EUROPEAN COMMISSION

HORIZON EUROPE PROGRAMME – TOPIC: HORIZON-CL5-2022-D2-01

FASTEST

Fast-track hybrid testing platform for the development of battery systems

Deliverable D5.3: Integration plan for Digital Twin on the platform

Primary Author Antonio Paolo Passaro

Organization Comau S.p.A.

Date: 31.03.2025

Doc.Version: [Complete]

Co-funded by the European Union

UK Research and Innovation

Co-funded by the European Union and UKRI under grant agreements N° 101103755 and 10078013, respectively. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the

Document Control Information	
Settings	Value
Work package:	WP5
Deliverable:	Integration plan for Digital Twin on the platform
Deliverable Type:	Report
Dissemination Level:	Public
Due Date:	31.03.2025 (Month 22)
Actual Submission Date:	DD.MM.YYY
Pages:	< # >
Doc. Version:	Draft 1.0 / Draft 2.0 / Final
GA Number:	101103755
Project Coordinator:	Bruno Rodrigues | ABEE (bruno.rodrigues@abeegroup.com)

Formal Reviewers		
Name	Organization	Date
Laura Oca	MGEP	24.03.2025
Antonio Silvio de Letteriis	FLASHBATTERY	24.03.2025

Document History			
Version	Date	Description	Author
0.1	03.02.2025	Document structure	Antonio Paolo Passaro, Daniela Fontana (Comau)
0.2	10.03.2025	Complete Comau contents	Antonio Paolo Passaro, Daniela Fontana (Comau)

0.3	14.03.2025	Formatting	Antonio Paolo Passaro, Daniela Fontana (Comau)
0.4	18.03.2025	Complete Inegi contents	Marco Rodrigues, Nuno Marques (Inegi)
1.0	21.03.2025	Final check and formatting to prepare draft for reviewers	Daniela Fontana (Comau)
2.0	28.03.2025	Final version for submission	Antonio Paolo Passaro, Daniela Fontana, Eliana Giovannitti (Comau)

Project Abstract

Current methods to evaluate Li-ion batteries safety, performance, reliability and lifetime represent a remarkable resource consumption for the overall battery R&D process. The time or number of tests required, the expensive equipment and a generalised trial-error approach are determining factors, together with a lack of understanding of the complex multiscale and multi-physics phenomena in the battery system. Besides, testing facilities are operated locally, meaning that data management is handled directly in the facility, and that experimentation is done on one test bench.

The FASTEST project aims develop and validate a fast-track testing platform to deliver a strategy based on Design of Experiments (DoE) and robust testing results, combining multi-scale and multi-physics virtual and physical testing. This will enable an accelerated battery system R&D and more reliable, safer and long-lasting battery system designs. The project's prototype of a fast-track hybrid testing platform aims for a new holistic and interconnected approach. From a global test facility perspective, additional services like smart DoE algorithms, virtualised benches, and DT data are incorporated into the daily facility operation to reach a new level of efficiency.

During the project, FASTEST consortium aims to develop up to TRL6 the platform and its components: the optimal DoE strategies according to three different use cases (automotive, stationary, and off-road); two different cell chemistries, 3b and 4 solid-state (oxide polymer electrolyte); the development of a complete set of physic-based and data driven models able to substitute physical characterisation experiments; and the overarching Digital Twin architecture managing the information flows, and the TRL6 proven and integrated prototype of the hybrid testing platform.

LIST OF ABBREVIATIONS, ACRONYMS AND DEFINITIONS

Acronym	Name
AMQP	Advanced Message Queuing Protocol
Angular	A component-based framework for building scalable web applications
AKS	Azure Kubernetes Service
BE	Backend
DevOps	Development and Operations
DB	Database
DX.Y	Deliverable n. Y of Work Package n. X
DoE	Design of Experiments
DT	Digital Twin
ETL	Extract, Transform, Load
Eureka Server	A service registry for devices and services detection on a network
FE	Frontend
FMI	Functional Mock-up Interfaces
FMU	Functional Mock-up Unit
FTPS	File Transfer Protocol Secure
Go (GoLang)	Open source programming language
HiveMQ	a trusted MQTT platform
HTTP	Hyper text transfer protocol
HTTPS	Hypertext Transfer Protocol Secure
IO	Input Output
JSON	JavaScript Object Notation
JWT	JSON web token
Kubectl	Kubernetes command line tool
LIMS	Laboratory Inventory Management System
MongoDB	A document database
MQTT	Message Queuing Telemetry Transport
NoSQL	Not Only SQL
OS	Operating System
RabbitMQ	an open-source, message broker
RBAC	Role-Based Access Control
REST API (or RESTful API)	REpresentational State Transfer Application Programming Interface
Spring Boot	An open-source framework for application creation
SW	Software
TLS	Transport Layer Security
TX.Y	Task n. Y of Work Package n. X
URL	Uniform Resouce Locator
UUID	Unique identifier

UUT	Unit under test
VNET	Virtual NETwork
WAF	Web Application Firewall
WebSockets	A computer communications protocol
WP	Work Package

LIST OF FIGURES

Table of Contents

1. EXECUTIVE SUMMARY

The deliverable leverages specifications from D1.3 and insights from the DoE in WP2 to define the capability requirements and architecture for the DT implementation platform. It also utilizes the ontology from T5.1 and DT models from T5.2. By analysing the most suitable solutions for the mapped data, required data flow, and system capabilities, the key functional modules, communication protocols, software, hardware, front-end, and platform interfaces are identified. This includes incorporating simulation models for battery parameters and behavioural prediction tools from the inputs of WP3 and WP4. Additionally, various information output methods, such as user interfaces, are considered.

An architecture specification detailing the requisite capabilities, validation criteria for system performance, and frontend requirements for visualizing DT components is established.

The data pipeline, system architecture, and all necessary requirements for integrating the data and model outputs into the platform are outlined.

2. OBJECTIVES

The objective of WP5 is to create, instantiate and validate the digital twin of the battery systems and its components. The work started with the development of a digital twin ontology and a data mapping library: the results are reported in the deliverable D5.1. Afterwards, we proceeded with the definition of the DT component structure within Task T5.2 (deliverable D5.2) and then with the development of a plan to integrate the different components in the DT platform (Task T5.3). This deliverable reports the results of this last task and represents the starting point for the integration that will be implemented in the final task of WP5.

3. INTRODUCTION

Taking into account the data management and communication architecture (deliverable D1.2), the requirements and specifications for the Digital Twin (deliverable D1.3), the Digital Twin ontology, and the defined data assets, this document delivers a comprehensive description of the Digital Twin system architecture. It details the implementation of the core components of the Digital Twin, the application requirements, data pipeline, and integration processes necessary to implement the solution within the overall project platform (LIMS).

