Ref. Ares(2026)4055916 - 20/04/2026

EUROPEAN COMMISSION

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

FASTEST

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

Deliverable D5.4: Digital Twin Integration

Primary Author Maria Grazia Toma

Organization Comau S.p.A.

Date: [17.04.2026]

Doc.Version: [Complete]

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 European Climate, Infrastructure and Environment Executive Agency (CINEA). Neither the European Union nor CINEA can be held responsible for them

Document Control Information	
Settings	**Value**
Work package:	WP5
Deliverable:	Digital Twin Integration
Deliverable Type:	Report
Dissemination Level:	Public
Due Date:	31.01.2026 (Month 32)
Actual Submission Date:	DD.MM.YYYY
Pages:	28
Doc. Version:	V0.1 / Final
GA Number:	101103755
Project Coordinator:	Bruno Rodrigues | ABEE (bruno.rodrigues@abeegroup.com)

Formal Reviewers		
Name	**Organization**	**Date**
Name	Short name organization	DD.MM.YYYY
Iñaki Leciñana Arregui	Ikerlan	23.02.2026
Shuchen Liu	FEV	10.03.2026

Document History			
Version	**Date**	**Description**	**Author**
0.1	14.01.2026	First Draft before review	Maria Grazia Toma (Comau)
1.0	17.04.2026	Final version for submission	Maria Grazia Toma (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 develope and validate a fast-track testing platform able 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 TRL 6 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 datadriven 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	A component-based framework for building scalable web applications
Angular	A component-based framework for building scalable web applications
CORS	Cross-Origin Resource Sharing
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
Iframe	Inline frame
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
Nginx	Web server that can also be used as a reverse proxy
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
SFTP	Secure Model File Transfer
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 TABLES

Table 1 Keycloak Client configuration 15
Table 2 Configuration of Spring Cloud Gateway 15

LIST OF FIGURES

Figure 1 Communication and validation diagram 14
Figure 2 Configuration of Nginx server 17
Figure 3 Definition and the implementation of MeiComponent (Models Exchange Interface Components) in Angular. We can see also the template definition and the sanitized Iframe url 18
Figure 4 Terminal output showing the successful execution of an SFTP command from inside the Digital Twin container. The log confirms the connection to the LIMS storage account and the upload of a test file via the Azure Private Endpoint. 19
Figure 5 Test result LIMS forwarding test results to DT via MQTT 20
Figure 6 Fastest UI 21
Figure 7 Registration process 22
Figure 8 Digital twin dashboard 23
Figure 9 Test summary page 24
Figure 10 Test details page 25

Table of Contents

1. EXECUTIVE SUMMARY 7
2. OBJECTIVES 8
3. INTRODUCTION 8
4. DEPLOYMENT OF DIGITAL TWIN PLATFORM 9
4.1 Cloud Infrastructure and Orchestration 9

4.2 Microservices Deployment Configuration 9
4.2.1 Private Container Registry Setup 9
4.2.2 Kubernetes Orchestration: Deployments and Services 10
4.2.3 Deployed Components 11
4.3 Authentication and Authorization (API Gateway & Keycloak) 12
4.4 Frontend Integration (Iframe and Nginx Configuration) 16
4.4.1 Nginx Configuration Details 16
4.4.2 Iframe Implementation 17
5. SYSTEM INTEGRATION CONFIGURATION 18
5.1 MQTT Bridge Setup 18
5.2 Secure Model Transfer (Azure Private Endpoint) 18
6. VERIFICATION OF INTER-PLATFORM INTERACTIONS (DT & LIMS) 19
6.1 Interaction Log 19
7. USER INTERFACE FUNCTIONALITIES 20
7.1 GUI Sections and Walkthrough 20
7.1.1 Registration and Secure Login (MFA) 21
7.1.2 Dashboard and Navigation 22
7.1.3 Test Summary Section 23
7.1.4 Test Details and Results Visualization 24
7.1.5 Models Exchange Interface 25
8. CONCLUSION 27
9. REFERENCES 28

1. EXECUTIVE SUMMARY

This deliverable reports the activities performed within Task 5.4, focusing on the technical deployment, configuration, and integration of the Digital Twin (DT) platform. Following the architecture defined in D5.3, the DT solution has been successfully instantiated on the Azure Cloud infrastructure using a microservice-based approach orchestrated by Kubernetes.

The document details the specific configurations applied to the core components, including the implementation of the Data Collector, Analysis Service, and Model Management systems. Furthermore, the document provides a transparent overview of the integration with the LIMS platform (WP6), detailing the active MQTT bridge for real-time messaging and the Azure Private Endpoint configuration for secure model file transfer (SFTP). Finally, a walkthrough of the deployed User Interface is presented.

2. OBJECTIVES

The main objective of this deliverable is to demonstrate the operational status of the Digital Twin platform and its effective integration with the project ecosystem. Specifically, this document aims to:

- Describe the deployment environment and the configuration of the microservices on the Azure Virtual Machine.
- Detail the technical implementation of the communication interfaces (MQTT and REST APIs).
- Document the specific configurations required for data and file exchange with external systems (LIMS).
- Present the final look and feel of the Digital Twin User Interface.

3. INTRODUCTION

Work Package 5 (WP5) constitutes the core development phase for the Digital Twin (DT) of the battery systems within the FASTEST project. The journey began with the definition of the semantic framework (D5.1) and the structuring of DT components (D5.2). Subsequently, Deliverable D5.3 laid the theoretical foundation, defining the system architecture, the integration plan, and the requirements for the platform.

This deliverable, D5.4 "Digital Twin integration", represents the transition from design to execution. It reports the activities performed within Task 5.4, detailing the actual deployment, configuration, and technical integration of the Digital Twin solution into the project's ecosystem.

While D5.3 outlined "how the system should be built", D5.4 describes "how the system has been built and configured". This document provides a transparent and technical overview of the Digital Twin implementation on the Azure Cloud infrastructure, utilizing the specific technologies selected during the design phase: Docker for containerization and Kubernetes for orchestration.

