EUROPEAN COMMISSION

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

FASTEST

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

Deliverable D6.3: Integration of DoE algorithms to hybrid platform

Primary Author [Dr. Shuchen Liu]

Organization [FEV]

Date: [12.02.2026]

Doc. Version: [V1.0]

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 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:	WP6
Deliverable:	Integration of DoE algorithms to hybrid platform
Deliverable Type:	R – Document, report
Dissemination Level:	PU - Public
Due Date:	30.11.2025 (Month 30)
Actual Submission Date:	12.02.2026
Pages:	< 29 >
Doc. Version:	V1.0
GA Number:	101103755
Project Coordinator:	Bruno Rodrigues | ABEE (bruno.rodrigues@abeegroup.com)

Formal **Reviewers**		
Name	Organization	Date
Koen Vanden Boer	**FM**	30/01/2026
Mariana Fernández	**SIE**	21/01/2026
Nuno Marques	**INEGI**	22/01/2026
Laura Oca	**MGEP**	20.01.2026

Document History			
Version	Date	Description	Author
0.1	16.01.2026	First Draft	Peter May (FEV) Dr. Shuchen Liu (FEV) Murat Bayraktar (FEV) Felix Pischinger (FEV)
0.2	20/01/2026	Inputs, changes and comments from MGEP review	Laura Oca (MGEP)

0.3	21/01/2026	Inputs, changes and comments from SIE review	Mariana Fernández (SIE)
0.4	26/01/2026	Integration of inputs and comments	Peter May (FEV) Dr. Shuchen Liu (FEV)
0.5	11/02/2026	Change of Figure 2 to align with changes discussed in GA M33 online	Peter May (FEV) Dr. Shuchen Liu (FEV)
1.0	12/02/2026	Submission version	Peter May (FEV) Dr. Shuchen Liu (FEV) Murat Bayraktar (FEV) Felix Pischinger (FEV) Dr. Laura Oca (MGEP) Mariana Fernández (SIE) Dr. Iván Sanz Gorrachategui (Ikerlan)

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 generalized 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 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, virtualized benches, and digital twin (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 physics-based and data driven models able to substitute physical characterization experiments; and the overarching DT architecture managing the information flows, and the TRL 6 proven and integrated prototype of the hybrid testing platform.

LIST OF ABBREVIATIONS, ACRONYMS AND DEFINITIONS

Acronym	Name
ACR	Azure Container Services
AKS	Azure Kubernetes Services
Blob	Binary large object
CLI	Command Line Interface
CSI	Container Storage Interface
CSV	Comma-Separated Values
DoE	Design of Experiment
DT	Digital Twin
ECU	Electronic Control Unit
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
GUI	Graphical User Interface
HiL	Hardware-in-the-Loop
IoT	Internet of things
IP	Internet Protocol
IPC	Inter-Process Communication
JSON	JavaScript Object Notation
LIMS	Laboratory Inventory Management System
MQTT	Message Queuing Telemetry Transport
OSA	Offline Simulation Application
PDF	Portable Document Format
PID	Process Identifier
POC	Proof Of Concept
QoS	Quality of Service
SFTP	Secure File Transfer Protocol
SiL	Software-in-the-Loop
UI	User Interface
UTS	Unix Timesharing System
UUID	Unit under test ID
UUT	Unit Under Test
VEOS	Virtual ECU Operating System
VPC	Virtual Private Cloud
XiL	X in the loop

LIST OF FIGURES

Table of Contents

1. EXECUTIVE SUMMARY

Deliverable D6.3 documents the integration approach and proof-of-concept (PoC) implementation for embedding Design of Experiments (DoE) into Laboratory Inventory Management System (LIMS) and executing the results of DoE on the virtual test benches. The work establishes a cloud-ready integration concept in which LIMS acts as the central orchestration and communication hub, interfacing with DoE and Digital Twin (DT) services while managing simulation assets, execution requests, and results in a consistent, scalable manner.

Today's battery test-facility landscape remains largely local, bench-centric, and resource intensive, with fragmented data handling and limited interoperability between physical testing, virtual models, and experiment planning. Within the FASTEST project, this translates into the need for a near real-time, scalable, and reliable communication and execution backbone that can retrieve optimized experiment parameters from DoE, retrieve the correct model artifacts from the Digital Twin (DT), and execute and observe tests across physical and virtual benches under strict latency and integrity requirements.

This deliverable addresses the challenge by defining and demonstrating an FMU-based co-simulation workflow and its cloud execution pattern (Azure/Kubernetes + Blob Storage), including standardized signal interfaces, interconnection via JSON specifications, automated build steps into VEOS OSA artifacts, and Message Queuing Telemetry Transport (MQTT) based publication of simulation outputs for consumption by LIMS/DoE/DT. The implementation is validated using synthetic FMUs for end-to-end connectivity testing and a functional output-collection FMU embedding an MQTT client to ensure consistent, structured telemetry exchange between platform components.

2. OBJECTIVES

The objective of D6.3 is to define and demonstrate the integration of FASTEST DoE services into the platform workflow, with LIMS orchestrating the retrieval of DoE outputs and DT model assets and triggering virtual test execution. Therefore, the deliverable D6.3 will describe the implementation of Functional Mock-up Units (FMU).

In addition, the deliverable establishes the interfaces, data exchanges, and execution workflow needed to run DoE-driven experiments on virtual benches and to return results in a form consumable by platform services. It will also contain validation checks and logging mechanisms that maintain data integrity during experiment runs.

3. INTRODUCTION

The following chapter focuses on laying the foundation of the project from a technical overview. First, the purpose of the test benches is explained, followed up by the problem statement, technical requirements and the proposed solution in the form of a cloud architecture design. Detailed descriptions on the deployment method of each service used in our architecture supports the context mentioned.