The document begins by defining the Digital Twin system architecture, which is a web-based platform hosted on the Cloud employing a microservice architecture. It analyses all the relevant Cloud resources and services, with a focus on the Azure VPC architecture and the databases.

In particular, the Cloud will host the In.Grid platform, Comau IoT platform for data collection, monitoring and representation, and a series of additional microservices that will compose the overall Digital twin platform. From this point in the document, In.Grid Digital Twin (In.Grid DT) refers to the ecosystem of microservices developed by Comau to implement the Digital Twin platform.

Next, the software architecture is described, which is based on Docker and Kubernetes. Digital Twin components are implemented as microservices running in Docker containers that are deployed to a Kubernetes cluster on the Cloud. These components communicate with each other and external systems (LIMS) using MQTT and REST API protocols. MQTT, a standard IoT protocol, is utilized primarily for data collection and real-time data exchange, while REST API, based on standard HTTP methods, is mainly used for the exchange of data between web services. Both communication protocols employ the JSON standard data format.

Communication between the Digital Twin platform and the LIMS platform is established via an MQTT bridge that connects the local Digital Twin broker to the external LIMS broker. This bridge enables data exchange between Digital Twin components and LIMS components by forwarding messages on predefined topics between the two remote brokers.

The implementation of all Digital Twin components is detailed as follows: starting with the data collector, which receives test results from LIMS, followed by the Digital Twin user interface for data visualization, the analysis service for data

aggregation, filtering, and representation, the services that manage the Digital Twin models, the Models Exchange Interface, the Test Request Handler, and the Models Management.

Given the continuous improvements within the project, there may be a need for modifications, adjustments, or the addition of new features throughout the project's lifespan.

4. DESCRIPTION OF WORK

4.1 Digital Twin system architecture

The Digital Twin platform is a web-based application hosted on Cloud that implements a microservice architecture. The microservice approach ensures flexibility and scalability of software modules that can be easily scaled up or down, according to users and customers' needs. All microservices are orchestrated by Kubernetes: in case of Azure cloud, an Azure Kubernetes Service (AKS) is the managed service of the orchestrator.

In Figure 1, the Digital Twin Architecture is presented, emphasizing its core components. All services run inside a subnet:

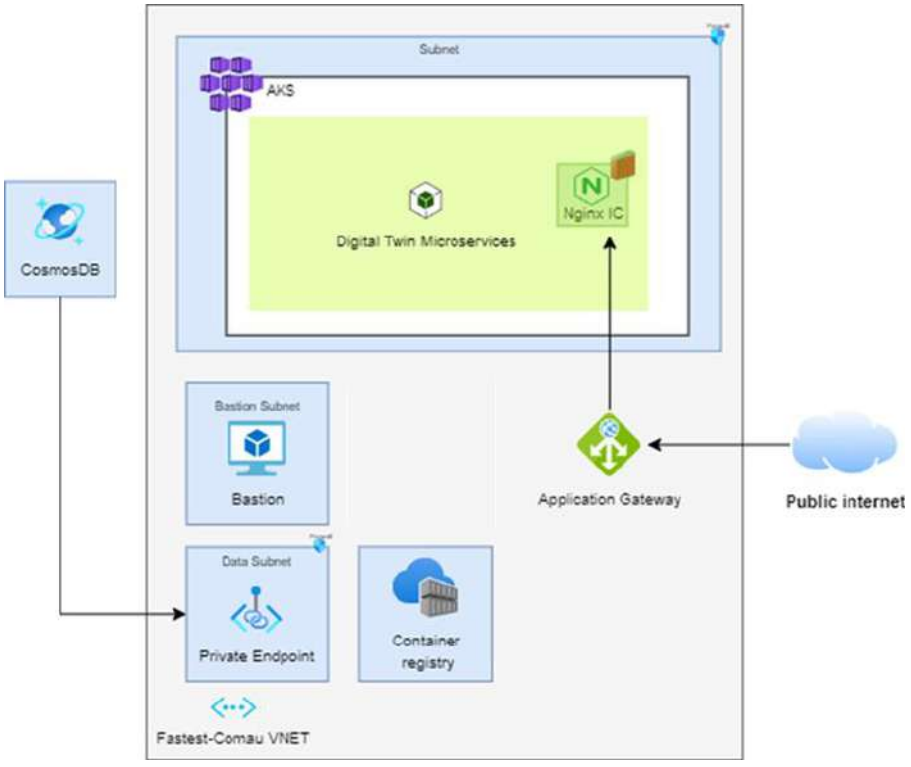

Figure 1 Digital Twin Architecture

In the design phase data isolation was considered as a requirement. For this reason, the subset of services is deployed for the project and, if shared, contains network segregation principles.

The network segregation is a critical concept of the solution designed, to ensure security of data flowing among different services. For this scope a segregated Virtual NETwork (VNET) and a subnet for Kubernetes services have been defined. In the defined VNET, every communication with external should be managed with Virtual Endpoints.

Starting from bottom level of the data stack, the non-relational database Cosmos DB has been connected to the cluster using a dedicated private endpoint. The business logic of the application is running in the green area, and the only point of access is the Nginx reverse proxy, directly connected to the Application Gateway. The application gateway works as Web Application Firewall (WAF) ensuring also security of requests coming from users on Internet.

A subset of services is deployed for the project and is shared across services:

- Container registry;
- Bastion host.

The container registry is useful to store all docker images used for the project and is connected to the DevOps pipelines of the development team.

Finally, the Bastion host is a virtual machine connected to the VNET of the deployment, used for monitoring, extracting logs and interacting with Kubectl command line interface. From Bastion host administrator users can deeply interact with all services in the deployment.

4.2 SW architecture

Digital Twin SW architecture consists of different components, each one representing a specific functionality of the system (Figure 2). Each component is a microservice running in a Docker container deployed to the Kubernetes cluster on DT Cloud. The user interface is web based, and it runs behind a reverse proxy that allows only the HTTPS communication. The components communicate between each other and with external systems by using MQTT/AMQP and REST API protocols. For this reason, a RabbitMQ broker is deployed within the cluster as for the other components.

Rest API are based on standard HTTP methods and are the preferred way to exchange data between web services.

The MQTT protocol is mainly used for data collection and for real-time data exchange with the subscribe/publish mechanism. The MQTT broker manages the communication between the different services thus allowing subscribing and publishing data on different topics.

JSON is the standard data format that will be used for data exchange both between the Digital twin and other systems and between the DT components. The JSON format will be used for exchange MQTT messages and for Rest API contents.