The document covers three main pillars of the integration process:

1. Platform Deployment & Orchestration: A detailed description of the microservices ecosystem (including Data Collector, Analysis Service, User Interface, and Model Management) deployed on the Azure Virtual Machine. It details the configuration of the Kubernetes cluster, the setup of the Container Registry, and the implementation of the security layer using Keycloak and an API Gateway to manage Authentication and Authorization.
2. Integration with LIMS (WP6): A comprehensive explanation of the connectivity established between the DT platform and the LIMS platform. This includes the configuration of the MQTT Bridge for real-time bidirectional data exchange and the specialized implementation of an

Azure Private Endpoint to enable secure SFTP transfer of FMU models without compromising network security.

3. User Interface: The document provides a walkthrough of the deployed User Interface, accessible via web browser. It highlights how the Models Exchange Interface has been seamlessly embedded using frontend integration techniques (Iframe).

By validating these technical configurations, this deliverable confirms that the Digital Twin platform is fully operational, integrated with the testing infrastructure, and ready to support the battery validation use cases defined in the project.

4. DEPLOYMENT OF DIGITAL TWIN PLATFORM

This section details the technical transition from theoretical design to the physical deployment of the Digital Twin platform within a cloud-native environment. It outlines the orchestration of microservices via Kubernetes, the establishment of robust security protocols through Keycloak, and the final integration of the frontend interface.

4.1 Cloud Infrastructure and Orchestration

The Digital Twin platform is hosted on a Linux Virtual Machine deployed within the Azure Cloud environment. To ensure scalability and efficient management of the software components, Kubernetes has been installed on the VM to act as the container orchestrator.

A dedicated Container Registry was set up on the cloud to store the Docker images of the DT microservices. During the deployment phase, Kubernetes pulls the specific images from this registry to instantiate the pods.

4.2 Microservices Deployment Configuration

The Digital Twin platform is built upon a microservice architecture, ensuring that each functional component is decoupled, scalable, and independently deployable. The entire ecosystem is hosted on a dedicated Linux Virtual Machine within the Azure Cloud, with Kubernetes installed to act as the container orchestrator [1].

All DT components (Data Collector, Analysis Service, Communication Monitoring, User Interface, Models Exchange Interface, Test Request Handler, Models Management) along with the supporting infrastructure services (RabbitMQ Broker, MongoDB) are deployed as pods within the Kubernetes cluster.

The deployment process involves three main stages: Image management via the Container Registry, the definition of Kubernetes resources (Deployments and Services), and the runtime configuration via environment variables.

4.2.1 Private Container Registry Setup

To manage the lifecycle of the software artifacts securely, a private Container Registry was established on the Azure Cloud. This registry serves as the central repository for all Docker images of the Digital Twin microservices.

- Workflow: All the docker images are pushed to this private registry.
- Security: Access to the registry is restricted. The Kubernetes cluster on the Linux VM is configured with specific credentials (image pull secrets) to authenticate against the registry and pull the latest versions of the microservices during the deployment phase.

4.2.2 Kubernetes Orchestration: Deployments and Services

For each microservice, specific Deployment and Service YAML files were defined to manage replicas, networking, and environment variables.

1. **Deployment configuration**

The Deployment resource defines:

- Container Image: The specific image path and tag to be pulled from the Container Registry.
- Replicas: The number of identical pods to run for high availability.
- Resources: CPU and Memory limits are defined to ensure fair usage of the VM's resources.
- Environment Variables**:** Runtime configurations (such as Database connection strings, RabbitMQ credentials) are injected into the containers as environment variables.

2. **Service Configuration (service.yaml) and External Accessibility**

The Service resource defines the networking rules to expose the pods. Depending on the visibility requirements of each component, different Service types have been adopted:

- ClusterIP (Internal Communication): The majority of the backend microservices (such as Data Collector, Analysis Service) are configured with the default ClusterIP type. This assigns an internal IP address accessible only within the cluster. This design choice adheres to the principle of least privilege, ensuring that backend logic and databases are not directly exposed to the public internet but can communicate securely with each other via internal DNS (e.g., http://mongodb-service:27017).
- NodePort and LoadBalancer (External Access): Specific components require accessibility from outside the Kubernetes cluster to interact with end-users or external platforms (LIMS). To achieve this, the service.yaml definitions for the RabbitMQ Broker, User Interface, and Models Exchange Interface were configured using NodePort or LoadBalancer types.
    - RabbitMQ Broker: Exposed to allow the establishment of the MQTT Bridge connection from the external LIMS platform and to accept telemetry data ingestion.
    - User Interface: Exposed to serve the frontend application to web browsers over the HTTPS protocol.

  - Models Exchange Interface: Exposed to allow direct interaction and proper rendering within the Iframe component of the main dashboard.

By mapping these services to specific ports on the host node (NodePort) or assigning them an external IP (LoadBalancer), the architecture ensures that the entry points are clearly defined and manageable, while keeping the rest of the ecosystem isolated.

3. Configurations and Secrets

In order to allow the operativeness of the software components and services previously defined as Kubernetes resources, some Secrets (Kubernetes objects which aim is to contain sensitive data) and ConfigMaps (Kubernetes objects for storing non-confidential data in key-value pairs) have been created [2], [3]. Specifically, the following information has been stored:

- Nginx server configurations: managed as Secrets and protected by Azure Key Vault.
- Keycloak realm and domain configurations: these define user levels, groups, and realm operations.
- Certificate paths: used for SSL authentication.
- DB, Storage Account, Eureka and Keycloak utilities credentials, managed as config maps.

4.2.3 Deployed Components

The deployment consists of a comprehensive set of microservices categorized into Infrastructure, Security, and Business Logic layers. To optimize resource usage and streamline the communication between tightly coupled functionalities, specific logical components (Test Request Handler, Models Management, and Models Exchange Interface) have been consolidated into a single deployment unit.

