apache Marathon is not involved; the task is directly instantiated by the Orchestrator on a matching service template which offers more flexibility in term of execution.

After the computation was completed, the data were sent back to the Webserver by the Orchestrator. These data were then served through an HTTP PATCH verb to update the previously sent content with the new data (or any other mean like `socket.io`). From the web app logic, this is like executing on the edge.

The following image illustrates a process flow. The arrows are numbered to show the different steps.

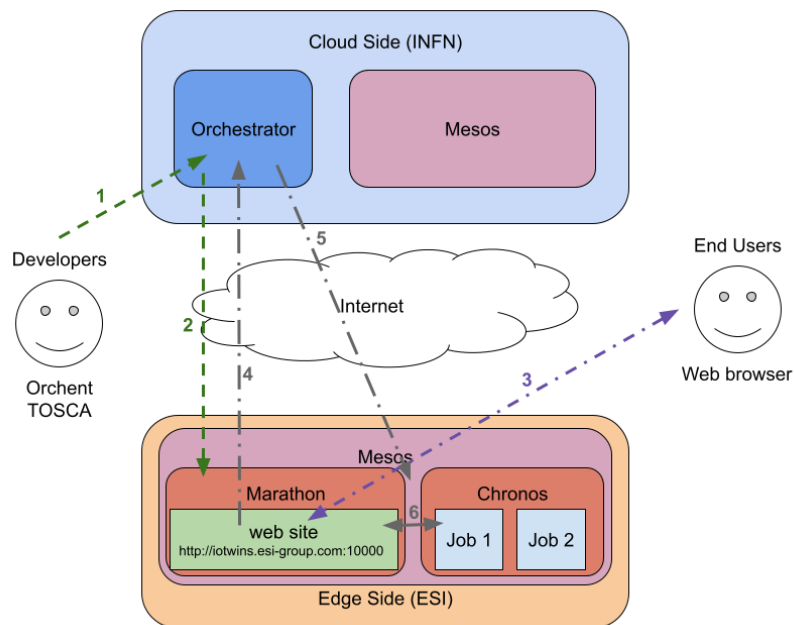


Figure 6: application behavior overview

Associated with the behavior overview, the following lines explain step by step by step how to proceed. These steps correspond to the arrow numbers. We will use the Command Line Interface (CLI) to execute the platform command.

```
Deployment [11ebf042-e743-f46b-afc8-02421473c2d3]:
status: CREATE IN PROGRESS
creation time: 2021-07-29T07:59+0000
update time: 2021-07-29T07:59+0000
outputs:
{}
```

Figure 7: orchent decreate output

Step 1: the developer requires the INDIGO Orchestrator to deploy the Web application by typing `orchent decreate website_service.yaml` which produced a status message. Developers can check whether anything is already deployed and authenticate by typing `orchent depls`. Status messages describe the deployment with a unique identifier, a status and time information.

To monitor what is going on at the orchestrator level the command `"watch orchent depshow"`, along with the unique identifier can be used. While the deployment is still ongoing, the same output will be retrieved.

Step 2: INDIGO orchestrator will deploy the web app making it available to end users. As requested by the developer in the INDIGO YAML file, the web app is deployed on the edge machine.

Step 3: INDIGO orchestrator will allow the end-users to access the web app on the edge machine and to pass through the edge infrastructure. A compute node will also be allocated and used depending on user interaction.

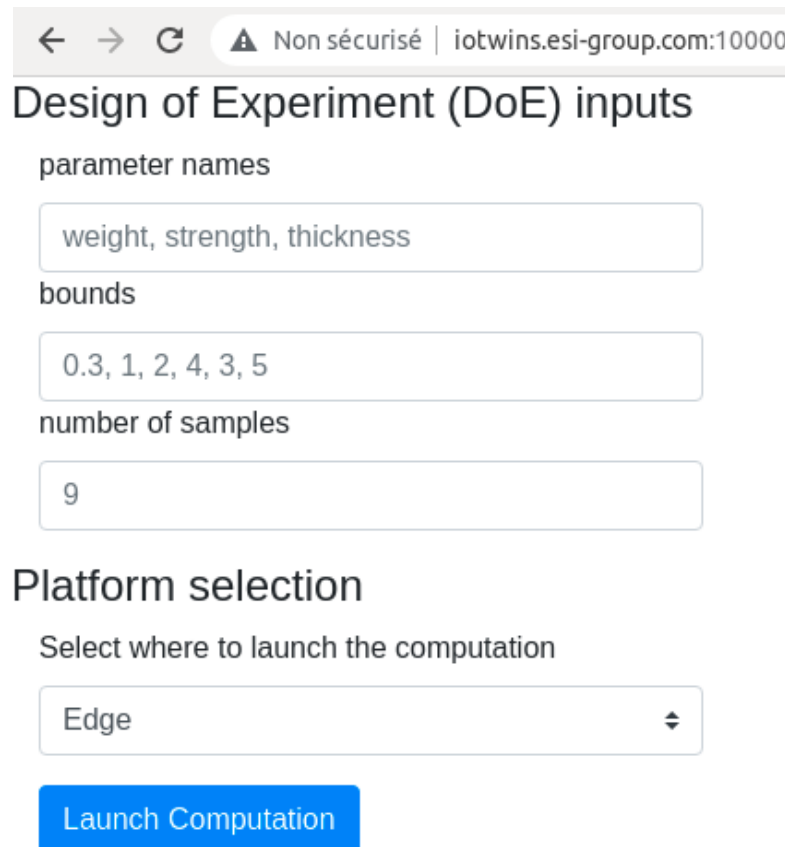
Step 4: after the services have been deployed and will be running, the deployment status change to CREATE COMPLETE and contains output information. In this case, two (2) endpoints URLs will be returned one for the web site and the other for the computations.

```
Deployment [11ebf042-e743-f46b-afc8-02421473c2d3]:
status: CREATE COMPLETE
creation time: 2021-07-29T07:59+0000
update time: 2021-07-29T08:00+0000
outputs:
{
  "compute": "http://iotwins.esi-group.com:10001",
  "website": "http://iotwins.esi-group.com:10000"
}
```

Figure 8: orchent depshow after completion

The web app implements a Design of Experiment computation and let the user enter input parameters (e.g. physical variables) associated with their bounds. The design is computed on a number of sample points entered by the user and the execution is performed on user's demand by clicking a Launch Computation button.

In the "Platform Selection" section the user will have the choice between "Edge" and "Cloud"; after filling the inputs of the dialog, the end-user may submit the run by clicking in on the button "Launch Calculation". The website will require interaction with the INDIGO orchestrator and will use Chronos to submit the jobs.



The screenshot shows a web browser window with the address bar displaying "Non sécurisé | iotwins.esi-group.com:10000". The main heading is "Design of Experiment (DoE) inputs". Below this, there are three input fields: "parameter names" with the value "weight, strength, thickness", "bounds" with the value "0.3, 1, 2, 4, 3, 5", and "number of samples" with the value "9". Below these inputs is a section titled "Platform selection" with the instruction "Select where to launch the computation". A dropdown menu shows "Edge" as the selected option. At the bottom, there is a blue button labeled "Launch Computation".

Figure 9: DoE web app

Platform selection

Select where to launch the computation

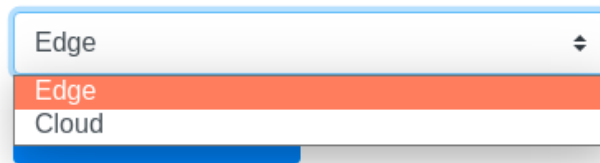


Figure 10: execution node selection

Step 5: INDIGO will submit the inputs and trigger a computation request on the edge machine.

Step 6: The jobs will be performed on the top of Chronos, the results sent to the web app, and displayed to the end user for further analysis and parameters tuning.

As a reference, Listing 1 implements a TOSCA template for a web server where the web server is a Node.js application inside a Docker container. Developing similar or more complex applications will not require editing the TOSCA template.

```
tosca_definitions_version: tosca_simple_yaml_1_0
imports:
  - indigo_custom_types: >-
    https://raw.githubusercontent.com
    /indigo-dc/tosca-types/master/custom_types.yaml
description: >
  TOSCA example template for long running service
topology_template:
  inputs:
    cpus:
      type: float
      description: Amount of CPUs for this service
      required: yes
      default: 0.2
    mem:
      type: scalar-unit.size
      description: Amount of Memory for this service
      required: yes
      default: 100 MB
    sla:
      type: string
      description: sla to use
      required: yes
  node_templates:
    marathon:
      type: tosca.nodes.indigo.Container.Application.Docker.Marathon
      properties:
        environment_variables:
          SLA: { get_input: sla}
        uris: []
      artifacts:
        image:
          file: iotwinsdemo/esi-services-demo:website
          type: tosca.artifacts.Deployment.Image.Container.Docker
      requirements:
        - host: docker_runtime
  docker_runtime:
```

```

type: toska.nodes.indigo.Container.Runtime.Docker
capabilities:
  host:
    properties:
      num_cpus: { get_input: cpus}
      mem_size: { get_input: mem}
      publish_ports:
        - protocol: tcp
          source: 1337
        - protocol: tcp
          source: 8000
policies:
  - deploy_on_specific_site:
      type: toska.policies.indigo.SlaPlacement
      properties:
        sla_id: { get_input: sla}
outputs:
  endpoint:
    value:
      concat:
        - "http://"
        - get_attribute : [ marathon, load_balancer_ips, 0 ]
        - ':'
        - get_attribute : [ docker_runtime, host, publish_ports, 0, target ]
compute:
  value:
    concat:
      - "http://"
      - get_attribute : [ marathon, load_balancer_ips, 0 ]
      - ':'
      - get_attribute : [ docker_runtime, host, publish_ports, 1, target ]

```

Listing 1: TOSCA yaml template for long running service like a web server

As a reference, Listing 2 implements a TOSCA template for a computation task where the task is a bash script within a Docker container. Depending on the computation, the template can be edited with more hardware resources.

```

tosca_definitions_version: toska_simple_yaml_1_0
imports:
  - indigo_custom_types: >-
      https://raw.githubusercontent.com/indigo-dc
        /tosca-types/master/custom_types.yaml
description: "Run DoE for IoTwinS."
metadata:
  display_name: DOE Service
  icon: >-
      https://indigo-paas.cloud.ba.infn.it
        /public/images/IoTwinS_Logo.png
topology_template:
  inputs:
    cpus:
      type: float
      default: 1
      description: Amount of CPUs for this job
      required: yes
    mem:
      type: string
      description: Amount of Memory for this job
      required: yes

```

```

    default: 100 MB
  api_host:
    type: string
    description: api host ip
    required: yes
  api_port:
    type: string
    description: api port
    required: yes
  sla:
    type: string
    description: sla to use
    required: yes
  node_templates:
    preprocessing_job:
      type: toska.nodes.indigo.Container.Application.Docker.Chronos
      properties:
        command: '/script/compute.sh'
        retries: 1
        environment_variables:
          API_HOST: { get_input: api_host }
          API_PORT: { get_input: api_port }
      artifacts:
        image:
          file: iotwinsdemo/esi-services-demo:doe
          type: toska.artifacts.Deployment.Image.Container.Docker
      requirements:
        - host: docker_runtime
    docker_runtime:
      type: toska.nodes.indigo.Container.Runtime.Docker
      capabilities:
        host:
          properties:
            num_cpus: { get_input: cpus }
            mem_size: { get_input: mem }
  policies:
    - deploy_on_specific_site:
        type: toska.policies.indigo.SlaPlacement
        properties:
          sla_id: { get_input: sla }