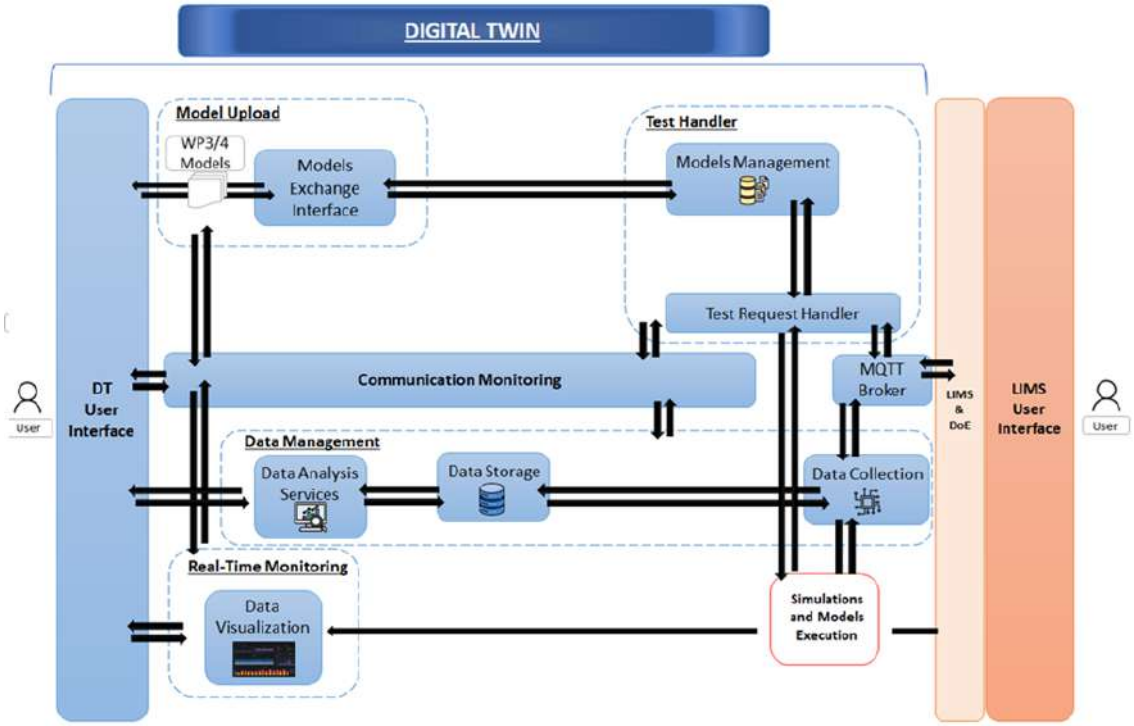

Figure 2 Digital Twin software architecture

4.3 Integration plan with LIMS (WP6)

The communication between DT platform and LIMS is based on the MQTT protocol and takes place through a MQTTS bridge between the HiveMQ broker (deployed on LIMS platform) and the RabbitMQ broker (deployed on DT platform). The bridge configuration is done on HiveMQ broker, using the HiveMQ bridge extension. The bridge allows the communication and the data exchange between the LIMS components and the DT components and will forward all the MQTT messages of predefined topics between the two remote brokers (Figure 3).

Communication is secured thanks to the TLS configuration. The HiveMQ broker will connect to the RabbitMQ broker using the TLS port 8883 and with user and password authentication. For the TLS configuration it is necessary to map the client and server certificates with "p12" format inside the HiveMQ docker container and add the path in the configuration file.

In the bridge configuration file it is possible to specify:

- ❑ RabbitMQ address and port
- ❑ Username and password for the authentication
- ❑ TLS configuration:
  - Path and password of the client certificate (keystore)
  - Private key password
  - Path and password of the server certificate (trustore)
- ❑ Topics configuration:
  - Topics to bridge messages
  - Mode: Publisher (PUB) or Subscriber (SUB)

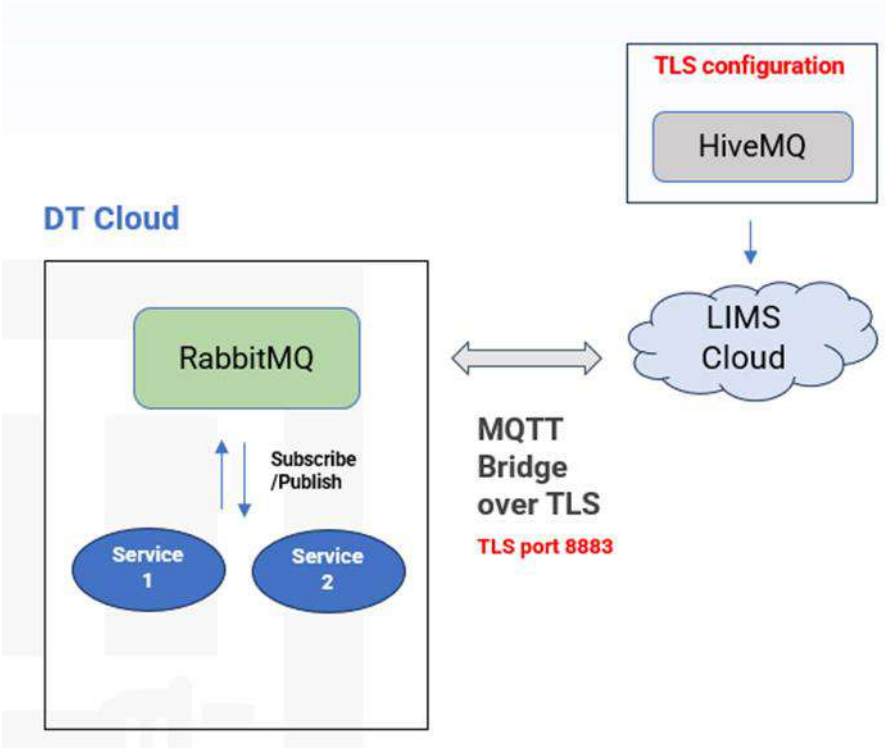

Figure 3 MQTTS bridge

4.4 Digital Twin components

4.4.1 Data collector

The Data Collector microservice plays a crucial role in the data acquisition and storage process within the system. Its primary responsibility is to capture and store test results from the Laboratory Information Management System (LIMS), handling both real and simulated battery tests.

At its core, the Data Collector is developed as a Java Spring Boot microservice designed for robustness and scalability. It integrates closely with RabbitMQ, serving as a message listener to facilitate real-time data processing and storage.

The Data Collector continually listens for incoming data through a RabbitMQ listener. This listener is configured to connect to the RabbitMQ broker, subscribing specifically to the AMQ queue named "fastest" via the AMQ protocol on port 5672. When test results are published to this queue, the listener captures the messages, ensuring that they are promptly processed.