The complete list of deployed components currently running in the Kubernetes cluster is detailed below:

1. *Infrastructure & Security Layer*
   - API Gateway (Spring Cloud Gateway): This service acts as the single entry point for all client requests. It handles routing to the appropriate backend microservices and works in conjunction with Keycloak to enforce security policies.
   - Identity and Access Management (Keycloak): A dedicated Keycloak instance is deployed to manage user identities, roles, and authentication flows. It secures the APIs by validating tokens before requests reach the business logic.
   - Communication Monitoring (Eureka Server): This component acts as the Service Registry. While Kubernetes handles internal DNS, Eureka is utilized here specifically for application-level health monitoring, ensuring all registered services are up and running.

- Message Broker (RabbitMQ): The central message broker that facilitates asynchronous communication between microservices and enables the external MQTT Bridge connection with LIMS.
- MongoDB: The NoSQL persistence layer used by the Digital Twin components (Data Collector, Analysis Service).
- PostgreSQL Database: A relational database instance deployed specifically to support the security infrastructure. It serves as the backend persistence layer for Keycloak (storing user credentials, realms, and client configurations) and for the Spring Cloud Gateway data.

2. *Core Business Logic Layer*
    - Digital Twin Core Service (Consolidated Pod): To maximize efficiency, the following three logical modules have been integrated into a single Docker image and deployed as a unified container:
        - *Test Request Handler*: Manages the workflow of incoming test requests and schedules.
        - *Models Management*: Manage the simulation models and testing procedures.
        - *Models Exchange Interface*: Models management user interface that facilitates the upload and storage of the simulation models. This interface allows for direct uploading without having internal access.
    - Data Collector: A specialized standalone microservice dedicated to high-throughput ingestion of real-time telemetry data coming from LIMS via MQTT.
    - Analysis Service: Responsible for processing the raw data ingested by the Data Collector, performing aggregation, and preparing data for visualization.
3. *Frontend Layer*
    - User Interface: The web server hosting the frontend application. It interacts with the API Gateway to retrieve data and renders the graphical dashboard for the end-users.

4.3 Authentication and Authorization (API Gateway & Keycloak)

Security is managed through an API Gateway coupled with Keycloak for Identity and Access Management (IAM).

- API Gateway: Acts as the single entry point for all backend API requests. It is configured to route traffic to the appropriate microservices based on the request path.
- Keycloak Configuration: A specific realm was configured to handle users, roles, and clients. All backend endpoints configured on the API Gateway are protected, requiring a valid OIDC (OpenID Connect) token [4]. The Gateway validates the token against Keycloak before forwarding the request to the internal microservices.

The gateway manages the entire lifecycle of an incoming request through a defined sequence of steps:

1. **Client Request:** A client initiates a network request directed to the external address of the API Gateway.
2. **Route Matching (Predicates):** The Gateway compares the incoming request against a defined set of routes. Each route specifies one or more predicates (e.g., matching the URL path segment, HTTP method, or host) that determine if the request should be processed by that specific route.
3. **Pre-Request Filtering:** Once a match is found, the Gateway executes pre-request logic defined by the filters associated with the matched route. Common pre-request actions include adding or modifying query parameters or headers.
4. **Authentication and Authorization:** The Gateway enforces security policies in cooperation with Keycloak.
    - **Role-Based Access Control (RBAC):** The Gateway verifies the client's authorization based on the roles and required HTTP methods defined for the target path. A valid JSON Web Token (JWT) is mandatory for protected endpoints.
    - **Tenant Verification:** It performs an additional security check by verifying that the identifier specified in the request header matches one of the authorized identifiers contained within the client's JWT.
5. **Proxy and Header Injection:** The request is then routed to the correct internal microservice (proxied service). Before forwarding, a proxy filter adds crucial context headers to the request for internal service consumption, including the identifier, the user name, and the related user role extracted from the validated JWT.
6. **Service Execution:** The target microservice executes the requested operation and returns a response to the Gateway.
7. **Post-Request Filtering:** Upon receiving the service's response, the Gateway executes post-request logic defined by its filters. This typically involves modifying the response headers (e.g., removing unwanted sensitive headers) before transmitting the final response back to the client.

All new API routes for the Digital Twin application must be explicitly defined and configured in the Spring Cloud Gateway's YAML configuration file to ensure proper routing and security enforcement across the platform.

Following a sequence diagram of the communication and token validation to GET any resource in the backend:

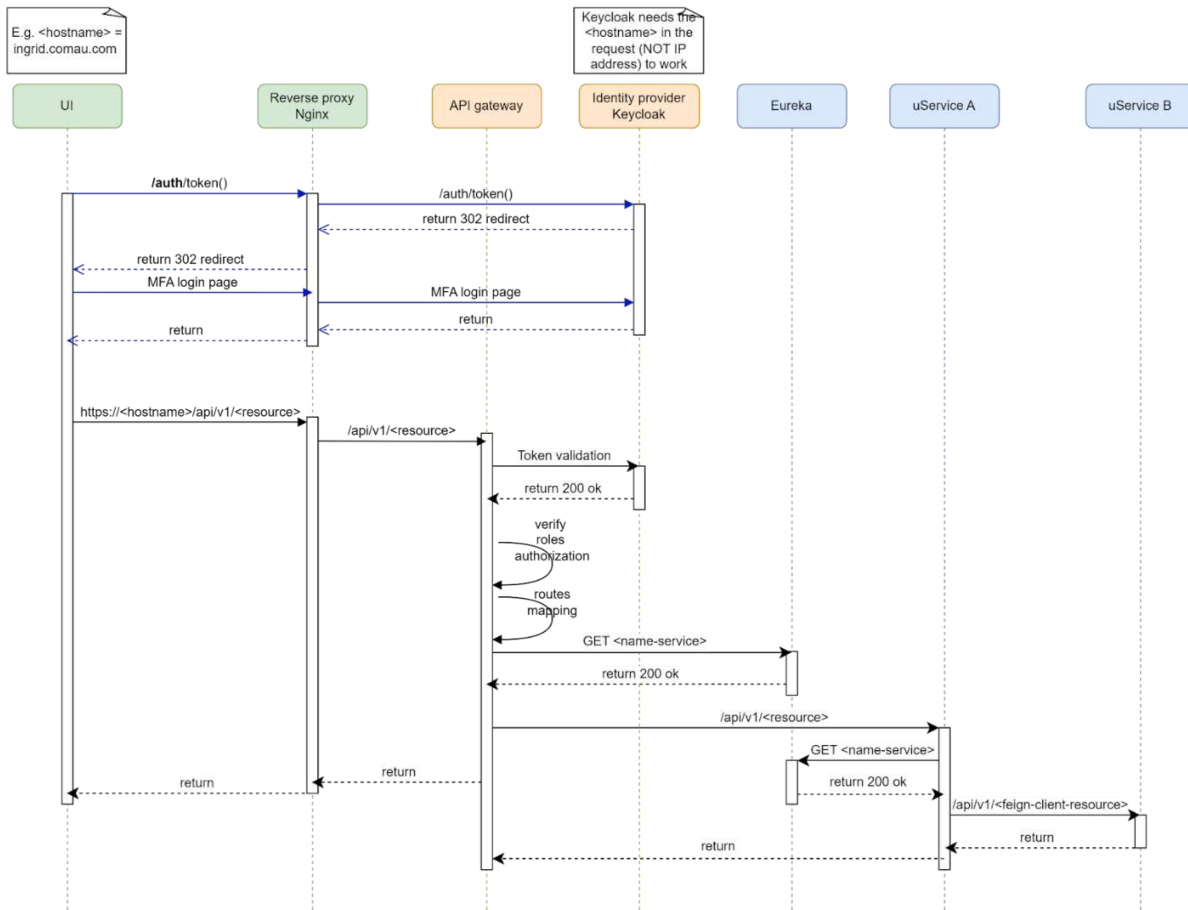

Figure 1 Communication and validation diagram

To enable the Spring Cloud Gateway to connect with Keycloak and validate tokens for securing the Digital Twin microservices, the following configuration steps and components are essential. The configuration involves setting up a dedicated Keycloak Client for the API Gateway within the established realm, and configuring the Gateway itself to act as an OAuth 2.0 Resource Server.

A specific Keycloak Client must be created and configured to represent the Spring Cloud Gateway application.

Setting	Value	Rationale
Client ID	`dt-api-gateway` (Example)	Unique identifier for the Gateway application.
Client Protocol	`openid-connect`	Standard protocol for token validation.
Access Type	`bearer-only`	The Gateway's primary role is to validate incoming Bearer tokens (JWTs) from clients (UI) before routing. It does not initiate login flows.

Setting	Value	Rationale
Standard Flow Enabled	OFF	Not needed for a `bearer-only` client.
Valid Redirect URIs	(Empty)	Not applicable for a `bearer-only` client.
Web Origins	`+` (or specific domain)	Allows the client to accept tokens originating from the specified frontend URL (e.g., the User Interface).
Roles	Defined Roles (e.g., `reader`, `administrator`)	These roles are assigned to users and included in the JWT. The Gateway uses these to enforce Role-Based Access Control (RBAC) on specific routes.

Table 1 Keycloak Client configuration

On the other hand, the Spring Cloud Gateway must be configured as a Resource Server to use the Keycloak server for token verification. This is typically done in the `application.yml` or `application.properties` file of the Spring Boot application.

Configuration Key	Example Value	Description
`spring.security.oauth2.resourceserver.jwt.issuer-uri`	`https://[Keycloak-URL]/realms/[Realm-Name]`	Specifies the base URL of the Keycloak realm. The Gateway uses this to retrieve the necessary public keys (via JWKS endpoint) for local JWT signature validation.
`spring.security.oauth2.resourceserver.jwt.jwk-set-uri`	(Typically derived from `issuer-uri`)	The URI where the Gateway fetches the JSON Web Key Set (JWKS) to verify the authenticity of the incoming JWT.

Table 2 Configuration of Spring Cloud Gateway

Finally, the API Gateway performs token validation via the following filters:

- **Token Parsing:** The Gateway extracts the JWT (JSON Web Token) from the `Authorization: Bearer <token>` header of the incoming request.

- **Signature Verification:** Using the public keys fetched from Keycloak's JWKS endpoint, the Gateway verifies the JWT's signature to ensure the token has not been tampered with and was issued by the trusted Authorization Server (Keycloak).
- **Claim Validation:** The Gateway checks key claims within the JWT:
  - `exp` **(Expiration):** Ensures the token is not expired.
  - `iss` **(Issuer):** Confirms the token was issued by the configured Keycloak realm.
  - `aud` **(Audience):** Confirms the token is intended for the Digital Twin platform.
- **Role-Based Access Control (RBAC):** As detailed in Section 4.3, the Gateway applies filters to specific routes, requiring the presence of certain roles (e.g., `dt_admin`) within the token's claims to grant access to the target microservice.
- **Header Injection:** Once validated, the Gateway extracts user context (User ID, Role, etc.) from the token and injects this information into new headers (e.g., `X-User-ID`, `X-User-Role`). These headers are then forwarded to the internal microservices, allowing them to rely on the Gateway for security enforcement and simplifying their own security logic.

4.4 Frontend Integration (Iframe and Nginx Configuration)

The Digital Twin User Interface consolidates different functionalities into a single web portal. A key integration challenge was embedding the Models Exchange Interface within the main Digital Twin UI. This was achieved using an Iframe component.

Nginx Configuration: To allow the seamless rendering of the external service within the main UI frame without CORS issues or routing errors, the Nginx server (acting as a reverse proxy) was configured to handle specific redirects.

- Endpoint Redirection: Nginx intercepts requests directed to the Models Exchange path and proxies them to the internal service port.
- Iframe Implementation: The frontend code utilizes an HTML Iframe component to load the specific view.

4.4.1 Nginx Configuration Details

As illustrated in Figure 2, the Nginx server acts as a Reverse Proxy, serving as the single-entry point for the application [5]. This setup is crucial for resolving CORS (Cross-Origin Resource Sharing) issues and managing complex routing between the frontend and various backend services.