3.1 Purpose of conceptualizing virtual and physical tests

The purpose of conceptualizing the virtual and physical tests is to establish an understanding of how these tests interact with components of the proposed solution. After the design phase comes the implementation phase where the concept is tested and iteratively enhanced.

3.2 Problem description

In this work package, a central communication hub between the test benches (physical/virtual) and the other FASTEST system components (DoE/DT) is to be developed. The software interaction with DT [1] and DoE [2] is summarized as follows.

It is meant to fetch both the optimized experiment parameters from the DoE and the correct model file from the DT model registry and forward it to the virtual test bench. The communication needs to be in a near real-time basis due to the nature of the tests being run on both physical and virtual benches, so an optimization algorithm can choose the best suiting timing of each test on each bench. This software is called LIMS and the communication technology it uses is called MQTT.

3.3 Requirements

The described problem requires the following:

- Maximum communication delay tolerance in the scope of seconds
- Logging communication and test results in a scalable database
- Secure deployment and communication between the database and LIMS
- Scalable MQTT broker deployment, as the expected total clients may increase according to partners, currently a total of 5 clients is expected.
- Portable deployment regardless of the chosen cloud provider
- QoS level 2 to ensure high reliability of receiving messages exactly as planned

4. INTEGRATION INTO LIMS ARCHITECTURE

The LIMS Azure infrastructure defines the interworking and deployment of LIMS, MQTT broker, Scheduling algorithm, and virtual test benches [3]. Please note that Figure 1 below is based on the first version of the same figure in [3] but with modified details along with implementation changes and the project development.

In this chapter a focus is put on the implementation and deployment of the virtual test benches of the co-simulation environment in the Azure cloud infrastructure.

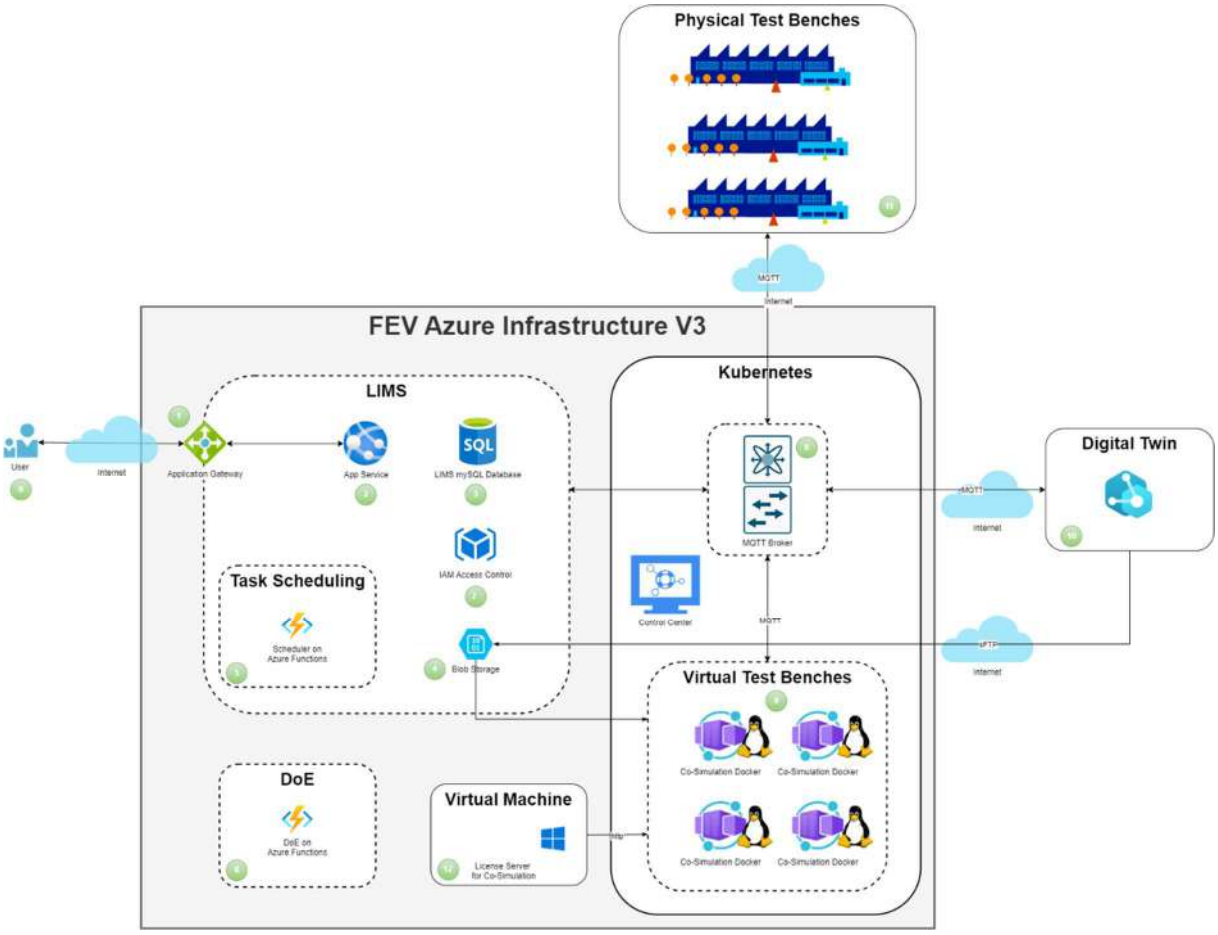

Figure 1 LIMS Azure Infrastructure for connecting virtual and physical test benches, DoE and Digital Twin

Cloud infrastructure provides the user with an interface to the co-simulation environment. It hosts and coordinates core components such as the scheduling algorithm, DoE, and virtual test benches. It supplies runtime services, resource management, and integration interfaces to ensure reliable execution. During the implementation several architectural adaptations were introduced.