Upon receiving test results, the Data Collector converts these results into the Digital Twin (DT) data format. This standardized format ensures consistency and compatibility with other system components. Once converted, the data is stored in a local NoSQL database, MongoDB, known for its flexibility and scalability in handling large datasets.

To facilitate seamless data acquisition, an MQTT to AMQP bridge has been configured in RabbitMQ broker. This bridge enables the effective transfer of

messages from an MQTT topic to the RabbitMQ AMQP queue. The configuration of this bridge is executed at runtime by the Data Collector's configuration module, ensuring dynamic adaptability to system requirements.

The configuration specifics include:

- MQTT Topic: `/test-results` (configured in the MQTT bridge)
- AMQP Queue: `fastest`
- Message Payload: As detailed in Deliverable D5.2 under "Communication Data Model"

The data acquisition flow begins with the remote HiveMQ broker, which publishes test results to the specified MQTT topic and forwards them to the RabbitMQ broker. The MQTT to AMQP bridge then transfers these messages to the corresponding AMQP queue in the RabbitMQ broker. The Data Collector's RabbitMQ listener, subscribed to this queue, receives the messages and initiates the processing and storage sequence (Figure 4).

Advantages of the Data Collector System:

1. Real-Time Data Processing: The integration with RabbitMQ allows for real-time reception and processing of test data, ensuring that the system remains updated and responsive to new information.
2. Scalability: Leveraging Spring Boot and MongoDB enhances the microservice's scalability, accommodating increasing volumes of data without compromising performance.
3. Flexibility: The use of a standardized DT data format and the dynamic configuration of the MQTT bridge provide flexibility in handling diverse data sources and meeting varying system demands.
4. Reliability: This architecture ensures reliable data transfer from the LIMS to the local database, maintaining data integrity and availability.

Considering the continuous advancements in the project, future enhancements may include:

- Enhanced Data Validation: Implementing more robust data validation mechanisms to ensure the quality and consistency of the incoming test data.
- Monitoring and Alerts: Developing monitoring tools and alert systems to promptly identify and address any issues in the data acquisition process.
- Optimized Data Storage: Exploring additional optimizations for the data storage process to improve performance and efficiency.

In conclusion, the Data Collector microservice is a vital component of the project, ensuring seamless data collection, conversion, and storage, while maintaining high standards of performance, scalability, and reliability.

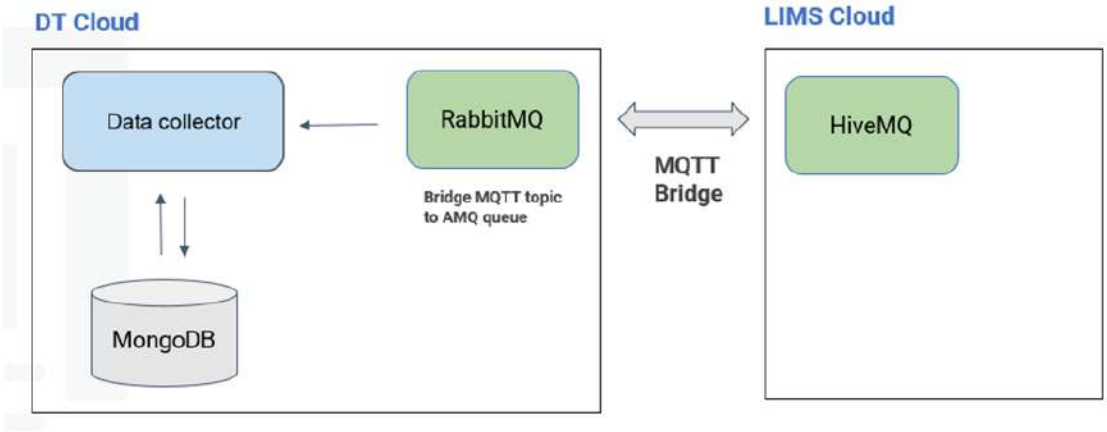

Figure 4 Data Collector

4.4.2 Analysis service

The Analysis Service is a backend microservice designed to manage and process test data, providing essential functionalities for data monitoring, aggregation, filtering, and representation. It serves as the intermediary between the data stored in the database and the Digital Twin (DT) user interface, ensuring that all relevant information is available for visualization in a coherent and efficient manner.

The Analysis Service is developed using Spring Boot java framework. This microservice reads data from a MongoDB database, which stores all test results and related information. By accessing this data, the Analysis Service enables comprehensive monitoring and analysis of the test results over time.

Key features:

1. Data Aggregation: The Analysis Service aggregates test data to provide summarized views and insights. This helps in understanding trends and patterns in the test results, facilitating more informed decision-making.
2. Data Filtering: To manage and deal with large volumes of data, the Analysis Service includes filtering capabilities. Users can filter data based on various criteria such as test dates, test types, and outcomes, enabling more focused and relevant data analysis.
3. Data Representation: The service converts raw test data into a format suitable for visualization on the DT user interface. This includes structuring data in a user-friendly manner, ensuring that users can easily interpret and interact with the information.

To facilitate interaction with the frontend (FE) user interface, the Analysis Service exposes a set of RESTful APIs. These APIs are designed to provide efficient access to all data related to the tests performed, ensuring that the frontend has all necessary information for comprehensive visualization.

By exposing well-defined REST APIs, it ensures that the frontend can efficiently fetch and display all necessary data, providing users with a coherent and interactive experience (Figure 5).

Advantages of the Analysis Service:

1. Enhanced Data Insights: By aggregating and summarizing test data, the service offers valuable insights, helping users to understand trends and make informed decisions.
2. Scalability: Built on Spring Boot and MongoDB, the service is designed to handle increasing volumes of data and growing user demands without compromising performance.
3. User-Friendly Interface: The REST APIs ensure that the frontend can easily interact with the backend, providing a seamless and user-friendly experience for data visualization.

As the project progresses, there are several potential enhancements to the Analysis Service that could further improve its functionality:

- Advanced Analytics: Incorporating more advanced analytical capabilities, such as predictive analytics and machine learning, to provide deeper insights into test data.
- API Expansion: Extending the range of APIs to include more granular data retrieval options and additional filtering capabilities.

In conclusion, the Analysis Service ensures effective data processing, aggregation, and representation. By providing robust and scalable backend support, it enables comprehensive visualization and analysis of test data, enhancing the overall functionality and user experience of the DT platform.