- Static Content Delivery (`location /`): Nginx serves the compiled frontend assets directly from the `/www/fastest` directory. The inclusion of the Content-Security-Policy header with the upgrade-insecure-requests directive ensures that all communication remains encrypted.

- API Routing (`location /api/`): Requests directed to the backend are proxied to the spring-gateway-application on port `8090`. By routing these requests through the same domain as the frontend, the browser treats them as "same-origin," eliminating the need for complex CORS configurations on the microservices.
- Authentication and Identity Management (`location /auth/`): This block handles communication with Keycloak. The configuration uses `proxy_set_header` directives to pass the original client's IP, protocol, and host information. This allows Keycloak to accurately manage redirects and security tokens while remaining hidden behind the proxy.

```
# serve static files
location  /  {
    root     /www/fastest;

    add_header Content-Security-Policy "upgrade-insecure-requests";
}

location /api/ {
    proxy_pass      http://spring-gateway-application:8090/api/;
}

location /auth/ {
    proxy_pass https://keycloak/auth/;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Port $proxy_protocol_port;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_set_header Host $host;

    add_header Content-Security-Policy "upgrade-insecure-requests";
}
```

Figure 2 Configuration of Nginx server

4.4.2 Iframe Implementation

The frontend integration is handled by a dedicated Angular component, MeiComponent, designed to embed the Models Exchange Interface (MEI) seamlessly within the Digital Twin layout.

Because modern web frameworks like Angular implement strict security policies to prevent Cross-Site Scripting (XSS), external URLs cannot be bound directly to an Iframe's src attribute. To resolve this:

- The component utilizes the `DomSanitizer` service.
- The `bypassSecurityTrustResourceUrl` method is called to explicitly trust the external MEI URL (`https://fastest.ingrid.comau.com/inegi/login`), allowing it to be loaded safely within the application context.

The successful loading of the interface via `https://fastest.ingrid.comau.com/inegi/login` is not merely a frontend redirect, but the result of a coordinated effort between the Angular application and the Nginx reverse proxy. When the Iframe requests the `/inegi/` path, Nginx intercepts this request. Instead of looking for a local folder, it proxies the traffic to the internal service port where the Models Exchange Interface is hosted. By serving the MEI through the same main domain, `fastest.ingrid.comau.com`, Nginx ensures the browser treats the Iframe as a "same-site" resource. This is critical for maintaining session consistency and passing authentication headers without triggering restrictive browser security blocks. This configuration allows the frontend to remain "agnostic" of where the MEI service actually resides in the backend network. The Angular component simply points to the public-facing URL, while Nginx ensures the request reaches the correct server destination.

```
@Component({
  selector: 'app-mei',
  imports: [],
  template: `<iframe class="w-full h-[calc(100dvh-80px)]" [src]="iframeUrl"></iframe>`,
  styleUrl: './mei.component.scss',
})
export class MeiComponent {
  private sanitizer = inject(DomSanitizer);

  iframeUrl = this.sanitizer.bypassSecurityTrustResourceUrl('https://fastest.ingrid.comau.com/inegi/login');
}
```

Figure 3 Definition and the implementation of MeiComponent (Models Exchange Interface Components) in Angular. We can see also the template definition and the sanitized Iframe url

5. SYSTEM INTEGRATION CONFIGURATION

This section focuses on the implementation of a bi-directional MQTT bridge and a secure Azure Private Endpoint for the protected transfer of Functional Mock-up Units (FMUs) between the DT and LIMS.

5.1 MQTT Bridge Setup

As planned in D5.3, communication between the DT platform and LIMS relies on an MQTT Bridge. This bridge connects the RabbitMQ broker (DT side) with the HiveMQ broker (LIMS side).

The bridge is configured to forward messages bi-directionally on specific pre-agreed topics. The bridge configuration is done on HiveMQ broker, using the HiveMQ bridge extension.Although initially planned, the use of broker-side TLS certificates was reconsidered as communication channel security is managed upstream through network infrastructure with IP whitelisting.

5.2 Secure Model Transfer (Azure Private Endpoint)

A critical requirement for the project is the secure transfer of functional mock-up units (FMU models) from the DT Test Request Handler to the LIMS/Virtual Test Bench. To facilitate this via SFTP without exposing the LIMS storage to the public internet (and avoiding the security risk of whitelisting public IPs), an Azure Private Endpoint was implemented.

This configuration links the Digital Twin's Azure VM directly to the LIMS Storage Account using the Azure backbone network. This ensures that file transfers occur over a private, secure link with lower latency and higher security compliance.

To verify the correct configuration of the Azure Private Endpoint and ensure that the application logic can successfully transfer models, a connectivity test was executed directly from the runtime environment. Access was established via shell into the consolidated microservice container (hosting the Test Request Handler, Models Management, and Exchange Interface). From within this isolated pod, an SFTP session was initiated towards the LIMS Azure Blob Storage using the private IP address resolved by the endpoint.

The image below demonstrates the successful outcome of this validation: a connection is established, authentication is accepted, and a sample test file is successfully uploaded to the target remote directory. This confirms that the secure data pipeline is active and capable of handling FMU model transfers without traversing the public internet.

```
root@inegi-interface-app-76b44445cc-fjf76:/# ls
bin   database.db  entrypoint.sh  filebrowser  healthcheck.sh  inegirabreu  lib64  mnt                  opt   root  sbin  sys  usr
boot  dev          etc            get.sh       home                         lib    media  mqtt_subscriber.py  proc  run   srv   tmp  var
root@inegi-interface-app-76b44445cc-fjf76:/# chmod 600 inegirabreu
root@inegi-interface-app-76b44445cc-fjf76:/# sftp -i ./inegirabreu stwefastest01.inegirabreu@stwefastest01.blob.core.windows.net
Connected to stwefastest01.blob.core.windows.net.
sftp> put srv/
Cell/                Module/                Safety and Reliability/   read copy.txt           read.json
sftp> put srv/Cell/Overcharge/simulation.txt
Uploading srv/Cell/Overcharge/simulation.txt to /simulation.txt
srv/Cell/Overcharge/simulation.txt                                                          100%   13    11.8KB/s   00:00
sftp> exit
root@inegi-interface-app-76b44445cc-fjf76:/# ^C
root@inegi-interface-app-76b44445cc-fjf76:/# ^C
root@inegi-interface-app-76b44445cc-fjf76:/# exit
command terminated with exit code 130
```

Figure 4 Terminal output showing the successful execution of an SFTP command from inside the Digital Twin container. The log confirms the connection to the LIMS storage account and the upload of a test file via the Azure Private Endpoint.