```

Listing 2: TOSCA yaml template for executing a computation task

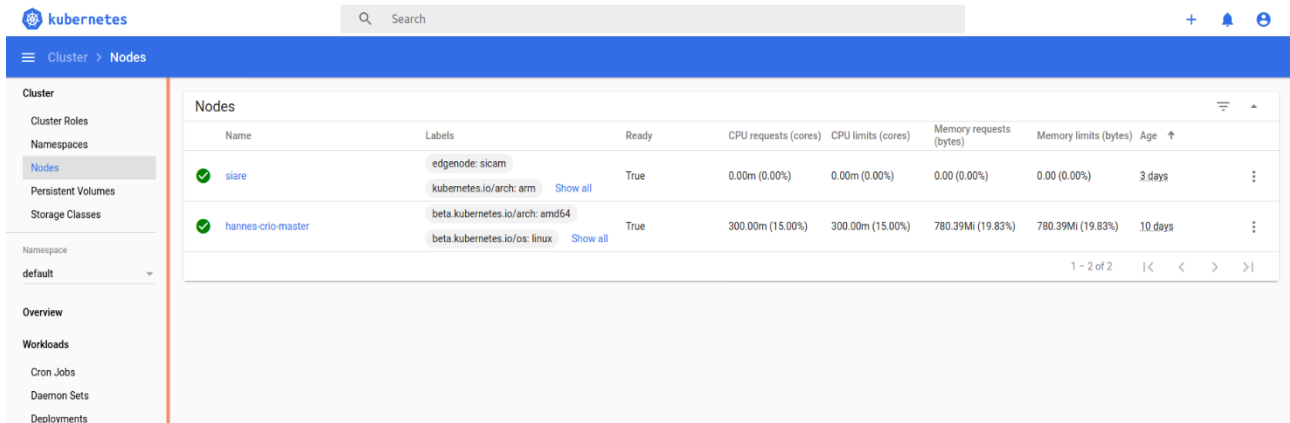
4.2 Integrated cloud and edge Level Service Management

This use-case implements the edge and cloud specification within an IoTwinS platform to provide an integrated software management which is used to create and customize applications for specific edge devices. In terms of software, Kubernetes is used for the application management on both the cloud level and the edge level by using the KubeEdge solution (for both solutions see D2.2).

In contrast to the example in the previous section, at most one person is involved in the described scenario: A *User* who interacts with the platform by defining system parameters, which are then automatically monitored by the system. The sequence of commands, however, is naturally equal to the previous example, only that the actors are different.

The system consists of a central Kubernetes installation. This can either be a bare metal installation, a custom solution by a cloud provider, or a local installation in a docker container. The edge devices are connected to

the Kubernetes System using the KubeEdge components CloudCore on the cloud level and EdgeCore on the edge level. Thus, the edge node becomes part of the Kubernetes cluster. Thereby the nodes are labelled to make them distinguishable from the other cluster nodes (see Figure 11 and Figure 13). The label is the edge node ID which directly correlates with the location of the edge device in the system (e.g., the transformer substation, or building part).



Name	Labels	Ready	CPU requests (cores)	CPU limits (cores)	Memory requests (bytes)	Memory limits (bytes)	Age
siare	edgenode: sicam kubernetes.io/arch: arm	True	0.00m (0.00%)	0.00m (0.00%)	0.00 (0.00%)	0.00 (0.00%)	3 days
hannes-crio-master	beta.kubernetes.io/arch: amd64 beta.kubernetes.io/os: linux	True	300.00m (15.00%)	300.00m (15.00%)	780.39Mi (19.83%)	780.39Mi (19.83%)	10 days

Figure 11: Dashboard of a Kubernetes cluster listing the contained nodes. Thereby the node named siare is labelled as an edge node.

```
ubuntu@clusterserver:~$ kubectl get nodes --show-labels
```

NAME	STATUS	ROLES	AGE	VERSION	LABELS
clusterserver	Ready	control-plane,master	140d	v1.20.2	beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux, kubernetes.io/arch=amd64,kubernetes.io/hostname=clusterserver, kubernetes.io/os=linux,node-role.kubernetes.io/control-plane=, node-role.kubernetes.io/master=
weidlingau	Ready	agent,edge	139d	v1.19.3-kubeedge-v1.5.0	beta.kubernetes.io/arch=arm64,beta.kubernetes.io/os=linux, kubernetes.io/arch=arm64,kubernetes.io/hostname=l2, kubernetes.io/os=linux,node-role.kubernetes.io/agent=, node-role.kubernetes.io/edge=, edgenode=weidlingau
hadersdorf	Ready	agent,edge	139d	v1.19.3-kubeedge-v1.5.0	beta.kubernetes.io/arch=arm64,beta.kubernetes.io/os=linux, kubernetes.io/arch=arm64,kubernetes.io/hostname=l2, kubernetes.io/os=linux,node-role.kubernetes.io/agent=, node-role.kubernetes.io/edge=, edgenode=hadersdorf
node1	Ready	<none>	140d	v1.20.2	beta.kubernetes.io/arch=amd64,beta.kubernetes.io/os=linux, kubernetes.io/arch=amd64,kubernetes.io/hostname=node1, kubernetes.io/os=linux

```
ubuntu@clusterserver:~$
```

Figure 12: Output of the kubectl command: List of nodes in Kubernetes cluster that contains one worker node (node1) and two edge nodes (weidlingau and hadersdorf).

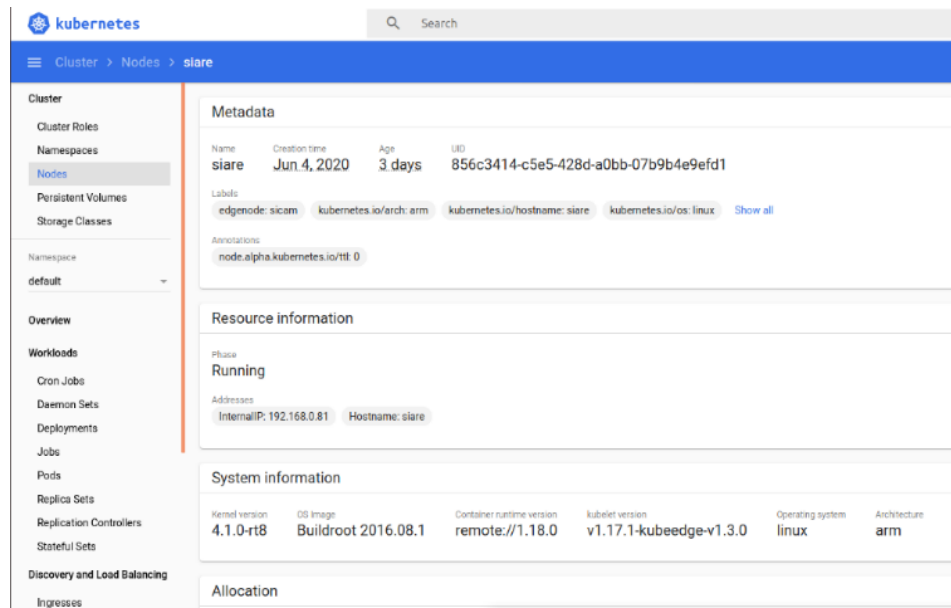


Figure 13: Details of an edge node in a Kubernetes cluster as displayed on the Kubernetes dashboard.

The system also offers a container registry (software repository) which stores the applications for both the cloud level and the edge level. In this scenario, AI models that detect errors and estimate values are stores in OCI containers as described in deliverable D5.2. Rolling out of AI models for edge devices is thus easily done by deploying containers to the edge devices using native Kubernetes mechanisms.

The sequence of steps needed to install a parameter monitoring solution to the system unfolds as follows (see also Figure 14 and DD.5 for the sensorless control scenario):

1. The *User* selects a part of the system which he/she wants to monitor or in which he/she wants the system to adhere to a set of thresholds.
2. The system then checks whether the according AI model for the monitoring and/or control, with respect to the systems setting (at the selected) location, is available in the container registry.
 - a. In case it is not available, the learning part in the system uses the data that was recorded for this part of the system (or starts recording) and creates the according AI component which is then packed into a container and stored specifically for the associated edge Device (see also D2.2 and D5.2)
3. The system then automatically creates a Kubernetes deployment descriptor or daemon-set descriptor, which then leads to an automatic installation of the location/setting specific application on the edge device. The descriptors can also contain all configuration values that are needed in addition to the edge device specific values in the container.
4. The *User* can then monitor the output of the deployed component directly or monitor the results of the control actions.

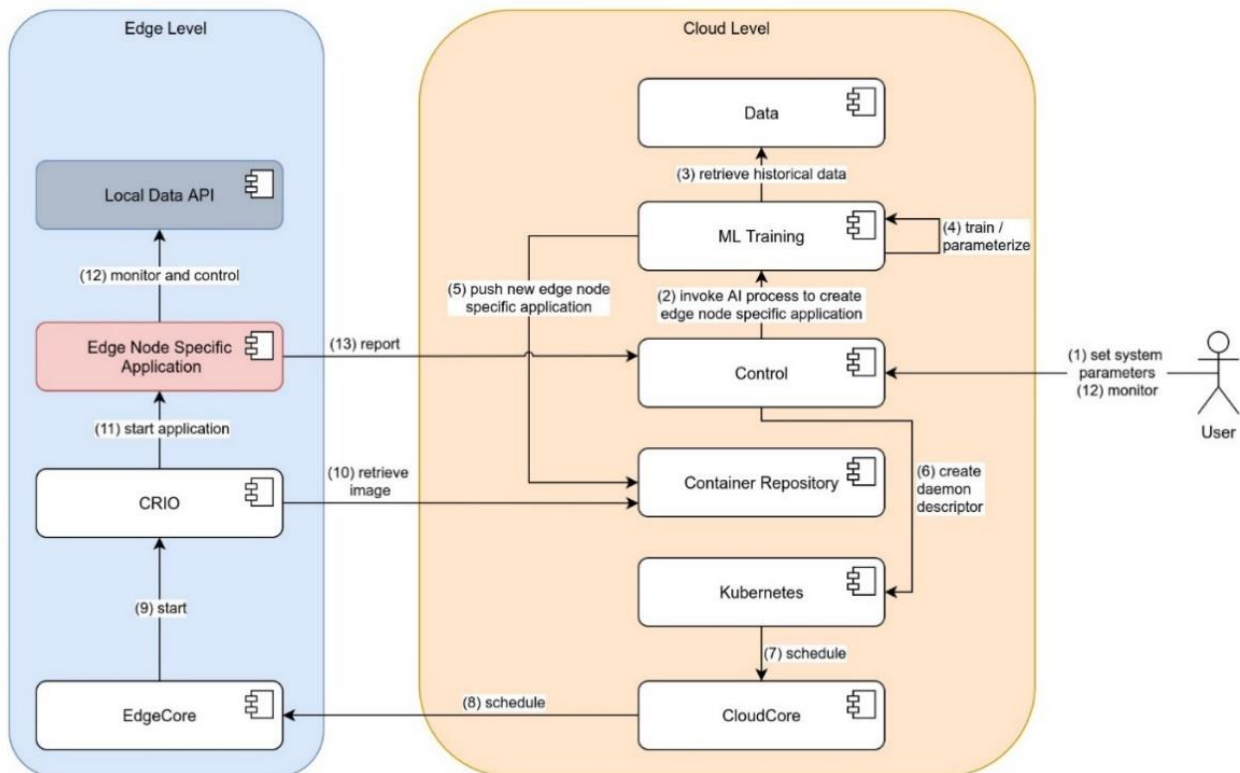


Figure 14: Processing sequence for the automatic deployment of AI based edge node specific components using Kubernetes and KubeEdge