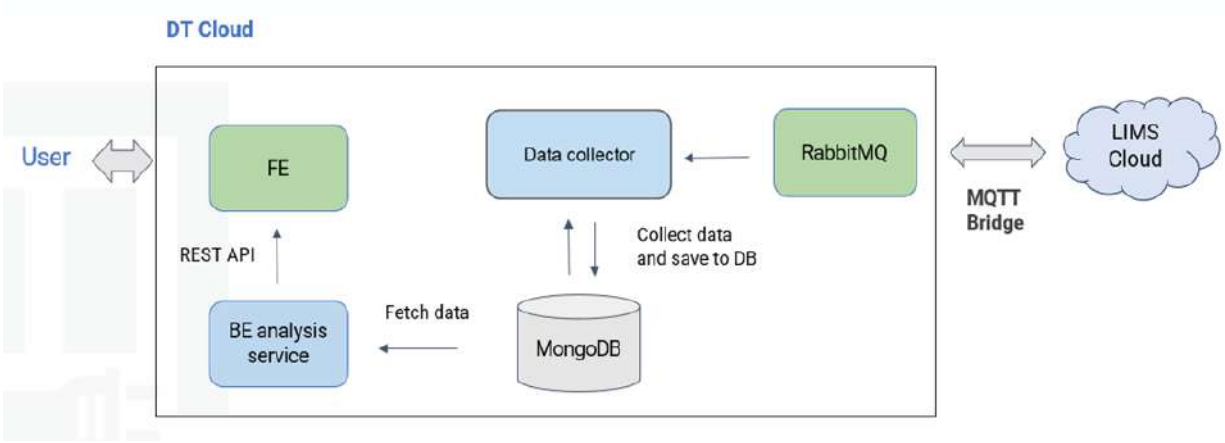

Figure 5 Analysis Service

Exposed APIs:

- Get list of tests, with the possibility to filter by completed, in progress, started tests. For each test, test name, test type, unit under test, scheduled date, status are specified.

- Get test details, with the possibility to filter by test id or test unique identifier. Each test is described by:
  - Test name (e.g. Overcharge)
  - Test type (e.g. Module or Cell level)

- UUT (Unit under test e.g. Cell-01)
- UUID (Unique identifier)
- Test bench (Physical/virtual)
- Scheduled date (UTC)
- Test status (In progress, completed, started)
- List of variables. For each variable, variable name, type (Input/output) and unit of measure are specified

- Get tests analytics: To retrieve the number of tests performed at cell or module level and the number of in progress, completed and started tests.
- Get test results: Filter tests by test id or test unique identifier and retrieve the input/output variables values over time to build the trend chart on the user interface.

4.4.3 Communication monitoring

The communication monitoring component ensures that all DT components are up and running by actively monitoring the health and the availability of the components over time. The communication monitoring component is implemented by the Eureka Server.

Eureka Server is a service registry provided by Netflix, crucial for the microservices architecture. It is an essential component within Spring Cloud Netflix and is widely used to maintain the Service Registry. The primary role of Eureka Server is to allow microservices to register themselves and to discover other registered services seamlessly, promoting service discovery within the ecosystem of microservices (Figure 6).

Eureka Server functions as a service registry server, where all independent microservices register themselves. The core idea is to enable these microservices to look up each other dynamically, facilitating communication and coordination without hard-coding any service addresses. This dynamic nature to locate services is essential in cloud environments where service instances frequently change due to scaling and auto-scaling activities.

Advantages of using Eureka Server:

1. Service Discovery: Eureka Server provides an intuitive way for services to discover each other in a microservices architecture. This ensures that the communication between services is always robust, even when instances are added or removed dynamically.
2. Load Balancing: Through Eureka Server, load balancing across multiple instances of a service is automated, enhancing the system's efficiency and resilience.
3. Failover and Resilience: Eureka’s client-side load-balancing capabilities enhance failover and resilience by allowing services to route requests to available instances, even if some instances fail.

4. Dynamic Scaling: Eureka facilitates dynamic scaling of services. When new instances are spun up, they automatically register with Eureka Server, making them instantly discoverable by other services.
5. Ease of Configuration: Integrating with Spring Cloud Netflix Eureka involves minimal configuration changes, allowing for quick setup and deployment, especially beneficial for iterative and agile development environments.

Disadvantages of using Eureka Server:

1. Overhead: Maintaining a service registry adds overhead to the system architecture. There is a need for an additional layer that needs to be managed and monitored.
2. Single Point of Failure: Unless configured in a high-availability setup (with multiple Eureka instances), the Eureka Server can become a single point of failure, risking the entire service discovery mechanism.
3. Latency: There might be a slight latency introduced due to service discovery through Eureka, which may affect the system's performance, especially when scaling at a large number.

In this project, microservices have been developed to connect to the Spring Cloud Netflix Eureka Server. Each microservice registers itself to the Eureka Server upon startup, making itself available for discovery by other services. This setup ensures that any microservice can discover and communicate with others by querying the Eureka Server, without needing to hardwire the addresses of other services.

The configuration involves setting up a Eureka Server and ensuring that each microservice client within the project is configured to register with the server. This involves straightforward integration using annotations such as `@EnableEurekaServer` on the server application and `@EnableEurekaClient` on the client microservices.

This configuration ensures that the communication within the microservices architecture remains flexible, scalable, and resilient, promoting a robust microservices ecosystem through dynamic service discovery and registration [1], [2], [3].

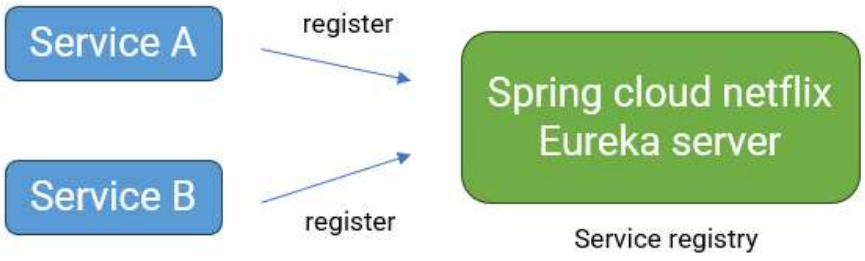

Figure 6 Eureka Server

4.4.4 Authentication and Authorization

In the context of this project, authentication and authorization are crucial elements to ensure secure access to the platform and to the backend services. To achieve

this, we have integrated Spring Cloud Gateway and Keycloak, leveraging their robust features for handling authentication and authorization processes.

Spring Cloud Gateway serves as the API gateway for the system, acting as a reverse proxy to route all incoming API requests to the appropriate microservices. It provides a centralized entry point to the backend services, managing and securing the API traffic.

- Redirection and Routing: Spring Cloud Gateway is configured to redirect all API calls to the respective microservices. This centralization simplifies the routing logic and provides a single point for managing requests.
- Pre-Authorization Check: Before redirecting requests, the API gateway performs a crucial role in checking authorization privileges. It achieves this by validating the received tokens through Keycloak, ensuring that only authorized requests are processed.