6. VERIFICATION OF INTER-PLATFORM INTERACTIONS (DT & LIMS)

This section shows the interactions tested between the two platforms, confirming that the MQTT bridge and the SFTP pipeline works correctly.

6.1 Interaction Log

Interaction N. 15a: Forward the fetched model files

- **From:** DT - Test Request Handler
- **To:** Virtual Test Bench (LIMS Storage)
- **Protocol:** SFTP (via Azure Private Endpoint)
- **Data Type:** FMU Files
- **Description:** The system successfully authenticated via the private endpoint and uploaded the `.fmu` file to the target container in the Blob Storage.

Interaction N. 17b: Push real-time test results

- **From:** LIMS
- **To:** DT - Data Collector
- **Protocol:** MQTT
- **Topic:** LIMS/metrics/veos/simulation_metrics

Test success evidence is shown below:

FASTEST 106

+ New Subscription

LIMS/metrics/veo... QoS 0

Plaintext

All | Received | Published


```
            "name": "Voltage",
            "type": "Input",
            "unit": "V",
            "values": [ { "timestamp" : 1050, "value" : 0.862 } ]
            },
            {
            "name": "Average_Temperature",
            "type": "Input",
            "unit": "degC",
            "values": [ { "timestamp" : 1050, "value" : 1.725 } ]
            },
            {
            "name": "SoC",
            "type": "Input",
            "unit": "%",
            "values": [ { "timestamp" : 1050, "value" : 1.725 } ]
            },
            {
            "name": "SoH",
            "type": "Input",
            "unit": "%",
            "values": [ { "timestamp" : 1050, "value" : 1.725 } ]
            },
            {
            "name": "Temperature_Diff",
            "type": "Input",
            "unit": "degC",
            "values": [ { "timestamp" : 1050, "value" : 0.862 } ]
            }
        ]
    },
    "test_date": "2026-04-13T00:00:00.0000000",
    "status": { "value" : "DT Complete", "last_update" : "2026-04-13T13:54:21.5516767Z" }
}
```


2026-04-13 15:58:33:677

Figure 5 Test result LIMS forwarding test results to DT via MQTT

7. USER INTERFACE FUNCTIONALITIES

This section explores the graphical user interface (GUI) and the functional features designed to facilitate user interaction with the Digital Twin platform. It provides a comprehensive walkthrough of the platform's front-end, ranging from secure multi-factor authentication (MFA) to the advanced visualization of test results and Model Exchange Interface component.

7.1 GUI Sections and Walkthrough

The Digital Twin User Interface is a single-page application developed using the Angular framework. The interface is designed to be intuitive, responsive, and secure, ensuring that only authorized personnel can access sensitive test data and models.

The application workflow covers three main phases: Authentication, Monitoring (Test Summary), and Model Management.

7.1.1 Registration and Secure Login (MFA)

Access to the platform is protected by a strict authentication and authorization flow managed by Keycloak.

New users must first register an account by clicking the "Register" button on the landing page. The registration form requires personal information including Email, Password, First Name, and Last Name. To comply with the security requirements, the platform enforces Multi-Factor Authentication (MFA), in fact, by clicking again on the “Register/Next” button, the user is required to configure a Mobile Authenticator.

1. The user must install a compatible OTP application (e.g., Microsoft Authenticator, Google Authenticator, or FreeOTP) on their mobile device.
2. The user scans the QR code displayed on the screen using the mobile app.
3. The user enters the one-time code generated by the app and clicks "Submit" to finalize the setup.

Once registered, users can access the platform by entering their email and password, followed by the specific OTP code generated by their mobile authenticator.

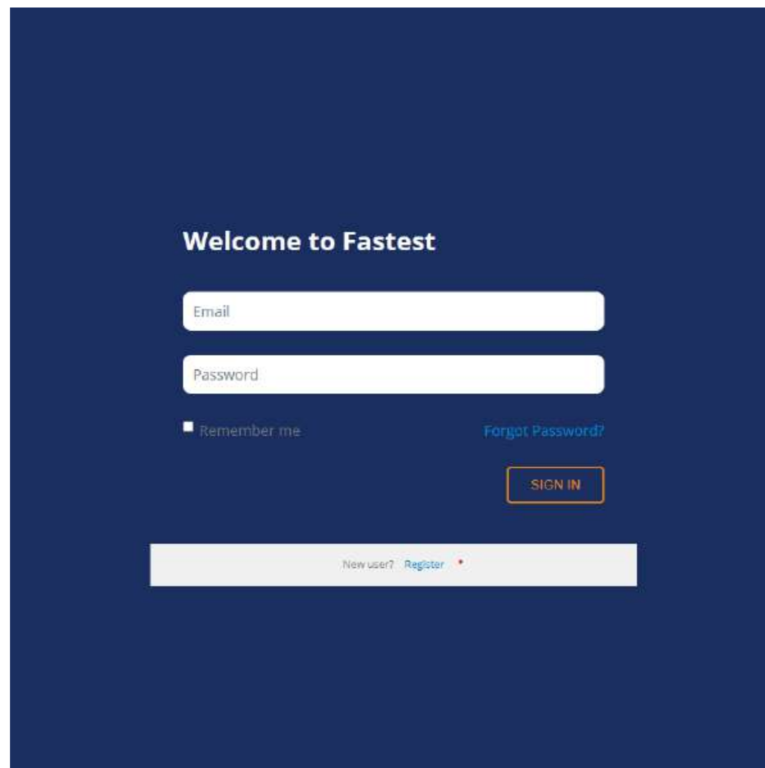

Figure 6 Fastest UI

Figure 7 Registration process

7.1.2 Dashboard and Navigation

Upon successful authentication, the user is redirected to the main Dashboard. The layout has been optimized to provide immediate access to the core functionalities. The header features global controls:

- Theme Toggle: A button in the top-right corner allows users to switch between Light and Dark modes, improving visibility according to user preference.
- Logout: Allows the user to securely disconnect from the session.