4.1 Azure Kubernetes Service

Kubernetes is an open-source system designed to automate the deployment, scaling, and management of containerized applications. It provides a consistent

framework for running applications across clusters of machines, ensuring reliability and scalability while abstracting the underlying infrastructure as much as possible.

The Azure cloud computing platform developed by Microsoft provides services for hosting, developing, and managing applications and data [4]. It offers infrastructure, platform, and software capabilities that can be used to run workloads on the cloud or in hybrid environments. Azure supports different programming languages, frameworks, and operating systems, making it adaptable to various scenarios such as web applications, databases, analytics, and machine learning. Its global network of data centers is designed to deliver scalability and reliability, while integrated tools help organizations monitor, secure, and optimize their resources.

Azure Kubernetes Service (AKS) is the managed implementation of Kubernetes offered within the Azure cloud platform [5] [6] [7]. It delivers the same orchestration capabilities as Kubernetes but removes the need for users to set up and maintain the control plane themselves. AKS integrates with Azure services to provide features such as automated upgrades, monitoring, and scaling, while supporting both Linux and Windows containers. By combining Kubernetes with the managed infrastructure of Azure, AKS enables organizations to adopt container-based architectures more easily, streamline operations, and focus on application development rather than cluster management.

4.2 FMU Format

A Functional Mock-up Unit (FMU) is a standardized software package designed to enable the exchange and simulation of dynamic models across different tools. FMUs are built on the Functional Mock-up Interface (FMI) standard, which defines a common structure for packaging models so they can be shared and reused independently of the tool in which they were originally created [8]. An FMU typically contains the model equations, parameters, and metadata, and may also include compiled binaries that allow the model to run directly. Depending on the configuration, FMUs can be used in two modes: model exchange, where the equations are provided for another solver to handle, or co-simulation, where the FMU includes its own solver and runs alongside other components [9] [10] [11].

FMUs are used because they make complex system modelling more flexible and collaborative. By providing a universal format, they allow engineers and researchers to integrate models from different software environments without compatibility issues or the need of dedicated licenses. This interoperability is valuable for battery simulation environments, where systems are composed of several subsystems and are developed with various tools. FMUs also promote modularity, enabling teams to break down large systems into manageable components that can be tested and reused across projects. Their portability reduces development time, supports rapid prototyping, and facilitates system-level testing.

Since the FMI standard defines a common way to package and exchange models, FMUs can be platform independent. They can include compiled code for multiple operating systems (e.g., Windows, Linux, macOS), so the same FMU can run on

different platforms. Since the simulator in FASTEST will run in a Linux environment, the project will focus only on this platform. All FMUs that are provided by project partners will include binaries for this platform.

The folder structure inside an FMU consists of:

- `modelDescription.xml`
  Describes the model in a tool-independent way: variables, parameters, inputs/outputs, units, and capabilities. Every FMU must contain this file. It comprises:
  - Header information
  - Model variables
  - Model structure
  - Capability flags
- `binaries/`
  Contain compiled binaries (shared libraries such as .dll, .so, .dylib) for different operating systems and architectures (e.g., win64, linux64). These binaries implement the model equations or the solver logic.
- `sources/` (optional)
  May include C source code for the model, allowing recompilation if needed.
- `resources/` (optional)
  Hold auxiliary files such as tables, parameter sets, or other data needed by the model.
- `documentation/` (optional)
  Contains human-readable documentation (PDF, HTML, text) explaining the model, its usage, and assumptions.
- `model.png` (optional)
  A graphical representation of the model, often used for visualization in simulation tools.

In the project, it was decided for FMI 3.0 to be used, because it supports structured and array-based data exchange. This makes also MATLAB [12] matrix data types available, since they are broken down to arrays.

4.3 Communication

The co-simulation platform is composed of three FMUs, which are interconnected through signals.

- *FMU_Input_Handler* – Input Generation
- *FMU_Battery_Model* – Battery ageing and performance modelling (WP3)
- *FMU_Output_Handler* – Output collection and MQTT-Client

Figure 2 illustrates the interconnection of the FMUs by a set of signals given in **¡Error! No se encuentra el origen de la referencia.**. An actual co-simulation interconnection may consist of a subset of the signals given. The actual interconnection between the FMUs is defined in an inter-connection description file in JSON format. The current data structure is aligned with the WP3 in the GA meeting in M33 online. Possible changes might include variable name changes or modifications of data types.