Listing 3 shows a daemon-set descriptor template created by the system to automatically deploy the created application to the location specific edge device. Since the cloud level and the edge level are managed by the same Kubernetes instance the cloud level just needs to apply these descriptors to their own management backend, and the application is rolled out to the edge device. The following values are an example on how the automatic deployment process might fill the template (could also be used together with helm templates):

```
DeploymentID: "anomaly-detection-tsXXY",
AppID:      "anomaly-detection",
ImageID:    "anomaly-detection",
ModelID:    "2021-02-25-0002",
Selectors:  {"substationSetup": "TransformerXXY"},
Environment: {"LOCAL_BACKEND": "192.168.0.1"}

{
  "apiVersion": "apps/v1",
  "kind": "DaemonSet",
  "metadata": {
    "name": "{{.DeploymentID}}",
    "labels": {
      "app": "{{.AppID}}",
      "mlmodel": "{{.ModelID}}"
    }
  },
  "spec": {
    "selector": {
      "matchLabels": {
        "app": "{{.AppID}}"
      }
    }
  },
}
```

```

"template": {
  "metadata": {
    "labels": {
      "app": "{{.AppID}}",
      "mlmodel": "{{.ModelID}}"
    }
  },
  "spec": {
    "containers": [
      {
        "name": "{{.AppID}}",
        "image": "{{.ImageID}}",
        "env": {{joinList .Environment}}
      }
    ],
    "nodeSelector": {{joinObject .Selectors}}
  }
}

```

Listing 3: Kubernetes daemon-set descriptor template (JSON format) that can be used to automatically deploy AI models to the edge Level using KubeEdge.

The screenshots in **Figure 15** and **Figure 17** show the state of the pods (a grouping of containers¹⁴) before and after the deployment of the edge node specific application using the shown daemon-set descriptor (see also **Figure 16**). Finally, **Figure 18** shows the description of the created pod.

```

ubuntu@clusterserver:~$ kubectl get pods --selector=app=anomaly-detection --show-labels
No resources found in default namespace.
ubuntu@clusterserver:~$

```

Figure 15: Output of kubectl command for listing all anomaly-detection apps before the app was rolled out to the edge device.

```

ubuntu@clusterserver:~$ kubectl get daemonsets

```

NAME	DESIRED	CURRENT	READY	UP-TO-DATE	AVAILABLE	NODE SELECTOR	AGE
anomaly-detection	1	1	1	1	1	substationSetup=TransformerXXY	11s

```

ubuntu@clusterserver:~$

```

Figure 16: Output of kubectl command for listing all daemon descriptors after the template was applied.

```

ubuntu@clusterserver:~$ kubectl get pods --selector=app=anomaly-detection --show-labels

```

NAME	READY	STATUS	RESTARTS	AGE	LABELS
anomaly-detection-6bd4fd7757-pslkg	1/1	Running	0	8m	app=anomaly-detection,mlmodel=2021-02-25-0002,pod-template-hash=6bd4fd7757

```

ubuntu@clusterserver:~$

```

Figure 17: Output of kubectl command for listing all anomaly-detection apps after the app was rolled out to the edge using a daemon the descriptor template.

¹⁴ See also <https://kubernetes.io/docs/concepts/workloads/pods/>

```

ubuntu@clusterserver:~$ kubectl describe pod anomaly-detection-6bd4fd7757-pslkg
Name: anomaly-detection-6bd4fd7757-pslkg
Namespace: default
Priority: 0
Node: weidlingau/172.161.149.135
Start Time: Fri, 25 Jun 2021 07:39:52 +0000
Labels: app=anomaly-detection
        pod-template-hash=6bd4fd7757
Annotations: cnl.projectcalico.org/podIP: 10.0.166.137/32
             cnl.projectcalico.org/podIPs: 10.0.166.137/32
Status: Running
IP: 10.0.166.137
Controlled By: ReplicaSet/anomaly-detection-6bd4fd7757
Containers:
  anomaly-detection:
    Container ID: cri-o://f457a769558c425cbf5a046f9474f1d2a4983aaf169e6dd651ce39b4cc9f7511
    Image: 171.161.149.136/anomaly-detection:1.0
    Image ID: 171.161.149.136/anomaly-detection@sha256:5678ad5b83a5c24cf9e784450709c9daf6c233d0fcddd93fea882eb722c3b64
    State: Running
      Started: Fri, 09 Feb 2021 16:36:53 +0000
    Ready: True
    Restart Count: 0
    Environment:
      LOCAL_BACKEND: 192.168.0.1
Conditions:
  Type              Status
  Initialized       True
  Ready             True
  ContainersReady   True
  PodScheduled      True
QoS Class: BestEffort
Node-Selectors: substationSetup=TransformerXXY
Tolerations: node.kubernetes.io/not-ready:NoExecute op=Exists for 300s
             node.kubernetes.io/unreachable:NoExecute op=Exists for 300s
Events:
  Type     Reason      Age   From              Message
  ----     -
  Normal   Scheduled   22s   default-scheduler Successfully assigned default/anomaly-detection-6bd4fd7757-pslkg to weidlingau
  Normal   Pulled      22s   kubelet           Container image "171.161.149.136/anomaly-detection:1.0" already present on machine
  Normal   Created     22s   kubelet           Created container anomaly-detection
  Normal   Started     22s   kubelet           Started container anomaly-detection
ubuntu@clusterserver:~$

```

Figure 18: Output of kubectl command for a pod created based on the described daemon set descriptor

5 Summary of Test-bed implementation

This section is intended to give an overview of different approaches of platform implementation at selected test-beds. The descriptions are deliberately short as extensive documentation of test-bed's activity is available in the dedicated deliverables of the respective work packages.

5.1 Categories of platform implementation

Overall platform adoption can be categorized in three different levels:

- Fully Integrated

Fully integrated test-bed implementations have their edge and cloud layer managed by the Indigo PaaS Orchestrator. This level includes the replicability test-beds by design.

- Partially Integrated

Partially integrated test-beds mostly have their own solutions for the edge layer but orchestrate cloud services with the help of the Indigo reference implementation.

- Autonomic Implementation

Test-beds in this category mostly follow the proposed IoTwinS architecture, but implement edge and cloud services on their own, without integration in the reference implementation, which is based on Indigo PaaS Orchestrator

5.1.1 Fully Integrated

5.1.1.1 Test-bed N.8: CETIM pre-industrial prototype on smart manufacturing

Test-bed 8 will showcase a pre-industrial prototype that is somehow pre-trained for the manufacturing sector and will provide a set of steps to be followed to ground/refine the prototype with different settings. TB 8 will exploit both Edge and Cloud infrastructure/services provided by the IoTwinS platform to implement the distributed digital twins that will govern the manufacturing process. The IoT level encompassed in the IoTwinS architecture will be emulated via a proprietary IoT gateway programmed to provide data streams to the Edge. The diagram below represents a schematization of the data stream flow and operations throughout the IoT-Edge-Cloud computing chain:

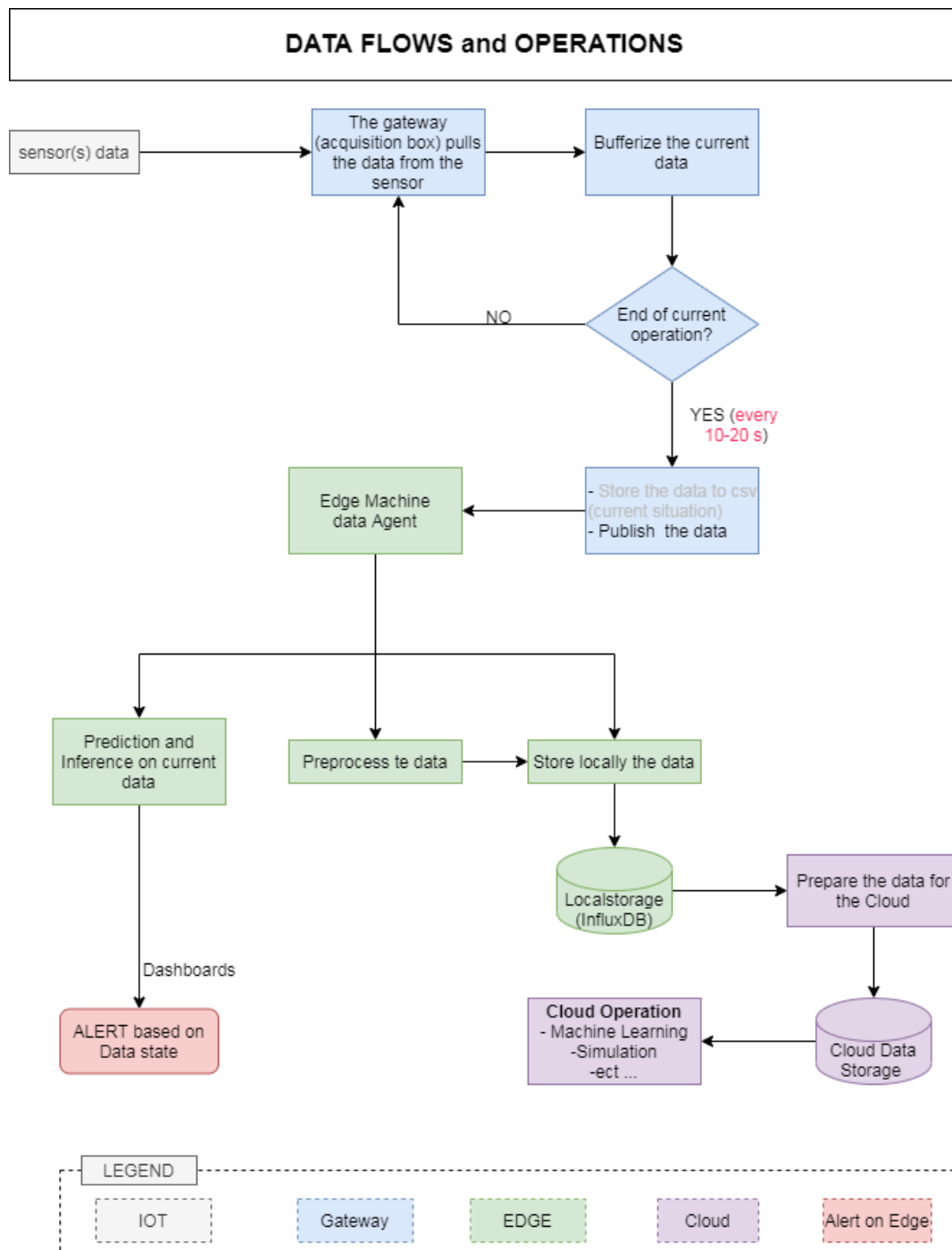


Figure 19: TB8 data stream flow and operations

Basically, Edge is exploited to pre-process data and set-up a monitoring application that infers/predict potential failures and fire alarms. In the Cloud, machine learning and simulation tools are deployed.