The main navigation provides access to the two primary operational sections:

1. Test Summary: The control center for monitoring ongoing and completed tests.
2. Models Exchange Interface: The integrated environment for managing simulation models.

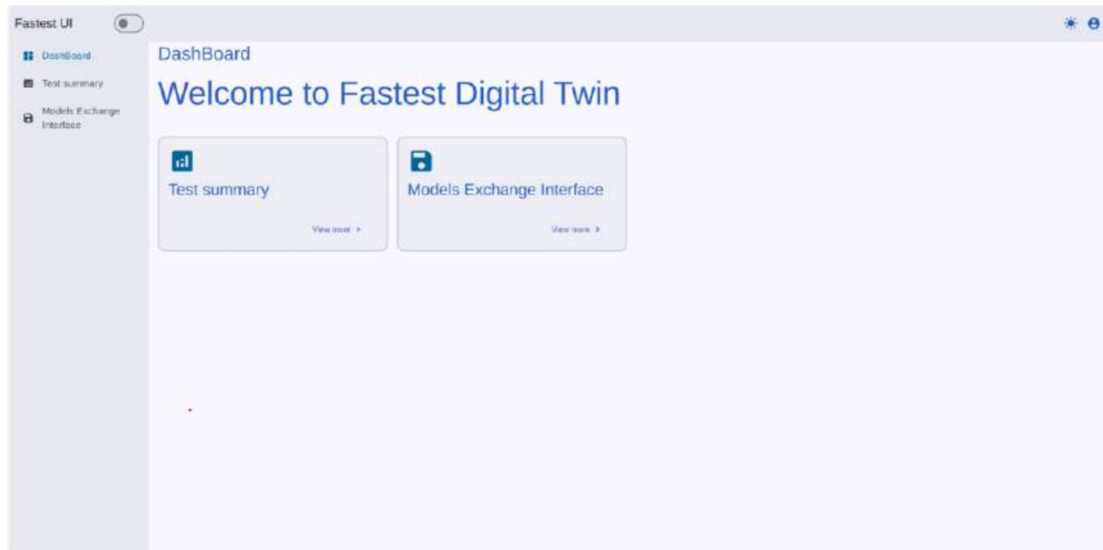

Figure 8 Digital twin dashboard

7.1.3 Test Summary Section

The "Test Summary" page has been redesigned to offer a comprehensive view of the testing ecosystem in a single glance. The page structure is divided into three logical blocks:

1. **High-Level Metrics (Top)**
   At the top of the page, a series of counters provide real-time statistics on the testing workload, categorized by test level (Cell, Module, Pack) and status (In Progress, Waiting, Completed).
   - *Overall Total Tests* (Cell/Module/Pack)
   - *In Progress Tests* (Cell/Module/Pack)
   - *Waiting Tests* (Cell/Module/Pack)
   - *Completed Tests* (Cell/Module/Pack)
2. **Filtering (Center)**
   A dedicated filtering bar allows operators to quickly locate specific tests within the database. Users can filter the list by Test Name or by Progress status.
3. **Test List (Bottom)**
   The main table lists all physical and virtual tests (scheduled, running, completed or failed) with the following key details:
   - Test Name: (e.g., Overcharge)
   - Test Type: (e.g., Cell-level)
   - UUT (Unit Under Test): The identifier of the physical component (e.g., Cell-01).
   - Date: The scheduled timestamp of the test.
   - Progress: The current status (e.g., Completed, Running, Waiting).

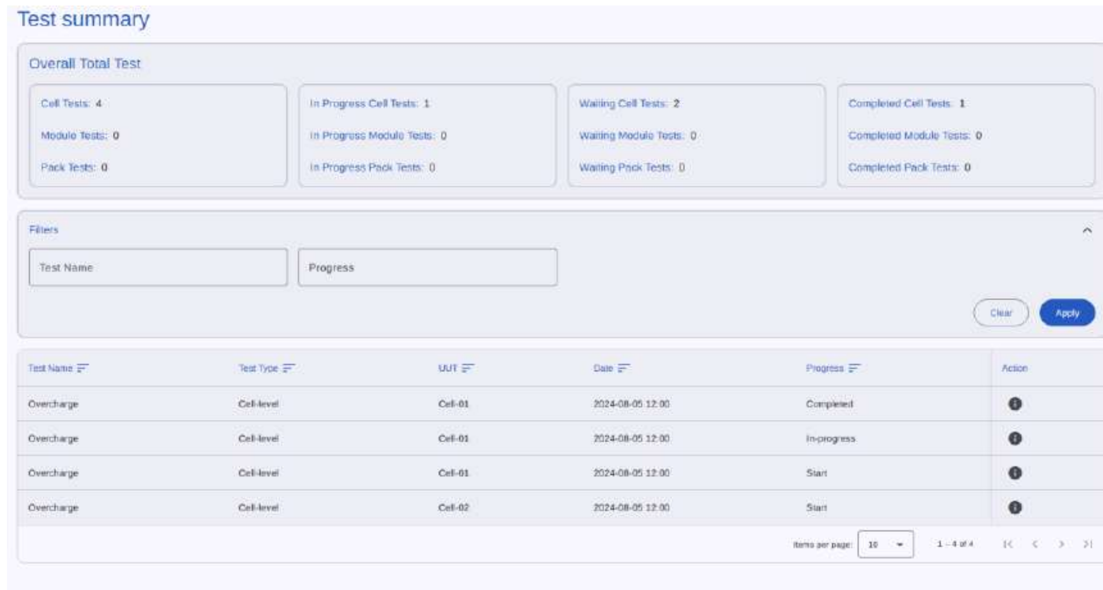

Figure 9 Test summary page

7.1.4 Test Details and Results Visualization

By clicking the "Action" button associated with any specific test in the summary list, a detailed modal window opens. This view merges the metadata information with the results visualization, replacing the separate "Trends" section described in previous deliverables.