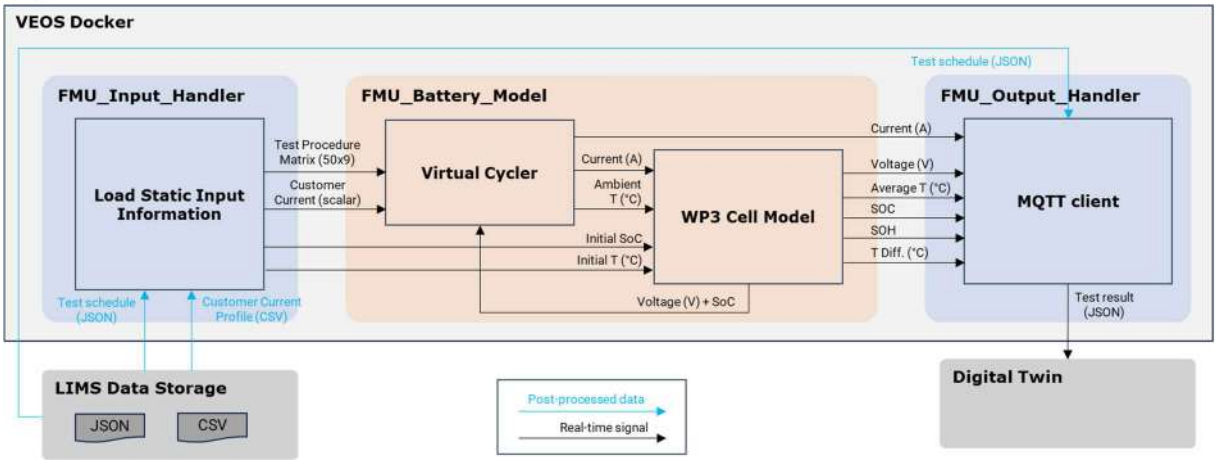

Figure 2 FMU integration concept into Co-Simulation platform

FMU_Input_Handler generates and delivers stimulus input signals for the simulation. The signals will be read out from a text file in CSV or JSON format. *FMU_Battery_Model* contains the implementation of the actual simulation models.

Within FMU_Battery_Model, two main blocks are implemented: the WP3 Model (virtual battery) and a virtual cycler. The virtual cycler manages the charge and discharge patterns of the virtual battery, with two main goals:

- Protect the battery, preventing it from going out of voltage (or SoC) range when applying a certain precomputed current profile and temperature setpoint.
- Implement test procedures whose stages and transitions depend on battery evolution on simulation time and cannot be known *a priori*.

This Virtual cycler consists of a test-plan interpreter, and a current regulation loop. It receives a test procedure matrix and / or a precomputed current profile (customer current) and uses the battery voltage and SoC as feedback to provide the actual current and ambient temperature setpoints for the battery model.

FMU_Output_Handler collects the signals and generates a text message in JSON format. Within this FMU, an MQTT client is implemented to publish the signal text messages, ensuring their availability to the LIMS Data Management Infrastructure.

Figure 3 displays a sample MQTT client message in JSON format. Each published signal is listed with its name, unit, type, value and a timestamp list. The structure follows a predictable schema to facilitate automated parsing and downstream processing.

The current values are not representative of real measurements. They originate from non-functional FMUs that emit placeholder data for connectivity testing. These FMUs are intended only to verify message formats and end to end delivery (see section 6.1 below).

```
Received message: {
    "test_name": "Overcharge",
    "test_type": "Cell-level",
    "test_UUID": "c61da39a-ace5-4f92-ad76-9a64516dc84d",
    "test_UUT": "Cell-01",
    "test_bench": "Virtual",
    "use_case": "Automotive",
    "variables": [
        {
            "name": "Current_Profile", "type": "Input", "unit": "A",
            "values": [ { "timestamp" : 0.180, "value" : 0.169 } ]
        },
        {
            "name": "Cell_Average_Temperature_01", "type": "Input", "unit": "degC",
            "values": [ { "timestamp" : 0.180, "value" : 0.159 } ]
        },
        {
            "name": "Cell_Temperature_Gradient_01", "type": "Input", "unit": "degC",
            "values": [ { "timestamp" : 0.180, "value" : 0.319 } ]
        },
        {
            "name": "Cell_Voltage_01", "type": "Input", "unit": "V",
            "values": [ { "timestamp" : 0.180, "value" : 0.319 } ]
        },
        {
            "name": "SoC", "type": "Input", "unit": "%",
            "values": [ { "timestamp" : 0.180, "value" : 0.319 } ]
        },
        {
            "name": "SoH", "type": "Input", "unit": "%",
            "values": [ { "timestamp" : 0.180, "value" : 0.159 } ]
        },
        {
            "name": "Internal_Resistance", "type": "Input", "unit": "mOhm",
            "values": [ { "timestamp" : 0.180, "value" : 0.159 } ]
        },
        {
            "name": "Capacity", "type": "Input", "unit": "Ah",
            "values": [ { "timestamp" : 0.180, "value" : 0.159 } ]
        },
        {
            "name": "Cell_Temperature_02", "type": "Input", "unit": "degC",
            "values": [ { "timestamp" : 0.180, "value" : 0.159 } ]
        },
        {
            "name": "Cell_Voltage_02", "type": "Input", "unit": "V",
            "values": [ { "timestamp" : 0.180, "value" : 0.757 } ]
        }
    ]
} on topic LIMS/metrics/veos/simulation_metrics
```


Figure 3 Sample MQTT-Client message containing simulation signals in JSON format

5. IMPLEMENTATION

An initial demonstration framework of an Azure-based co-simulation that treats virtual and physical benches as unified test resources was prepared , using cloud-native orchestration, storage, and event-driven triggers. Next steps will put emphasizes on automated workflows triggered by Blob Storage uploads and focus on improving reliability, flexibility, and strengthened monitoring by structured logging for failed steps and operational workflows.

5.1 FMUs for Input Generation and Output Collection

FMU_Input_Handler and *FMU_Output_Handler* denote the outer framework of the actual simulation engine, which is implemented in *FMU_Battery_Model*. While *FMU_Input_Handler* works as the data source to feed the simulation, *FMU_Output_Handler* collects the simulation output and forwards it to the MQTT broker for further processing.

5.2 Non-Functional FMUs

Non-functional FMUs provide a lightweight means of validating signal exchange and system integration without requiring the full functional logic of each component. This approach reduces complexity during early test phases, ensures that communication pathways are correctly established, and allows incremental development of functional FMUs to proceed in parallel.

5.3 Co-simulation Runtime Environment

In order to run the co-simulation process, the project decided to use VEOS version 2024-B. VEOS is a PC-based simulation platform developed by dSPACE for virtual validation of electronic control unit (ECU) software and system-level models [13]. Linux ports are available. It enables integration and system tests without dedicated hardware by executing virtual ECUs, models, and network communication components on standard PC infrastructure, thereby supporting early verification and validation activities in the development lifecycle.