In Figure 20, the scheme of services implemented at the IoTwinS Edge level is depicted.

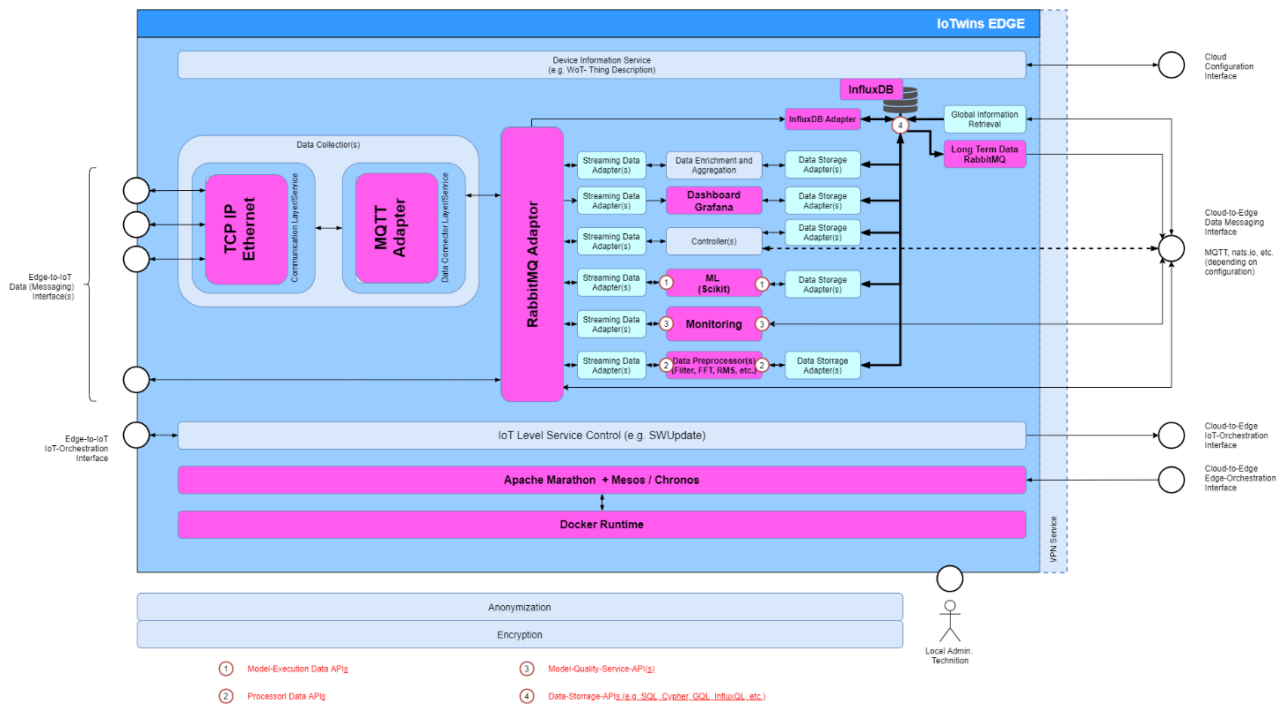


Figure 20: TB8 architecture implementation at Edge level

The following are activities that are being carried out integrate Test-bed 8 with the IoTwinS platform at Edge level:

- Installation of the Apache-Mesos runtime on the Edge ESI node for development purpose
- Configuration and deployment of services
 - MQTT broker is used to spread data incoming from sensors and is used as notification tool between services.
 - Temporary Databases (InfluxDB and PostgreSQL) are used to store sensors information and data.
 - A service used to write data incoming from MQTT to Databases.
 - A service used to push data on the Cloud.
- Deployment and test of the chain of services
 - Sensor data generator from Cetim files.
 - Data transfer via MQTT.
 - Data reception and storage in InfluxDB and PostgreSQL.
- Coming later
 - Installation of the Apache-Mesos runtime on the Edge hosted by Cetim.
 - Grafana should be used to display dashboards.
 - Machine learning algorithm that uses generated models and incoming data to compute alarms.

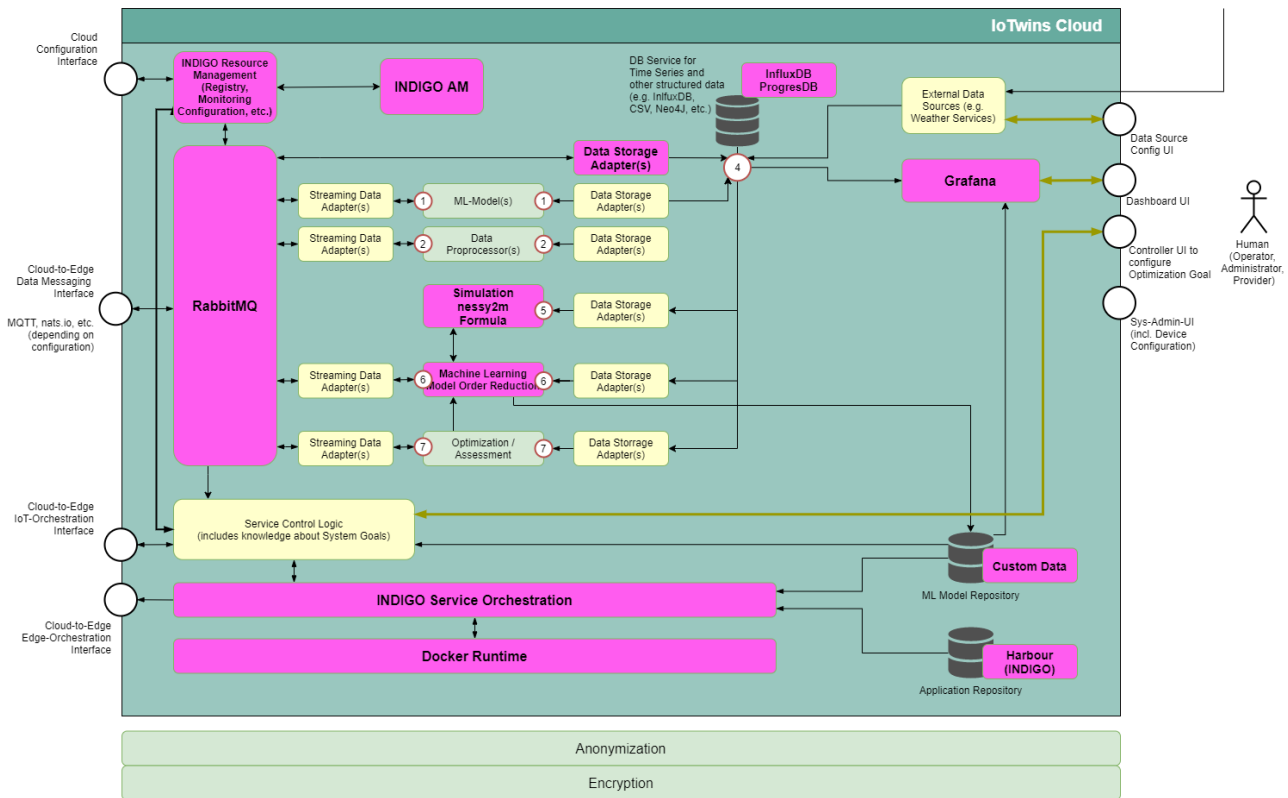


Figure 21: TB8 architecture implementation at Cloud level

The following are activities that have being carried out integrate Test-bed 8 with the IoTwinS platform at Cloud level:

- Configuration and deployment of services
 - MQTT broker is used as a notification tool between services.
 - Website is the user interface used to interact with services deployed on the edge and on the cloud. It uses the INDIGO IAM account to manage user login.
 - Permanent Databases (InfluxDB and PostgreSQL) are used to store information and data sensor.
 - DoE (Design of experiment) algorithm is used to generate data for simulations.
 - Freecad, a CAD tool used to visualize and modify meshes. It also prepares data for next simulations using DoE results.
 - Nussy is the simulation tool that will compute information to help the user to create the best initial configuration. Simulation outputs will also be used by the Model Order Reduction algorithm.
 - Machine learning algorithm that uses stored data to create models that will be used on the edge.
- Coming later
 - Model Order Reduction algorithm.
 - Installation of the Apache-Mesos runtime on the Edge Cetim node.

5.1.1.2 Test-bed N.10: INFN and BSC test-bed on EXAMON deployment (INFN)

The EXAMON IoTwinS deployment of Test-bed n.6 will be replicated on two data centers of INFN and BSC, with the purpose of identifying a common methodology for the monitoring infrastructure to be deployed on

different contexts. The two situations are heterogeneous both in terms of data collection and in terms of infrastructure upon which the services will be deployed.

On the INFN side, the collection of tens of TB of data is already performed by the production monitoring system at INFN-CNAF. The test-bed will cover the integration of part of the Examon services into the current infrastructure. The analysis will be adapted from the anomaly detection applications provided by WP3 to analyse log files as well as metrics and time series.

The containerization process of such analysis applications has been ongoing so that they could be instantiated via the PaaS Orchestrator. The metrics data collected by the production monitoring infrastructure has been connected to an analytical platform based on Jupyterhub and Spark, orchestrated by Kubernetes, to prototype and WP3 anomaly detection applications.

In parallel, we setup a tenant on the INFN Cloud and the PaaS system to instantiate the EXAMON components via Orchestrator. The integration of the Examon services in the INFN-CNAF monitoring system will require a rework of the current production infrastructure. A possible solution will be to configure clients to collect the monitoring data from few selected test machines using a Telegraf agent¹⁵, the agent will redirect the data both to the current production infrastructure and to the Examon service deployed on Cloud.

On BSC side, Examon will be deployed from scratch, without the restriction posed by a pre-existing monitoring infrastructure. The deployment will be performed on an unconventional low-power cluster. The cluster is based on NVIDIA Jetson development boards.

The first step is the deployment of the monitoring sensors to collect the metric data. Examon sensors were developed to run on specific Intel architectures, while the Jetson chipset are based on ARM processors. For these reasons the sensors must be adapted for the new architecture. An alternative approach would involve the use of the Telegraf agent to extract metrics also from the Jetson cluster. The data collected will be redirected towards the Examon deployment on INFN Cloud, using the Examon's MQTT broker as data channel.

¹⁵ Influxdata Telegraf: <https://www.influxdata.com/time-series-platform/telegraf/>

To be able to also replicate the analytical approaches of TB6 on the data from BSC and INFN, a common set of metrics from the different infrastructure will be selected. WP3 anomaly detection applications and algorithms will be performed on both set of data to allow a useful comparison of the resulting infrastructure and the monitoring approach.

Test-bed 12 goal is to define an innovative business model based on the intensive use of the IoTwins platform. In particular it focuses on exploiting the IoTwins solution in a PaaS fashion to take full advantage of HPC resources and Big Data services offered by Cloud computing solutions. It also makes use of artificial intelligence techniques (neural networks) both at Cloud and at Edge/IoT level in order to optimize plant operations and production.