The upper section of the window displays the static registry data of the test:

- ID & UUID: Unique identifiers for traceability.
- Test Name & Type: Configuration details.
- Test Bench: Indicates if the test is Physical or Virtual.
- Status & Date: Execution timestamp and final outcome.

The lower section features an interactive Trends Chart. This dynamic graph plots the temporal trend of the Input and Output variables recorded during the test execution. Users can analyze the behavior of the UUT (e.g., voltage, temperature curves) directly within this context.

Figure 10 Test details page

7.1.5 Models Exchange Interface

The "Models Exchange Interface" section allows users to manage the FMU files. From an architectural perspective, this interface belongs to a separate microservice but is seamlessly integrated into the main Digital Twin Dashboard via an Iframe. This integration allows users to upload, version, and share models without leaving the main application context, providing a unified user experience.

Architectural Context

The Models Exchange Interface microservice is designed as an independent backend component responsible for simulation model management and test procedure orchestration within the Digital Twin ecosystem.

From an architectural perspective, the Models Exchange Interface relies on the following core technical components:

- **MySQL Database (Test Procedures and Metadata Storage):** The microservice uses a MySQL relational database as its persistence layer. The database stores structured information about test procedures. The test procedures stored in MySQL are retrieved dynamically when a test execution is triggered. These procedures are then included in the MQTT message sent to the service responsible for running the tests. This ensures that the execution environment receives a fully defined and consistent set of instructions directly derived from database source.
- **MQTT Communication Capability:** The Models Exchange Interface can establish MQTT connections with other services in the ecosystem. It subscribes to test execution requests and, after retrieving the corresponding test procedures from the MySQL database, publishes a structured MQTT message containing:
    - The original test request information
    - The complete test procedure

    - The simulation model reference and location
- **SFTP Connectivity:** The microservice is capable of establishing secure SFTP connections to remote machines. This allows the transfer of the required simulation file to the target execution environment before the final MQTT execution message is published.

Additionally, the Models Exchange Interface manages a structured directory system dedicated to simulation file storage. The folder organization supports:

- Storage of multiple simulation models
- Management of different versions of the same model
- Logical grouping by test type or project
- Automatic selection of the latest version by default

By combining MySQL-based test procedure storage, MQTT-based message orchestration, secure SFTP file transfer, and structured version-controlled file storage, the Models Exchange Interface functions as a centralized and consistent bridge between simulation management and automated test execution services.

Models Exchange Interface Functionalities

This section provides a detailed description of the functionalities implemented in the architecture, outlining the operational workflow and integration mechanisms. Each functional element is explained individually.

1. **Reception of MQTT Test Request**

The DT platform receives a structured MQTT message; this message originates from the LIMS side and represents a request to execute a specific test.

2. **Retrievel of Test Procedures from local database**

Upon message reception, the background service performs a look up operation in its internal database, using the test identifier received in MQTT message to retrieve the associated test procedures.

3. **Identification of the Corresponding Simulation File**

The Models Exchange Interface stores uploaded simulation models. Once the correct model reference is identified, the system determines the physical storage location of the simulation file within the DT infrastructure. The latest version is selected by default unless explicitly specified otherwise in the incoming MQTT request.

4. **Secure Transfer via SFTP**

After retrieving the correct simulation file, the system initiates a secure file transfer toward the machine responsible for executing the test. The transfer is performed using SFTP over the Azure Private Endpoint connection. The file is copied to a predefined execution directory on the target machine. The exact destination path is included in the publish MQTT message for traceability.

5. **Publication MQTT Message**

Once the file transfer is successfully completed, the Models Exchange Interface microservice publishes a new MQTT message.

- This outgoing message contains:
- The original MQTT payload received from LIMS
- The retrieved test procedures
- The absolute location of the simulation file on the execution machine

This enriched message acts as the final execution command for the Models Exchange Interface. It ensures that the execution environment has all required contextual information: configuration parameters, procedural logic, and the physical path to the simulation model.

8. CONCLUSION

The work performed in Task 5.4 has successfully achieved the deployment and integration of the Digital Twin platform. The system is now fully operational on the Azure Cloud, with a robust microservice architecture secured by industry-standard authentication protocols.

By leveraging Kubernetes as the orchestrator, we have achieved a scalable and resilient infrastructure where all microservices, ranging from the Data Collector to the consolidated Digital Twin Core Service, are effectively containerized and managed. The implementation of a layered architecture, featuring an API Gateway and Keycloak backed by PostgreSQL, ensures that the platform meets high standards of security, authentication, and authorization, protecting the system from unauthorized access while maintaining flexibility for legitimate users.

A focal point of this deliverable was the technical integration with the LIMS platform (WP6). We have demonstrated that the connectivity challenges have been overcome through targeted technical solutions:

- Real-time Communication: The MQTT Bridge configuration between the RabbitMQ (DT) and HiveMQ (LIMS) brokers has been validated, enabling seamless bi-directional exchange of test schedules and telemetry data.
- Secure Model Transfer: The deployment of the Azure Private Endpoint proved to be the optimal solution for the secure transmission of FMU files. This approach enabled the establishment of an SFTP channel directly to the LIMS storage without exposing endpoints to the public internet, thus adhering to strict IT security compliance requirements.

Finally, the deployment of the web-based User Interface provides stakeholders with a unified and accessible portal to monitor operations, manage digital models, and analyze battery performance data in real-time.

9. REFERENCES

[1] https://kubernetes.io/docs/concepts/overview/.

[2] https://kubernetes.io/docs/concepts/configuration/secret/.

[3] https://kubernetes.io/docs/concepts/configuration/configmap/.

[4] https://www.keycloak.org/.

[5] https://nginx.org/.