The platform supports heterogeneous model execution and common automotive standards and interfaces. It provides virtual bus simulation for various automotive field bus protocols. VEOS integrates an embedded toolchain for message and topic handling, offers mechanisms for co-simulation and automated workflows, and can be coupled with dSPACE HiL systems to enable SiL-HiL transition and reuse of simulation artifacts across test stages.

VEOS is commonly used for software-in-the-loop (SiL) integration, regression testing, and virtual integration of ECUs and networks, allowing teams to validate communication pathways, exercise diagnostic and calibration interfaces, and run automated test sequences prior to hardware availability. Its open and extensible architecture facilitates integration with established test tools and data management infrastructures, supporting scalable test environments and

continuous validation practices in automotive and embedded-systems development.

The VEOS command-line interface provides a headless alternative to the graphical VEOS Player, allowing users to build, configure, and control offline simulation applications without launching the GUI. The CLI exposes operations for creating simulation targets, configuring communication and logging, collecting diagnostic information, and orchestrating automated test runs. It is intended for integration into scripted toolchains and continuous validation pipelines.

The '`veos build`' command is used to assemble a deployable simulation application from configured artifacts, producing an offline simulation package that can be executed independently of the VEOS Player. The resulting products from such a build process are Offline Simulation Application (OSA) files. OSA files are the packaged representation of a simulation system. They can be executed even from the command line without the GUI. They encapsulate the configured virtual ECUs, models, virtual bus and communication settings, logging configuration, and other runtime metadata. They are intended for reproducible, headless execution in automated or batch environments, enabling consistent simulation runs across different machines and integration pipelines.

The '`veos sim`' command launches and controls the execution of such offline simulations, providing options for runtime configuration, logging, and lifecycle control so that simulations can be executed reproducibly in batch or automated environments. These commands together enable separation of simulation assembly and execution, supporting automated build/test cycles and integration with external toolchains.

5.4 Kubernetes Platform

The Kubernetes platform supports automated deployment and scaling of containerized applications. In this sub-chapter, these elements arebriefly described and a summary of how they are used to implement virtual test benches in the co-simulation environment is also included [5].

The container runtime instantiates containers from images by pulling the image layers, unpacking them, and launching the configured process of the image as a container [14]. Images are possible artifacts and they are stored in registries. The image supplies the filesystem and metadata [15]. The runtime supplies the execution environment.

Kubernetes schedules pods, while the kubelet on a node requests the container runtime to create the containers declared in a pod specification. In other words, the pod is the Kubernetes construct that groups and configures containers, while the runtime performs the actual instantiation and process management. The kubelet is therefore the primary node agent that runs on each Kubernetes node. It ensures the containers described in the pod specification are created, running, and healthy. It registers the node with the API server, reports status and resource usage, executes health probes, and reconciles the desired pod state with the actual state observed on the host. The kubelet interacts with the local container runtime

to pull images, start and stop containers, and manage lifecycle hooks and volume mounts.

A pod is the smallest deployable unit in Kubernetes, representing one or more co-scheduled containers that share the same network namespace, IP address, and mounted storage volumes [16]. It models an application-specific logical host and provides a unit of deployment, scaling, and lifecycle management. Containers within a pod are tightly coupled and can communicate via localhost, share resource limits and probes, and include initialization containers for startup tasks or ephemeral containers for debugging. Pods are ephemeral by design. Higher-level controllers manage their desired state, scaling, and replacement to provide resilience and declarative orchestration.

In the co-simulation environment pods are scheduled, referred as VEOS pod, that shall execute the actual simulations. It comprises containers which are instantiated by an image that is specified by:

- Linux kernel based on Ubuntu 22.04,
- VEOS installation, version 2024-B,
- auxiliary scripts

A sidecar is a companion process or container that runs alongside a primary service to provide cross-cutting functionality [17]. It isolates concerns such as logging, proxying, configuration, and security so the main service remains focused on business logic. In Kubernetes, the sidecar is typically deployed in the same pod as the application. This co-location enables local communication, shared storage or network namespaces, and independent lifecycle management.

The sidecar gains access to the VEOS container by sharing the same Linux process namespace, which allows it to see the VEOS process identifiers (PID) directly through the kernel. Using elevated privileges, the sidecar leverages the Linux tool `nsenter` to attach to the VEOS process and enter its mount, PID, network, IPC, and UTS namespaces. Once inside these namespaces, any command executed by the sidecar runs as if it were executed from within the VEOS container itself. This approach bypasses application-level APIs and, instead, it uses kernel-level isolation mechanics to control VEOS directly. As a result, the sidecar can execute simulations and system commands dynamically without modifying, restarting, or exposing VEOS over the network.

A Blob Storage is an object based, scalable service for persisting large volumes of unstructured data [18]. Within Kubernetes deployments it is typically provisioned to containers via a Container Storage Interface (CSI) driver or accessed through gateway layers that presents the storage through a file-system interface. Exposing Blob Storage containers as persistent volumes enables pods to persist and share datasets across restarts and nodes while retaining centralized management of access control and backups.

In the co-simulation environment context, Blob Storage is used as a mounting point accessible to the VEOS pod for:

- FMUs containing models for simulation,
- meta-information (e.g. inter-connection specification),

- Simulation input runtime variable values (stimuli),
- Simulation results.