Test-bed 12 will implement a SaaS on top of services/tools provided by the IoTwins platform (thus acting as a PaaS). For the purpose of integration, test-bed 12 will exploit both Cloud and Edge functionality offered by the IoTwins platform. No support at IoT level is required (proprietary sensors/data gathering tools are employed at the customer side).

In Figure 23 the integration of test-bed 12 with the Cloud end of the IoTwinS platform (the Indigo PaaS) is depicted.

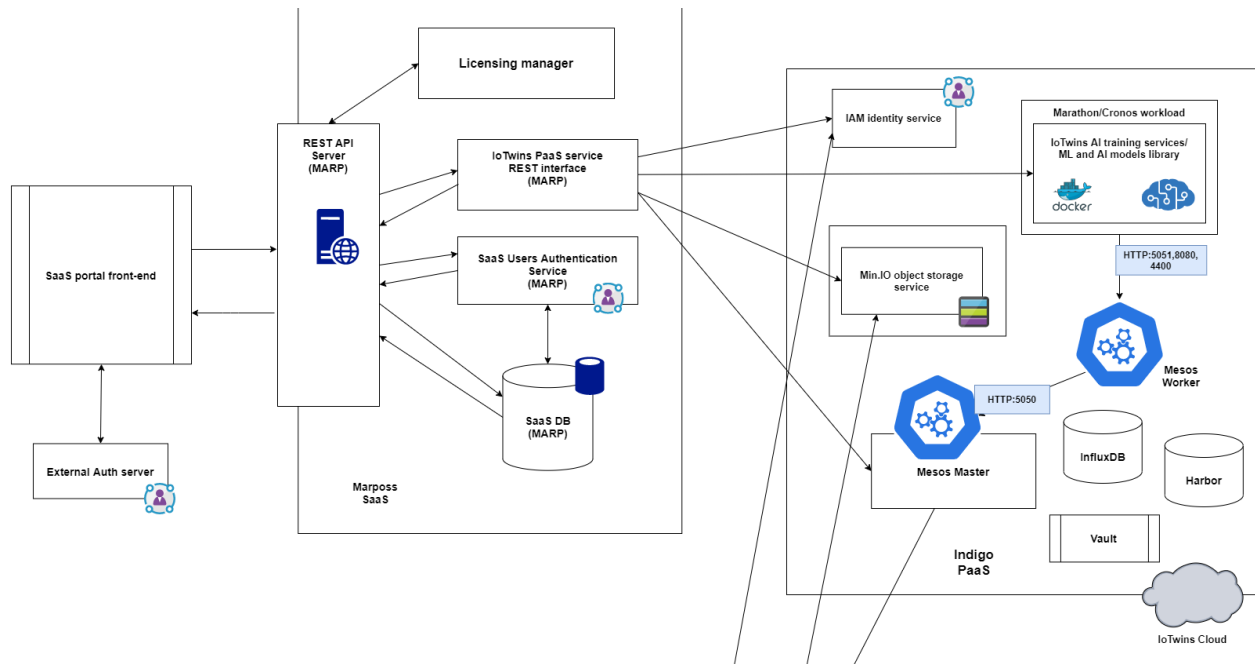


Figure 23: TB12 architecture implementation at Cloud level

The following activities were successfully carried out to pursue the above-mentioned integration with the Indigo Paas:

- Indigo IAM accounts activations
- MinIO server instance activation
- InfluxDB instance activated and running on test-bed 12 tenant
- Vault instance activated on test-bed 12 tenant and integrated with IAM and MinIO for secrets management
- Docker containers deployed correctly on Harbor for a first dummy version of the services required by test-bed 12

Regarding our SaaS implementation, it is divided into three logical components: a web UI front-end (still under design), a business logic and data management back-end and an authentication and authorization server owned by MARP (for customers authentication). The back-end (REST API Server) is fully integrated with IoTwinS' services and a first set of back-end REST APIs is under testing. These APIs are designed to decouple the UI front-end from the business logic that uses the Indigo PaaS deployment tool. They allow through Indigo PaaS both to start AI training jobs on the cloud tenant and also to control, thanks to Indigo's support for orchestrating services on edge resources, the execution of neural networks on edge nodes.

In Figure 24, the integration of test-bed 12 with the Edge end of the IoTwinS platform is depicted.

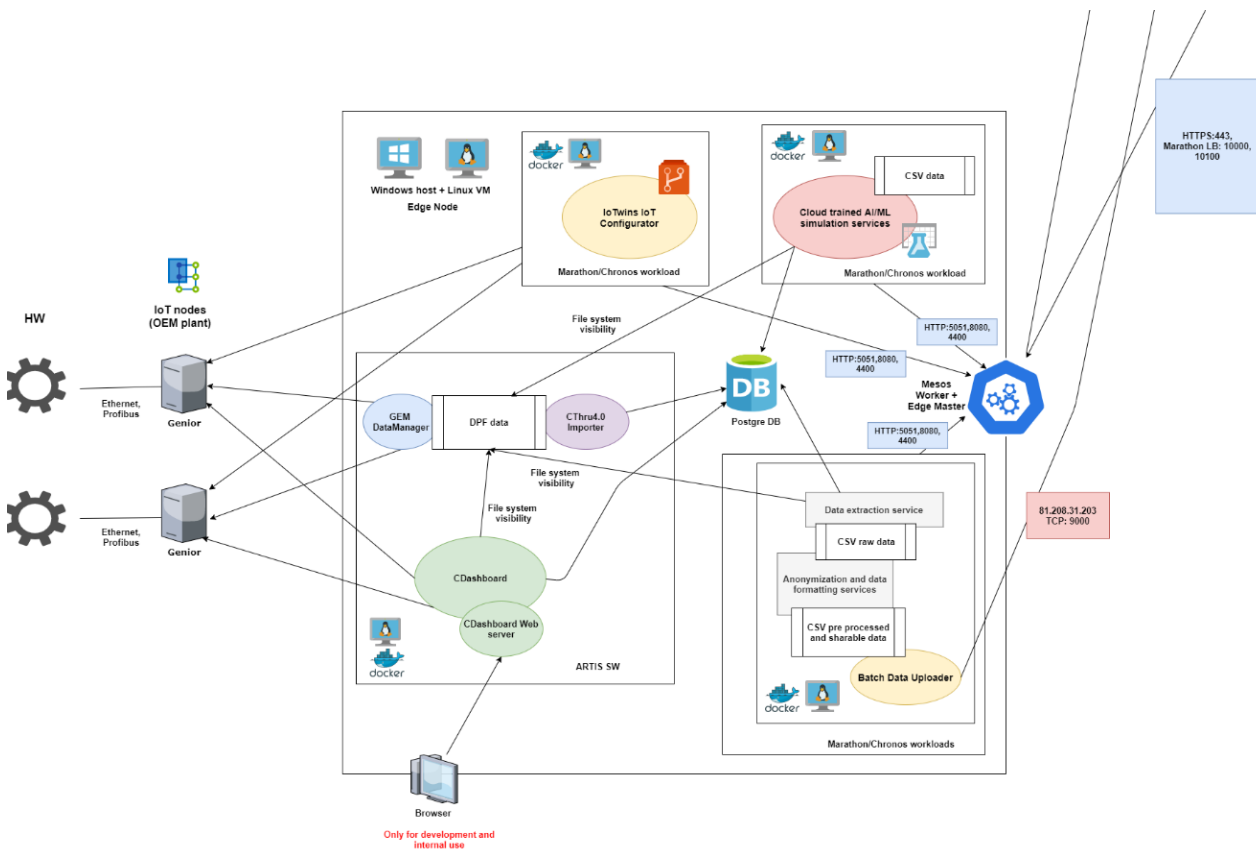


Figure 24: TB12 architecture implementation at Edge level

The following activities were successfully carried out to integrate test-bed 12 with the Edge end of the IoTwinS platform:

- Installation of the Apache-Mesos runtime on the Edge node
- A first version of a TOSCA template for edge services has been developed (further testing is needed)
- Testing of the edge node services (data anonymization, encryption and upload). Data are correctly anonymized and loaded into the cloud instance of InfluxDB
- Installation of RabbitMQ as local message broker for data dispatching to edge data anonymization, encryption and upload services

5.1.2 Partially Integrated

5.1.2.1 Test-bed N.1: Windfarm predictive Maintenance

The architecture for the testbed 1 has been defined (Figure 26) and installed into wind turbines in real field application. The architecture includes an edge device with larger computational capability higher than standard edge machine.

The IoT components are represented by the HW sensors operating at high frequency level and by Scada wind turbine generator (WTG) data available at low frequency level.

All these data are stored in the INFN cloud.

5.1.2.2 Test-bed N.2: Machine tool spindle predictive behaviour

Test-bed 2 follows a similar approach to Test-bed 1. The edge solution is based on TTEch's NERVE platform, which is aligned to the reference architecture. Cloud services are deployed in the INFN cloud site and data is being pushed from the edge services there. Further documentation to the adaption of Test-bed2 can be found in D4.3.

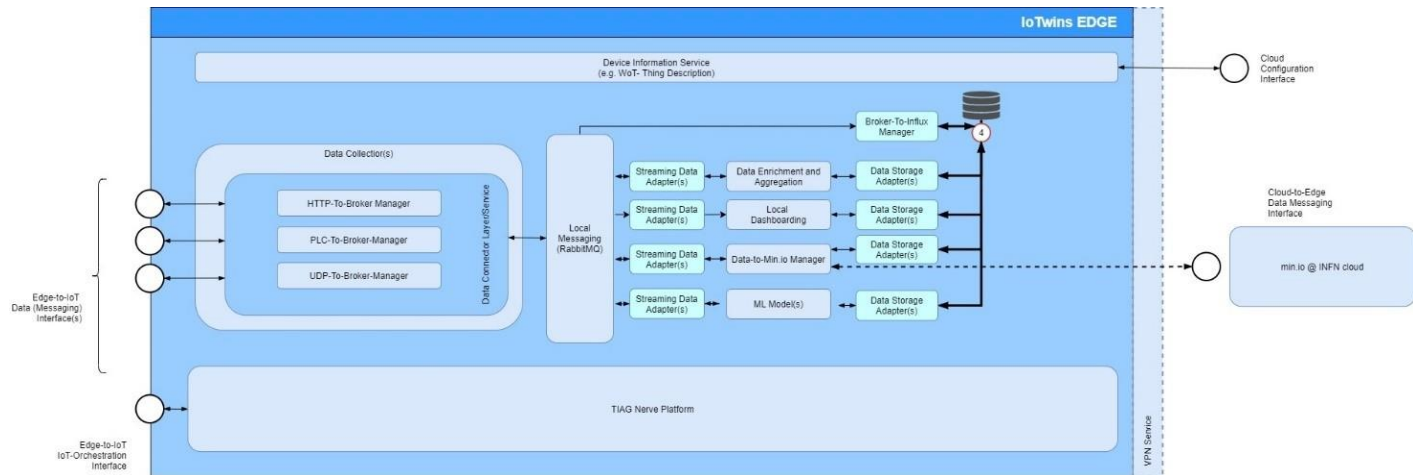


Figure 27: TB2 Edge Implementation

5.1.3 Autonomic implementation

5.1.3.1 Test-bed N.5: NouCampNou – Facility management and maintenance (FCB + BSC)

Football Club Barcelona's infrastructure, prior to IoTwinS project, was composed by a pseudo-edge layer, without iot-edge management, strictly intended to collect data from controllers and sensors, for its later shipment to the FCB's DATA department, that integrated all data in an internal Data Warehouse.