Keycloak is utilized as the identity and access management service within the project. It helps in managing all aspects of user authentication, authorization, and identity management.

- User and Group Management: Keycloak stores all user credentials, user groups, and their roles, providing a centralized store for managing user identities.
- External Identity Providers: Keycloak supports integration with external identity providers, allowing users to authenticate using various third-party services such as Microsoft Entra ID or other OAuth-compliant identity providers.
- Token Management: When a user attempts to access a secured endpoint, Keycloak issues a JWT token upon successful authentication. This token contains all the necessary user information and access privileges encoded within it.
- Role-Based Access Control (RBAC): In.Grid DT implements RBAC to manage access control, ensuring that users have appropriate permissions to access various resources based on their assigned roles.

Implementation:

1. Authentication Flow:

- When a user attempts to access a resource, they are redirected to the login page.
- Upon successful authentication, In.Grid DT issues an access token (JWT), which is then used for subsequent requests to access the secured endpoints.

2. Authorization Flow:

- For every incoming API request, Spring Cloud Gateway intercepts the request and extracts the token.
- The gateway consults Keycloak to validate the token and verify the user's permissions.

- If the token is valid and the user has the required permissions, the request is routed to the respective microservice.
- If the token is invalid or the user lacks the necessary permissions, the gateway denies access, ensuring that unauthorized requests are not processed.

Advantages of combining Spring Cloud Gateway and Keycloak:

- Centralized Security Management: By leveraging Keycloak, we have a unified solution for managing user identities, roles, and permissions across the entire system.
- Scalability and Flexibility: The use of Spring Cloud Gateway allows for scalable routing and handling of API requests, facilitating dynamic routing and load balancing.
- Enhanced Security: The strict validation of tokens and enforcement of access controls through RBAC ensure that the system remains secure and only authorized users can access sensitive data.
- Ease of Integration: Both Spring Cloud Gateway and Keycloak are built to work seamlessly with microservices architectures, simplifying integration and configuration processes.

In conclusion, the combination of Spring Cloud Gateway and Keycloak provides a robust and scalable solution for managing authentication and authorization, ensuring secure access to the platform and to the backend services [4], [5], [6].

4.4.5 User interface

The user interface allows users to access the Digital Twin data and implements authentication and role-based authorization with possibility of data export.

The Digital Twin user interface is based on Angular. The user interface has a side menu where it is possible to select different sections of the application, each with a specific functionality. All sections call the REST APIs made available by the backend microservices that implement the business logic and process the information.

From the dashboard (Figure 7) it is possible to navigate to the other sections of the application as for the left side menu.

There are two main sections:

- Test summary
- Trends

The Test summary section (Figure 8) shows a list with the details of all the tests that have been completed, running or started but waiting to be executed. It shows for each test:

- Id: Progressive number of the test
- Type of test: Ex. Overcharge

- Date: Scheduled date of test
- Progress: Current status of the test
- Action: It opens the detail of the single test

On the right side, the cumulative number of tests is shown based on the number of tests performed per cell or module and based on their status.

By clicking on the button “Action” of a particular test, a popup will appear that will show all the test details (Figure 9): Name, Type, Unit under test, UUID (Unique identifier), test bench, current status, and the list of input and output variables.

The Trend section instead allows to show the results of completed tests or tests in progress. In particular, by clicking on the "filter" button it is possible to select the test to view the results of by searching for the test by name, type, unit under test or by unique identifier (Figure 10) Finally, after selecting the test, the temporal trend of the input and output variables is shown (Figure 11).