5.5 Simulation Workflow

Since the virtual test benches of the co-simulation consist of several interconnected FMUs, the build workflow inside VEOS is as follows:

1. Build all FMU models in VEOS and export as individual OSA files.
2. Open one OSA file and import the remaining FMU models from the corresponding OSA files into that initial OSA file.
3. In that OSA file, interconnect the FMUs and keep the resulting OSA file. This is the combined model for the co-simulation.

A script has been implemented to automate these steps. It encapsulates invocation procedures to standardize inputs and outputs and produce consistent deployment artifacts and runtime settings for integration into the virtual environment.

The following steps describe how a simulation is deployed into the Kubernetes cluster and is then executed by the VEOS simulator. They are illustrated in Figure 4 below.

1. Digital Twin:
    a. DT sends the FMUs and the inter-connection JSON file `connections.json` to Blob Storage in a predefined folder and with predefined file name.
2. Design of Experiment:
    a. DoE sends the input signals for simulation in a CSV file `input.csv` to LIMS, in order to be combined with the simulation models. The CSV contains signal values and timestamps.
3. LIMS:
    a. LIMS put all simulation relevant files of the model and simulation data in one folder within the Blob Storage
    b. Builds the FMUs and combine them into a single OSA file. Interconnects FMUs as defined in the inter.connection and JSON file according to the build workflow described above. This is performed in an Azure Function using the Python script `BuildFmusAndConnect.py` described in Appendix A.
    c. Makes sure the target pod is empty, eventually clear all the data in `/data/current_simulation_temp/`
    d. Transfers the OSA containing the inter-connected FMUs and the CSV file containing the simulation date from Blob Storage to the Kubernetes pod (Azure Function) `/data/current_simulation_temp/`
    e. Makes sure that the path of the newly uploaded OSA file is correct.
    f. Triggers the VEOS pod in order to load and start the simulation

4. VEOS pod:
    a. Loads the simulation by calling `veos sim load model.osa`
    b. Starts the simulation by calling `veos sim start`

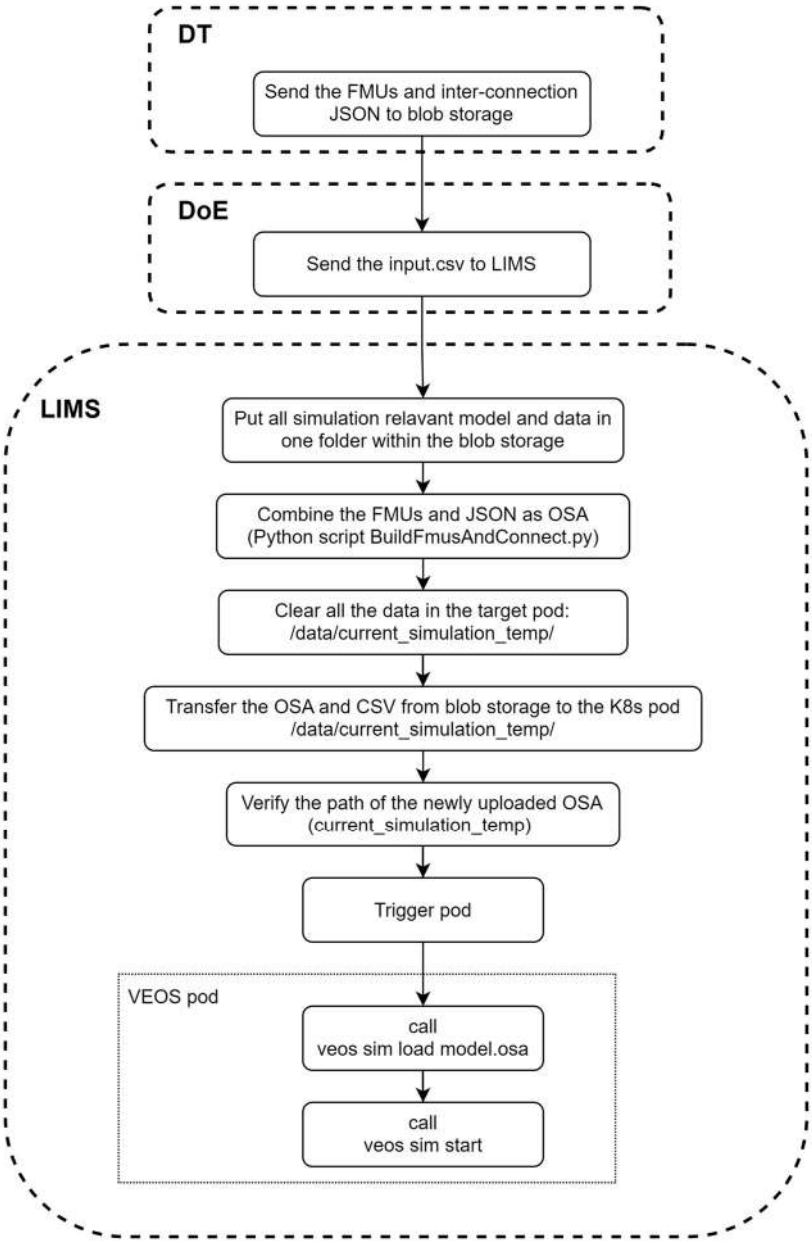

Figure 4 Simulation deployment and execution – sequence chart of workflow

It should be noted that steps 3.a., 3.c., 3.d., and 3.e. will have to be automized in future enhancements within the timeframe of this project. In the current implementation, these steps require manual interaction to proceed. The final goal is for these steps to run fully automated in the cloud environment.

Figure 5 illustrates how multiple simulations are referenced from different VEOS pods through a common link that resolves to distinct simulation datasets. In this arrangement, each VEOS pod references a link to a pod-specific dataset location, providing uniform access semantics while preserving dataset separation and isolation.