Along with IoTwinS project development, FCB tunned and opened an API, in order to provide access to its data to TB5's partners, f. i. BSC. The TB's architecture was extended, by adding BSC's analysis and simulation capabilities, as cloud layers, as seen in the figure below.

CURRENT ARCHITECTURE

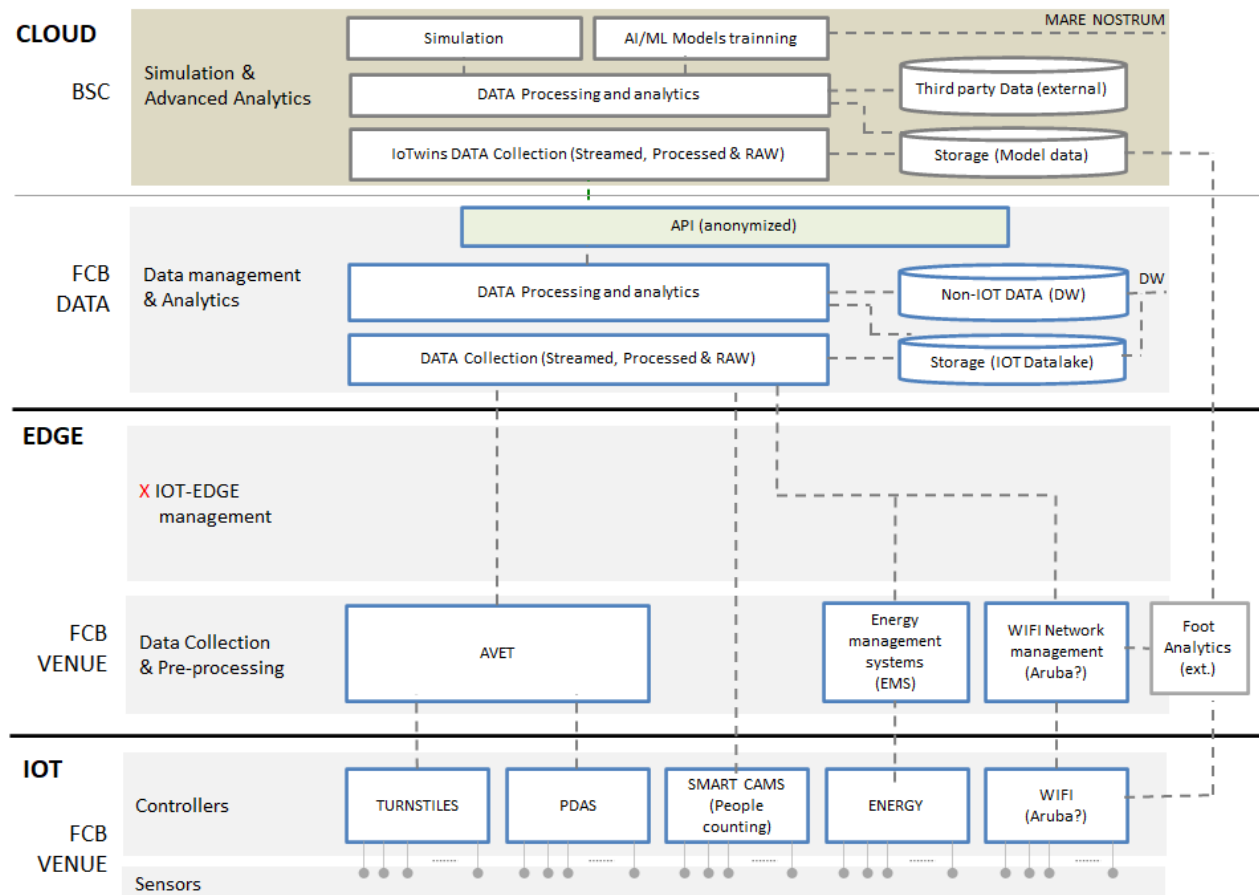


Figure 28: TB5 Current Architecture

One of the main objectives of this TB is to integrate further smart cameras, concretely, edge camera devices, that already provide computing capabilities, thanks to an integrated GPU device. In order to enable the communication of this edge devices, the test-bed is intended to integrate IoTwinS connection mechanisms, from Edge to Cloud and from Cloud to Edge. Moreover, it intends to add further devices, not just sensors, but also actuators, that are expected to benefit from real-time data and simulations, creating a virtuous cycle. The future expected architecture can be seen in the figure below.

FUTURE EXPECTED ARCHITECTURE

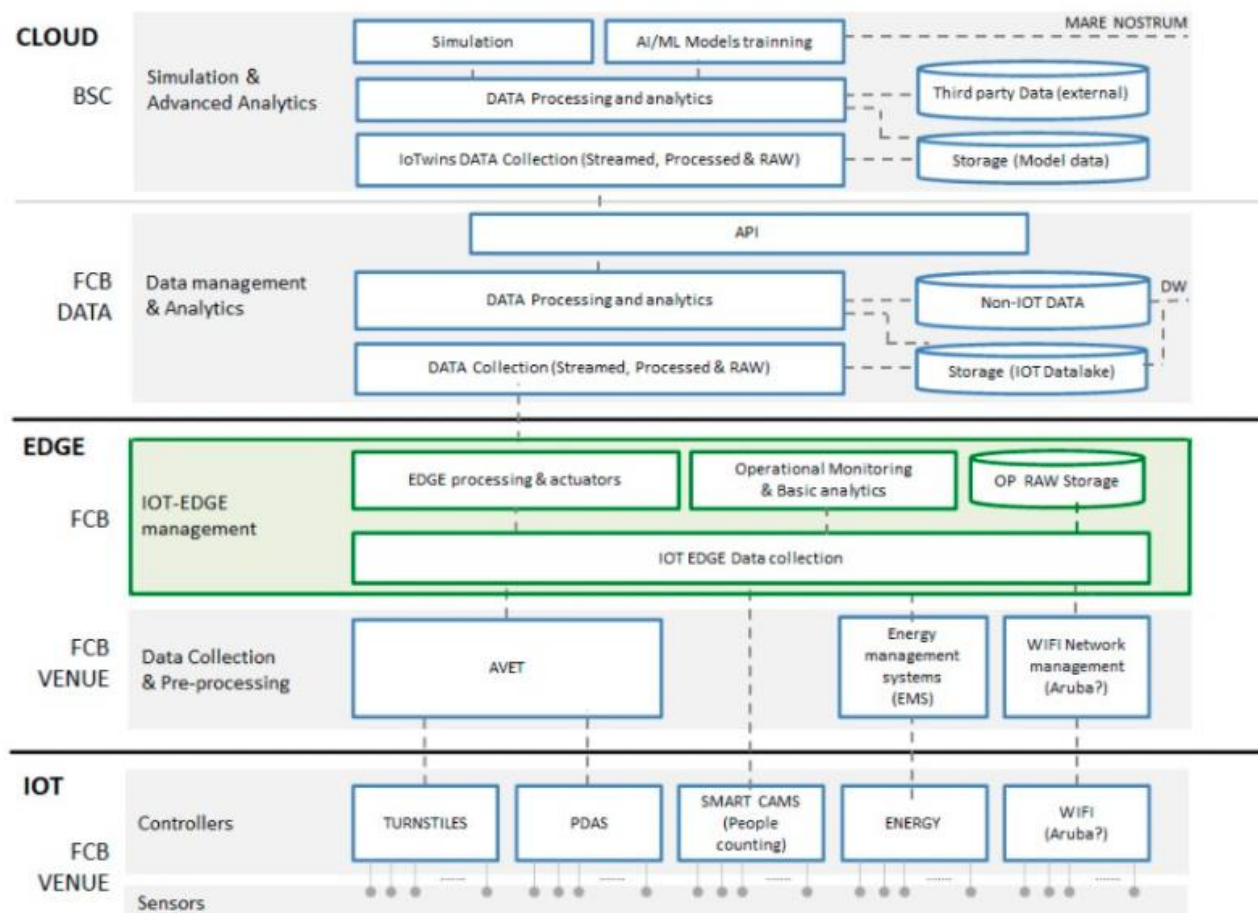


Figure 29: TB5 Future Architecture

Thus, TB5 is expected to extend FCB's current architecture, by complementing it with further improvements that IoTwinS project and its partners may provide.

5.1.3.2 Test-bed N.6: Large IT infrastructure management (CINECA)

The architecture for the CINECA test-bed was already partially established prior to the beginning of the IoTwinS project. In this particular test-bed, there is no actual edge component (the computing nodes can be seen as having the functional role of edges but with a much larger computational capability than standard edge machines); the IoT components are represented by the HW sensors operating at low-level on the computing nodes and the cloud part is represented by the High-Performance computing capability granted by the computing nodes.

The architecture of the TB06 consists of a framework for data center monitoring and maintenance called EXAMON. At the lowest level, there are components to read the data from several sensors scattered across the system and deliver them, in a standardized format, to the upper layers of the stack. The low-level data gathering modules have access to hardware resources and can also retrieve information from the job scheduler deployed on the supercomputer and other diagnostic tools used by system administrators.

The infrastructure relies on the MQTT protocol for the communication. The publishers are the SW components that collect sensors data and delivery it to the transport layer. The publisher worker contains two objects: the sensor API that implements all sensor-specific routines and the interfaces to EXAMON

transport layer. The MQTT API implements all the communication routines and acts as an abstraction layer between the sensor data buffer and the low-level routines provided by the MQTT protocol. For each different type of publisher we specialize the sensor API implementing the specific sensors routines while the MQTT API remains almost the same.