Figure 7 Digital Twin dashboard

Figure 8 Test summary section

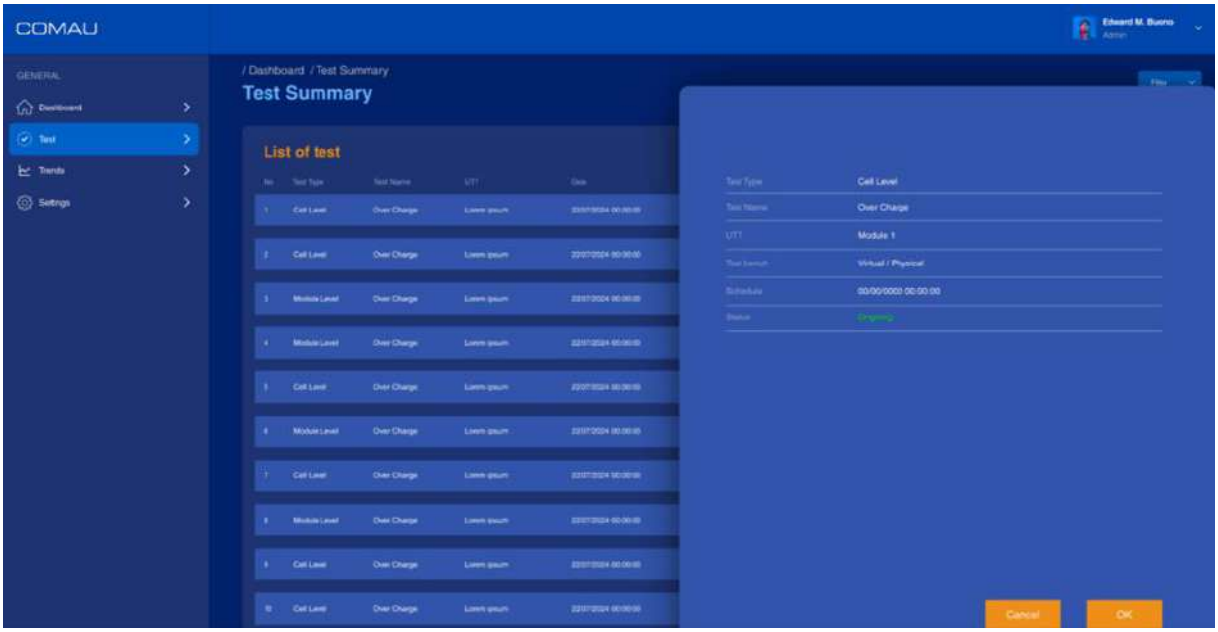

Figure 9 Test details section

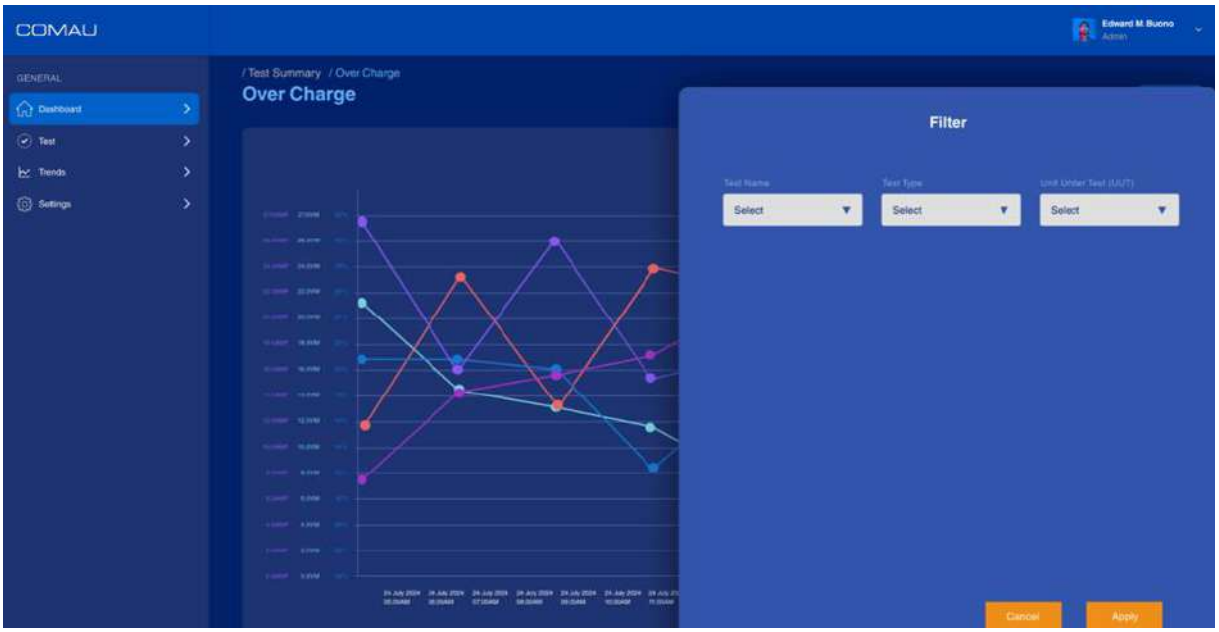

Figure 10 Test filter section

Figure 11 Trend chart section

4.4.6 Models Exchange Interface

The Model's Exchange Interface is designed to facilitate the efficient and secure sharing of simulation files among consortium partners, adhering to the Functional Mock-up Unit (FMU) standards. This interface ensures structured access control and file organization. It is implemented as a containerized solution, deployed on Comau's Kubernetes-based platform, guaranteeing scalability, security, and ease of access for all project stakeholders (Figure 12).

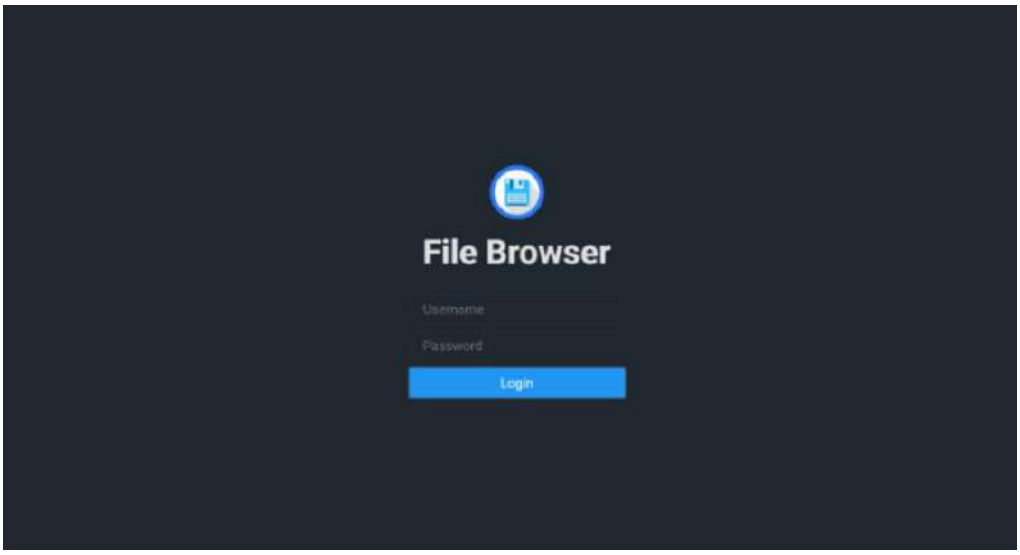

Figure 12 Models Exchange Interface Login

The Interface is developed in Go (Golang) and utilizes the standard net/http package alongside Gorilla Mux for routing. The frontend is built with Vue.js, providing a modern and responsive user interface. Communication is handled through HTTP/HTTPS, with support for TLS encryption to ensure secure connections. WebSockets facilitate real-time updates for file operations. Authentication mechanisms include JWT-based authentication. File system interactions rely on Go's OS and IO packages, supporting local storage. Role-based access control (RBAC) allows user permission management, with JSON-based configurations to define access restrictions. A logger to monitor file activities will not be implemented in the current setup, as access is limited to a small number of project partners. However, logging capabilities will be considered and integrated as the project scales and access requirements grow.

To ensure data integrity and secure collaboration, WP3 and WP4 members are granted full permissions, including read, write, and delete access, enabling active contributions and updates to simulation files. Non-WP3 and WP4 members are limited to read-only access, ensuring they can download and utilize simulation files without altering the data (Figure 13).