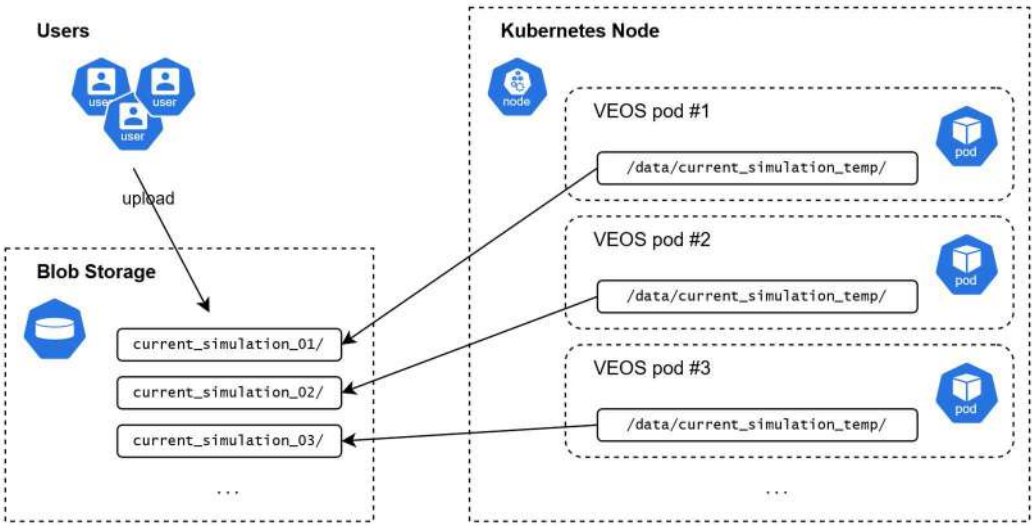

Figure 5 Simulation deployment and execution – Kubernetes block diagram

Users upload their simulation files to predefined, dedicated locations within the Blob Storage, a practice reinforced by namespace partitioning and access control rules to prevent conflicting uploads and accidental overwrites. Consistent metadata conventions and naming schemes further support traceability and reproducibility, and the storage layout enables concurrent execution, integrity verification, and automated retrieval by the VEOS runtime pod.

The integration of the VEOS runtime pod into the project's Kubernetes based deployment workflow will be addressed in a later project phase. This future work will cover the containerized execution of VEOS, its orchestration within Kubernetes, and the validation of the complete simulation deployment pipeline.

6. RESULTS

The project adopted the FMU format as the canonical exchange format for the virtual simulation runtime. A formal workflow for integrating FMUs into a co-simulation environment was developed.

According to this workflow a sample co-simulation was implemented in a Kubernetes container. The sample included partially non-functional FMUs that produced placeholder outputs for connectivity testing. The FMUs will later be replaced by FMUs containing the functional models.

6.1 Test Signal Generation and Interconnectivity Verification

In order to test and verify interconnectivity, FMUs *FMU_Input_Handler* and *FMU_Battery_Model* in MATLAB were implemented as non-functional FMUs. These FMUs are used exclusively for interconnectivity testing by exchanging signals without implementing the underlying component logic.

While non-functional *FMU_Input_Handler* is generating arbitrary test signals by copying the clock signal, non-functional *FMU_Battery_Model* executes some trivial signal processing for emulation purposes. In summary, those non-functional FMUs just take the clock signal, pass it forward, and are doing some occasional signal additions beyond of any physical context (see Figure 6, Figure 7, and **¡Error! No se encuentra el origen de la referencia.**).

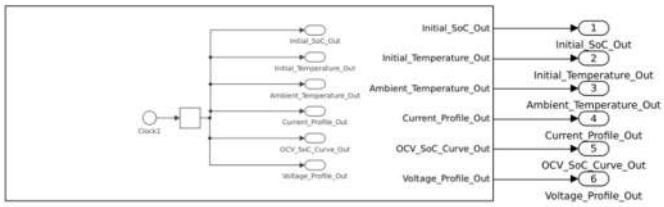

Figure 6 Non-functional FMU sample implementation for FMU_Input_Handler

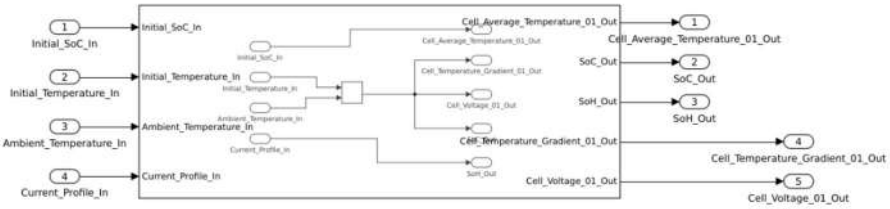

Figure 7 Non-functional FMU sample implementation for FMU_Battery_Model

6.2 *FMU_Output_Handler* – Output Collection

This FMU has been implemented in MATLAB, where the MQTT client functionality is realized through a dedicated S-function. The design ensures that the FMU can operate as a fully functional component within the simulation environment while,

at the same time, acts as a communication gateway. By embedding the MQTT client directly into the FMU, the system gains the ability to handle message formatting, topic assignment, and broker communication natively, without requiring external scripts or middleware (see Figure 8).