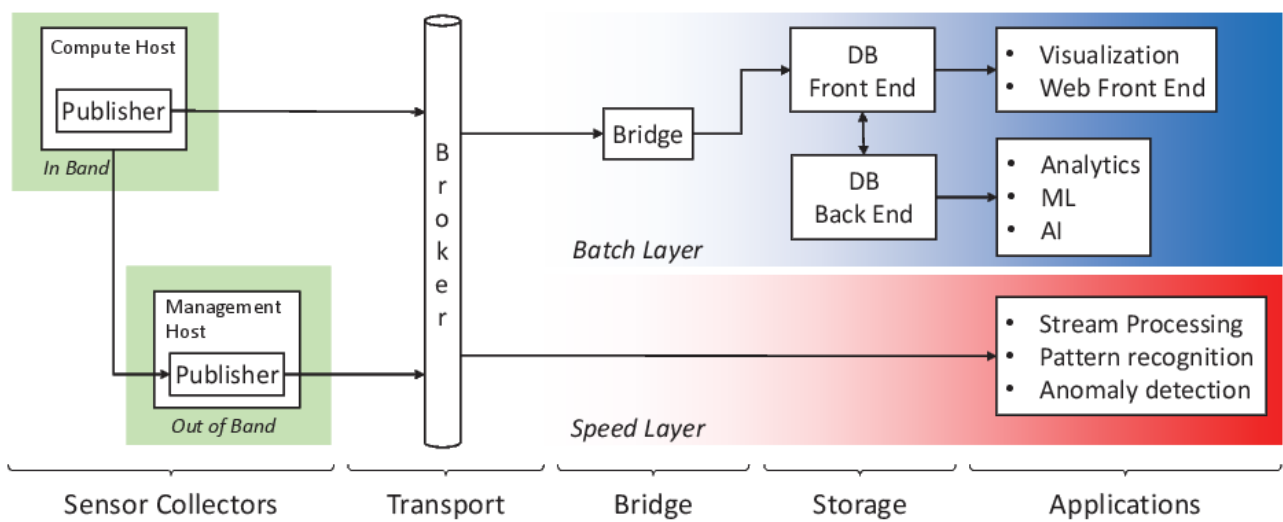


Figure 30: TB6 Architecture

On top of the low- and mid-layers, respectively devoted to data gathering and transportation and storage, the higher layer of EXAMON provides to the users (e.g., system admins and accountants and facility owners) a set of functionalities mainly aimed at predictive maintenance tasks and to optimize the management of the facility and the data center. These functionalities are collectively referred to as EXAMON-X. The functionalities are explicitly tailored to HPC requirements, such as handling large amount of heterogeneous data. EXAMON-X users can use the proposed functionalities independently to perform simple tasks (e.g., retrieving data from the DB or processing it) or combining them to perform macro-activities composed of a sequence of functions (e.g., anomaly prediction with a DL model).

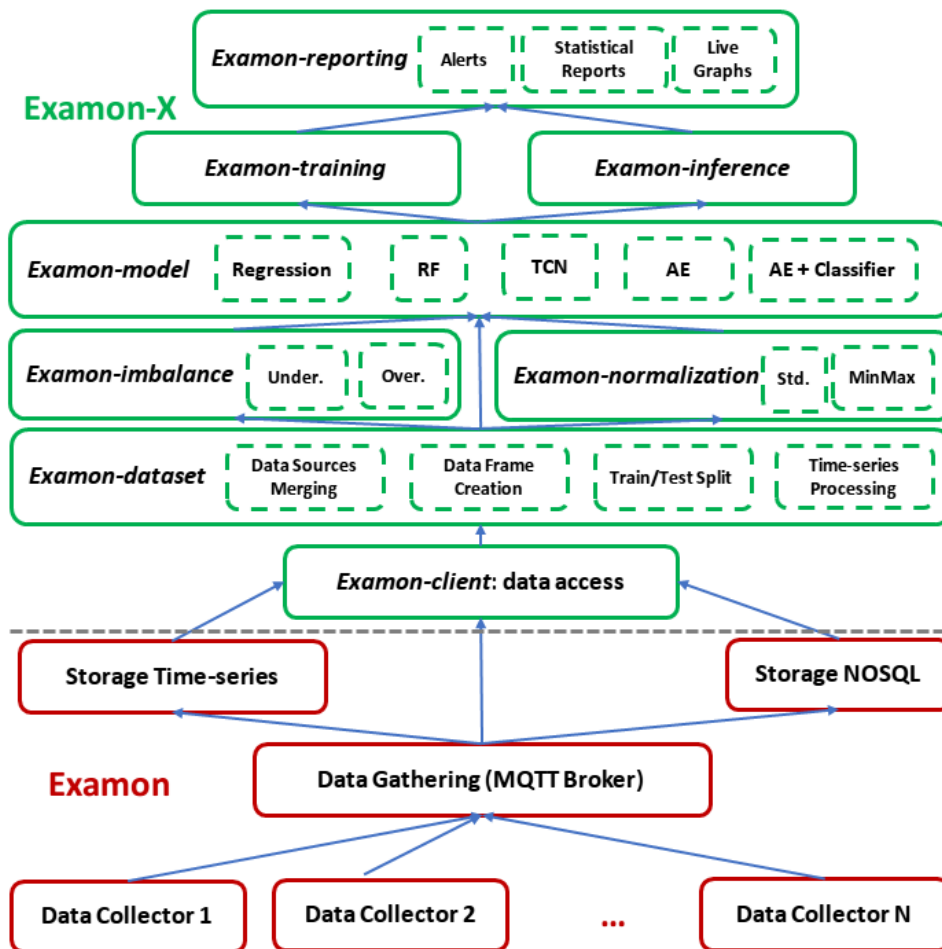


Figure 31: TB6 Examon

We grouped the proposed functionalities into eight different subgroups, ranging from data access from the EXAMON lower-layers to a dedicated client to the creation of complex deep/machine learning models for different tasks (e.g., power/thermal prediction, anomaly detection, etc.), passing through data manipulation, data set creation, handling of imbalanced data sets, data normalization and so on. These functionalities are at the moment run as a set of Python script and daemons running “bare-metal” on the HPC resources (in a sense, the monitored data centre is used to analyse itself), as this is the standard practice in high-performance computing contexts, where virtualization is typically not used not to introduce unnecessary overhead. The possibility to deploy such high-level functionalities as Docker services is currently under investigation.

5.1.3.3 Test-bed N.7: Smart grid – power quality management (SAGOE, SAG)

Three examples of autonomic test-bed deployments are listed below.

The architecture of the Smart Grid test-bed follows the IoTwinS architecture and implements the services for the on-premise backend (cloud level) and the field devices (edge level) using the Kubernetes and KubeEdge technologies described in deliverables D2.1, D2.2 and D2.3. In this way the test-bed is an autonomic implementation at the time being. However, a Kubernetes adapter is planned to be implemented for the IoTwinS cloud level (see Figure 33). Using this adapter, the testbed would become fully integrated.

As described in deliverable D5.3 there are different use cases implemented for test-bed 7. Figure 32 shows the detailed view of the software components needed to implement the sensorless control use-case. As depicted the implementation integrated different edge hardware platforms (Siemens SGW 1050, Siemens CP8050, Siemens IOT2050 and Wago PFC 200). This becomes possible by using the KubeEdge technology that was cross compiled for the different CPU architectures.

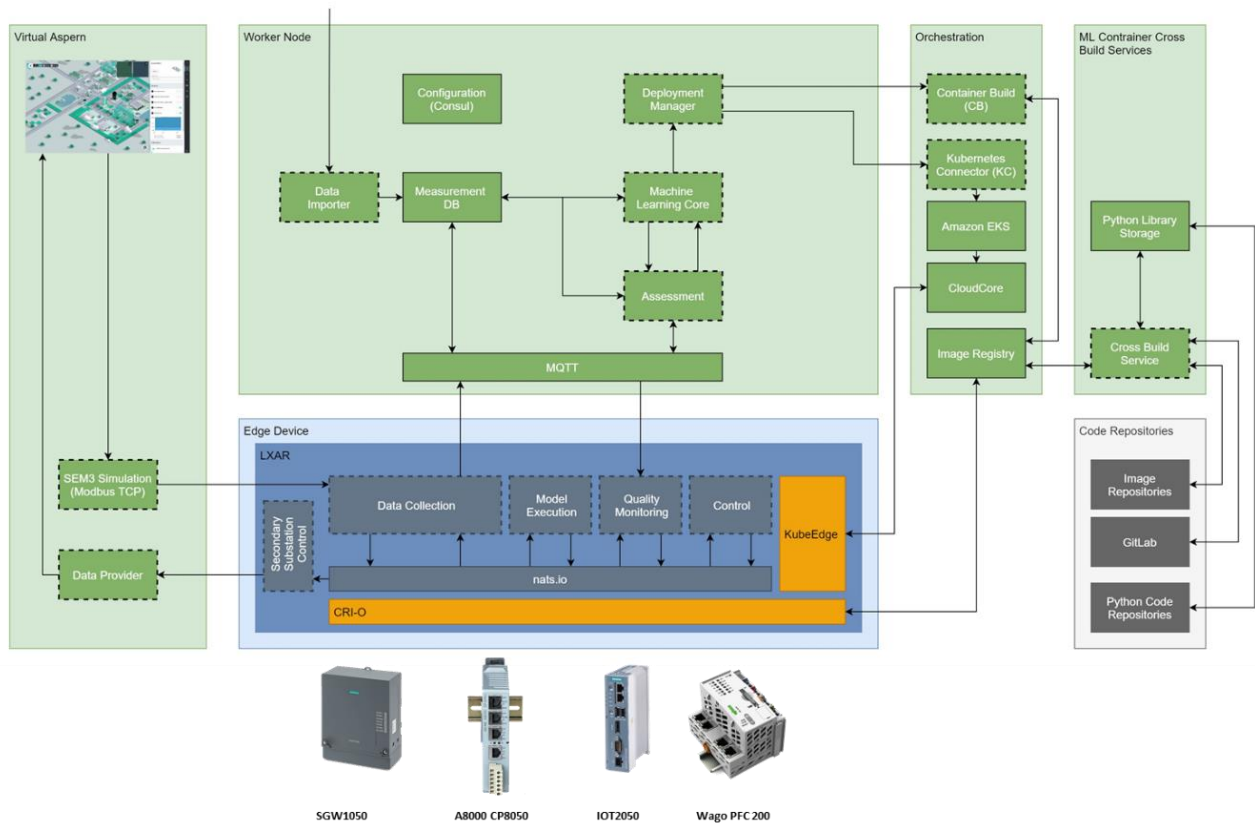


Figure 32: Test-bed 7 components to implement the "sensorless control" usecase as described in D5.3

While Figure 32 shows the control and data flow for the sensorless control use-case, the standard components that are used in all implemented use-cases are shown in Figure 34 and Figure 35. These figures show, that the implementation of the test-bed follows the IoTwinS architecture. An example of how services can be deployed to the edge level is described in section 16. For communication between edge and cloud level test-bed 7 evaluated OPC UA Pub/Sub, MQTT, and Kafka. It is important for the targeted customers that the data connection to the backend is flexible and can be adapted to the customers' needs and existing software environment.