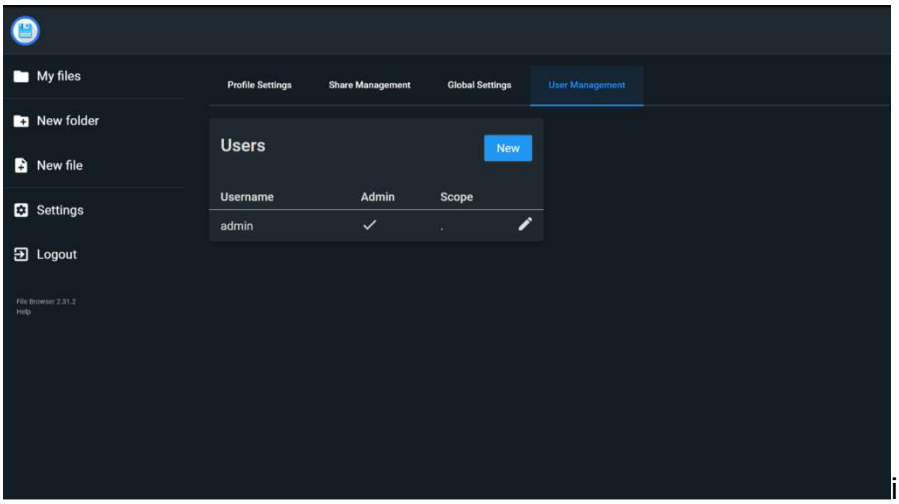

Figure 13 User's Management Interface

The interface follows a systematic folder organization designed for easy navigation and efficient file management. It comprises top-level folders dedicated to different test categories, specifically WP3 - Performance and ageing Tests and WP4 - Safety and Reliability Tests . Within each top-level folder, the structure further categorizes tests based on the component level, including Cell, Module, and Pack. Subsequently, it specifies the chemistry involved in the tests (Gen3b or solid-state battery). After selecting the chemistry, the structure differentiates between Physics-based Models and Data-driven Models.
Only after navigating through these layers are the FMU (Functional Mock-up Unit) simulation files available. All FMU files within each test folder adhere strictly to a structured naming convention (e.g., TestName_testVersion), providing clarity, facilitating version control, and ensuring quick identification and retrieval (Figure 14). Given that the simulation files are relatively small (around 5 MB), no file size limitations or performance constraints are expected when sharing the FMU files

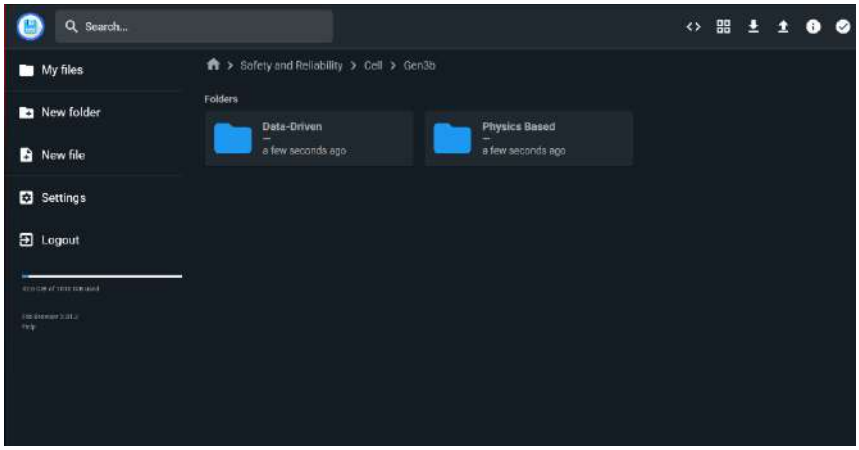

Figure 14 Structured Folder Hierarchy

4.4.7 Test request handler

The Test Request Handler serves as a link between the Laboratory Information Management System (LIMS), the Design of Experiments (DoE), and the Models Management Module. Its main role is to enable the exchange of data while managing the entire lifecycle of test requests.
The handler manages the reception and transmission of data. It processes incoming requests from LIMS and forwards simulation and test procedure specifications to the DoE for calculating optimal procedures. Once the DoE processes the request, the handler receives the updated testing procedures and communicates them to the co-simulation platform, initiating the test execution.
Implemented as a Python script, the handler complies with safety norms and standards essential for data security and integrity. It is deployed within the same Kubernetes containerized environment as the Models Exchange Interface, promoting a cohesive and scalable solution.
Communication is facilitated using MQTT. The handler subscribes to a specific MQTT topic (to be defined) to receive detailed information about the test the user intends to execute. The structure of this message is outlined in D5.2. These specifications

are then shared with the Models Management Module, which returns the corresponding testing procedures. Additionally, the handler subscribes to another MQTT topic (to be defined) to receive optimized procedures calculated by DoE. Upon receiving the updated procedures, it transmits the data to the co-simulation platform. The Models Management Module then determines whether the simulation file requires transmission to the co-simulation platform. If it is needed, this will be done using Safe File Transfer Protocol (SFTP).
All exchanged and processed data is systematically stored in the designated database. The database's structure and schema are described in D5.2, ensuring efficient data retrieval and maintaining data integrity.

4.4.8 Models Management

The Models Management component is designed to manage the simulation models and testing procedures essential for accurate and efficient simulation workflows. Implemented as a Python script, its primary role is to process test requests received from the Laboratory Information Management System (LIMS) and determine the corresponding simulations, models, and test procedures to be executed based on the specific use case and Unit Under Test (UUT).
Upon receiving a test request, the script identifies and retrieves the appropriate testing procedures from its managed repository. It also performs a verification process to check if the latest version of a test has already been forwarded to the co-simulation platform. If the test version has not been sent, the Models Management component ensures that it is promptly forwarded to the Test Request Handler for further processing.

5. CONCLUSION

In summary, this document has provided a thorough overview of the Digital Twin system architecture, detailing the implementation of its core components, the application requirements, data pipeline, and integration processes necessary for its deployment within the overall project platform (LIMS). By leveraging the data management and communication architecture (deliverable D1.2), the Digital Twin specifications (deliverable D1.3), the ontology (deliverable D5.1), and defined data assets, we have established a solid foundation for the Digital Twin platform.

We have outlined the architecture as a Web-based platform hosted on the Cloud, employing a microservice approach to ensure scalability and flexibility. The utilization of Docker and Kubernetes for deploying microservices, along with MQTT and REST API protocols for communication, demonstrates our commitment to adopting industry standards and best practices.

The communication bridge between the Digital Twin platform and LIMS platform facilitates seamless data exchange, ensuring that all components operate harmoniously within the overall system. Furthermore, we have described the implementation of various Digital Twin components, including the data collector, user interface, analysis service, model management services, the models exchange interface, the test request handler and the model's management component highlighting their critical roles in achieving the project's goals.

As the project progresses, it is essential to remain adaptable to changes and advancements. The need for potential modifications, adjustments, or new features will be addressed proactively to ensure the system remains robust and capable of meeting evolving requirements.

In conclusion, the Digital Twin system architecture presented in this document lays the groundwork for a successful integration within the LIMS platform. By following the outlined implementation plan and remaining responsive to ongoing developments, a powerful and effective Digital Twin solution that will enhance the project's outcomes and objectives will be delivered.

6. REFERENCES

[1] «https://cloud.spring.io/spring-cloud-netflix/reference/html/,» [Online].

[2] «https://www.baeldung.com/spring-cloud-netflix-eureka,» [Online].

[3] «https://cloud.spring.io/spring-cloud-netflix/multi/multi_spring-cloud-eureka-server.html,» [Online].

[4] «https://spring.io/projects/spring-cloud-gateway,» [Online].

[5] «https://www.baeldung.com/spring-cloud-gateway,» [Online].

[6] «https://www.keycloak.org/,» [Online].