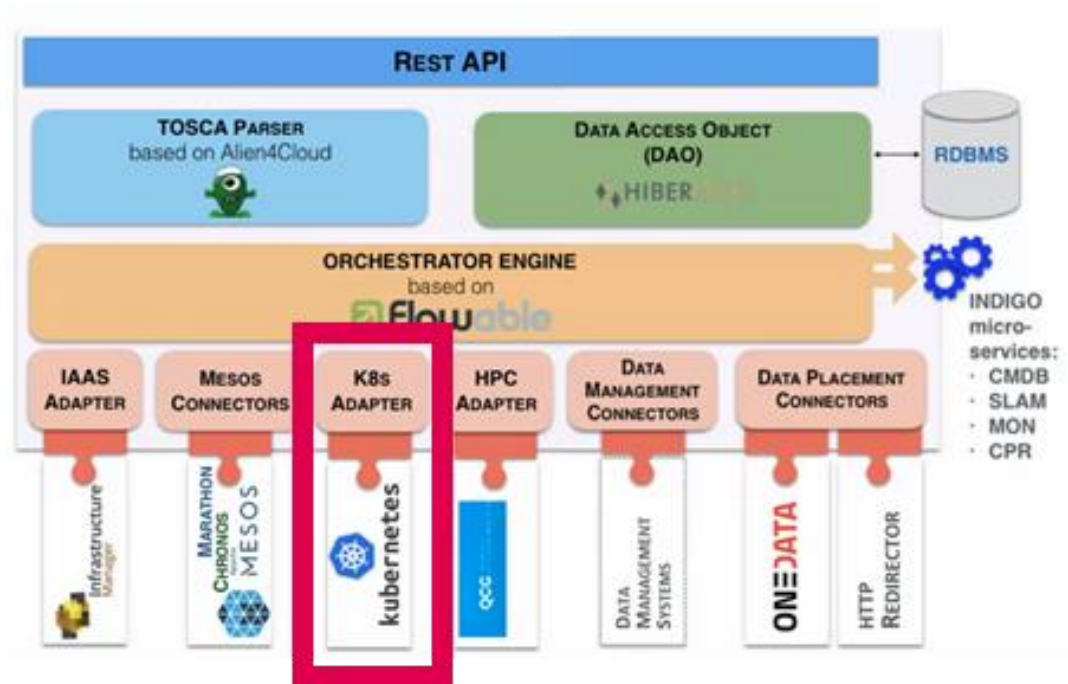


Figure 33: Kubernetes integration in IoTwinS cloud

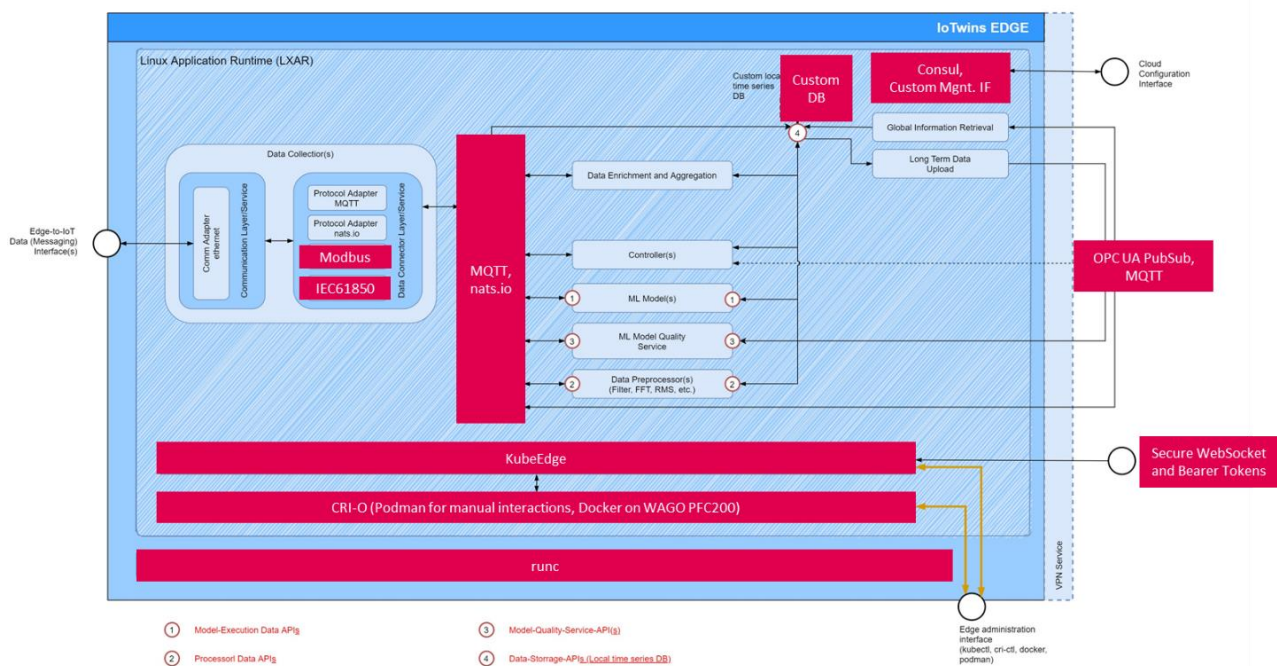


Figure 34: Standard components used on edge-level in all use-cases implemented in test-bed 7

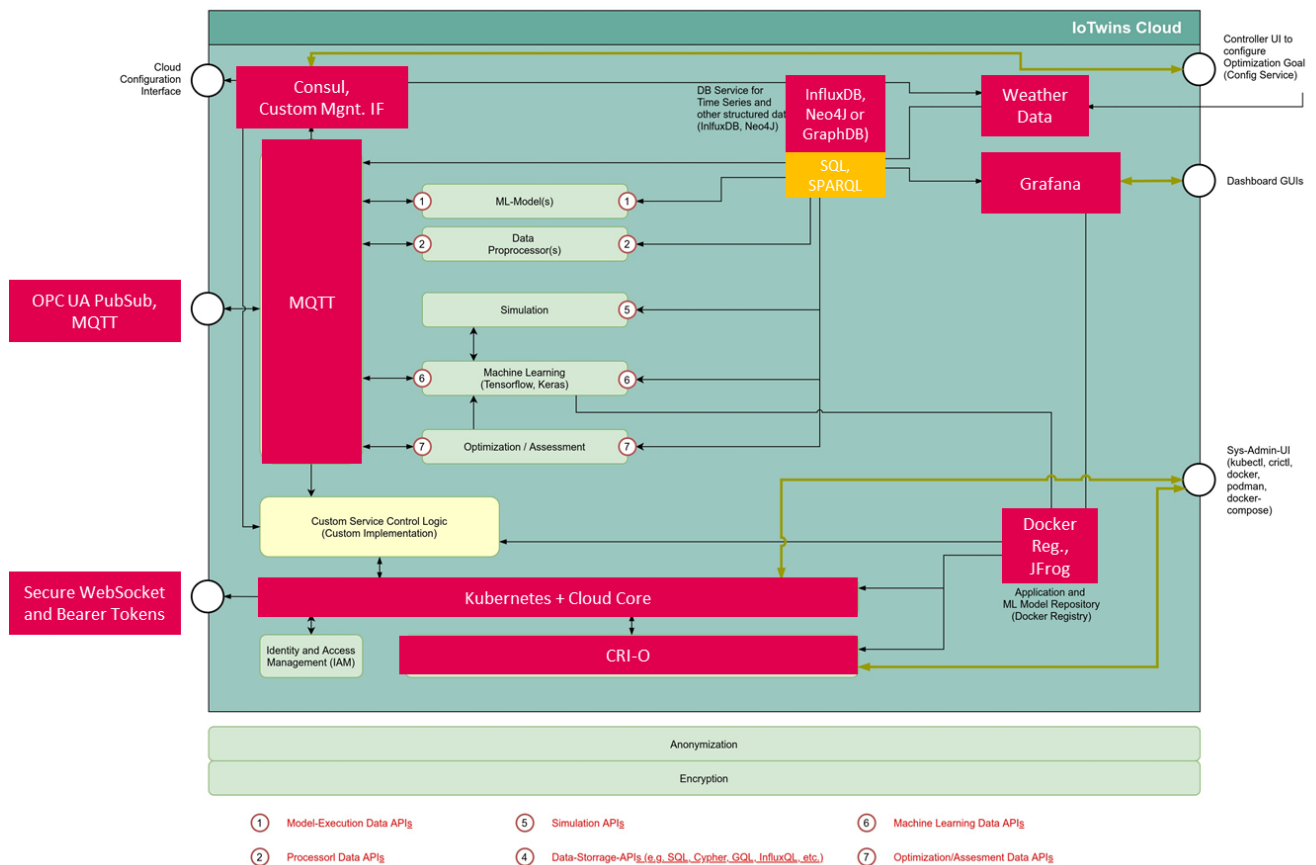


Figure 35: Standard components used on cloud-level in all use-cases implemented in test-bed 7

The figures shown above depict how the evaluated technologies are integrated in test-bed 7 to implement the different use-cases. Figure 36 shows to which layers of the IoTwinS architecture these technologies belong. On the Resource Middleware Layer CRI-O and Podman are used to run containerized applications; virtual machines and bare metal applications are not used but can be integrated (see also D2.2). The main part of the Resource Management Layer and Platform Service Layer are implemented using the different aspects of the Kubernetes and KubeEdge solutions, which proved to be flexible and extensive to cover a lot of the functionality that is needed on these layers. The components used on the Application Layer vary depend on the use-case and will be in most cases customer specific (see also D5.3).

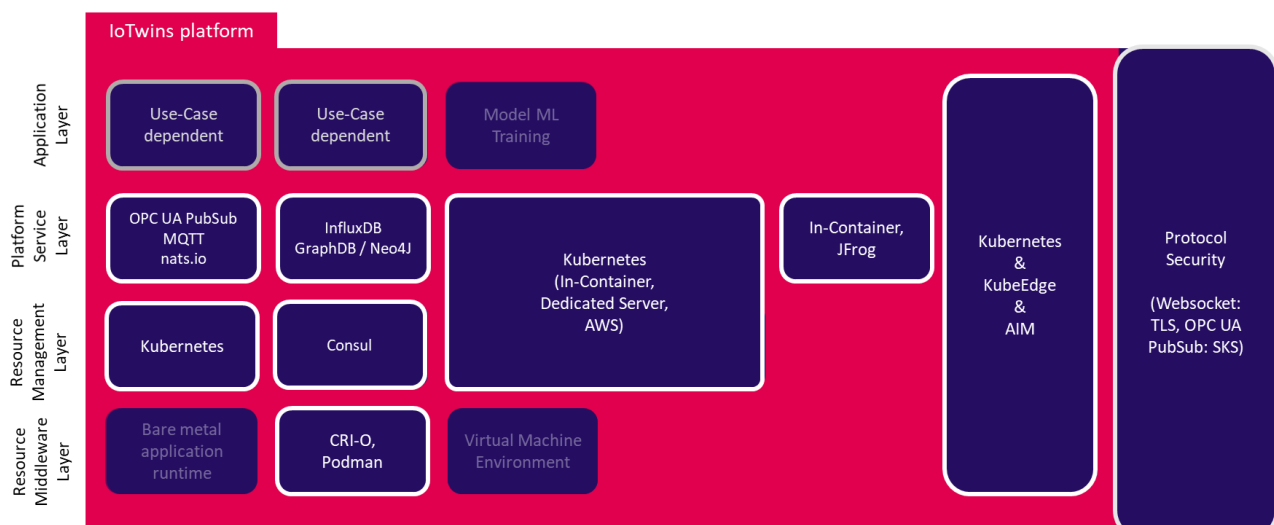


Figure 36: Technologies used to implement the IoTwinS platform for test-bed 7

6 Conclusion

This deliverable presents a summary on the daily activities around the work regarding the IoTwinS platform and the infrastructure provided to the project. The GitLab pipeline is described in section 2 and provides the base workflow for development. Especially AI services developed in WP3 and commonly used in different test-beds are hosted there.

The two showcases in section 3 underline the flexibility of the reference architecture and show how similar actions can be implemented with common off the shelf hard- and software.

Section 4 shows the different approaches of test-beds to implement the IoTwinS architecture. The provided INDIGO PaaS is well accepted and most test-beds have already been integrated. But even the ones, that decided for now not to use the platform are well aligned architectural wise, and therefore can be integrated easily later on with INDIGO-connectors.

Nevertheless, the different status of integration show that for the activities for the work package are not done yet, and some open points have to be handled by supportive actions by members of the WP2.