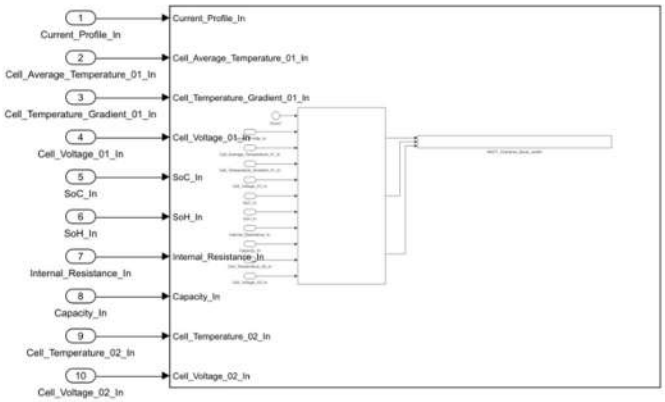

Figure 8 FMU_Output_Handler implementation with MQTT client

The primary purpose of this FMU is Output Collection. It gathers signals produced by other FMUs in the system and consolidates them into a structured output stream in JSON format. Its role is to ensure that all relevant signals are captured and prepared for transmission. Once collected, these signals are published as MQTT messages, making them accessible to external systems. This approach allows developers and integrators to validate that the simulation outputs are correctly propagated beyond the FMU aggregation, ensuring that communication pathways are reliable and consistent.

In *FMU_Output_Handler,* a signal value of `-1e9` is considered as invalid signal, for the signal that is not successfully connected to the port or no input is received from the previous model. In such a case the signal value is not published to the MQTT broker, i.e. it will not be listed in the JSON text message being published.

Systems such as the LIMS Data Management Infrastructure subscribe to the published MQTT topics and directly consume the signal data. This enables seamless integration of simulation results into broader data management workflows, supporting monitoring, analysis, and decision-making processes outside the FMI context.

The non-functional FMUs (*FMU_Input_Handler* and *FMU_Battery_Model*) and *FMU_Output_Handler* are interconnected as depicted in Figure 2 above. The combination has been tested in the VEOS simulator as described in the simulation workflow (section 5.5).

Figure 9 shows a sample screenshot of a running simulation just being started with non-functional data and the log output of a MQTT subscriber.

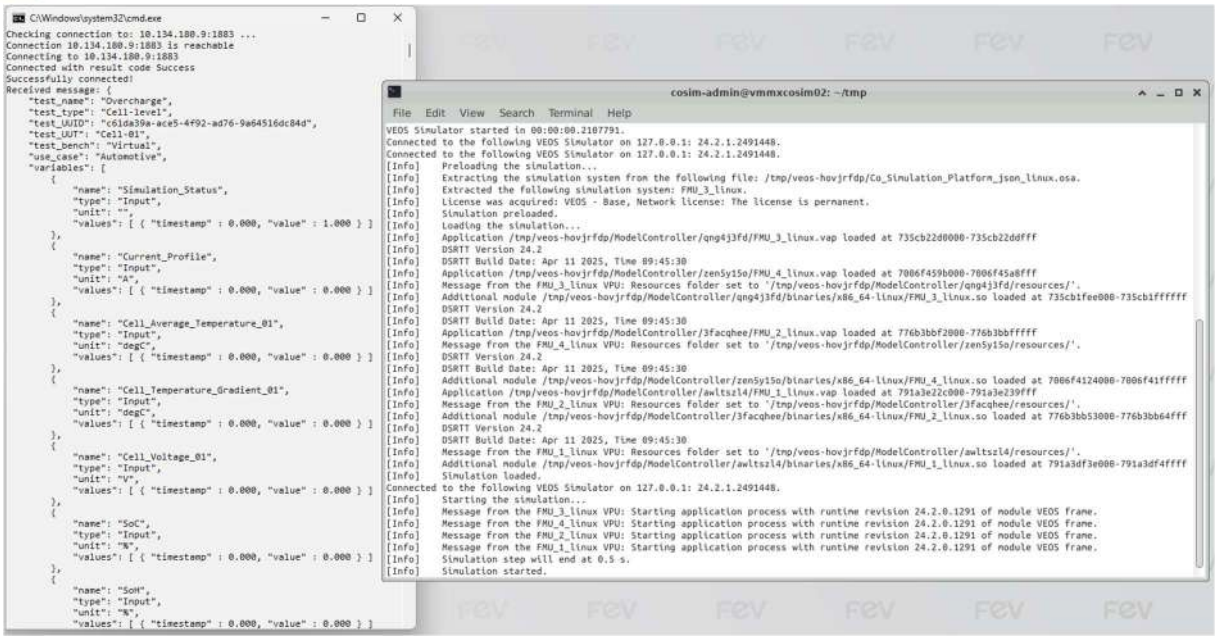

Figure 9 Screenshot of MQTT subscriber log (left) and simulator (right)

7. CONCLUSION & NEXT STEPS

The current results show how a sample implementation of the co-simulation environment could be implemented in the Azure infrastructure. In summary, the work demonstrates how virtual benches can be represented and operated within Azure using the same structural and operational semantics as physical benches. By leveraging Azure Kubernetes Service, Blob Storage, and event-driven communication, virtual benches can be integrated as test resources.

Furthermore, embedding use-case-specific Design of Experiment methodologies into the Azure-based orchestration layer would enable automated task distribution across virtual and physical resources. Decision logic in Azure can determine whether measurement campaigns should run virtually or physically based on model fidelity, resource availability, and test objectives.

Finally, extending the data model of the co-simulation environment and synchronizing it with Azure-hosted scheduling and execution components ensures consistent planning, configuration, and lifecycle management. This unified integration allows both virtual and physical benches to operate seamlessly within a scalable, cloud-supported testing infrastructure.

7.1 Next Steps

There are still remaining aspects that could need further clarification in optional future investigations, so the co-simulation environment becomes more reliable, flexible, and usable in daily operation. This includes refining interfaces between virtual and physical benches, improving virtual-bench fidelity, and validating automated task-distribution under realistic workloads. Strengthening monitoring of operational workflows will further support dependable and adaptable testing at scale.

Some FMUs will be implemented in Python. There is an ongoing investigation, under which preconditions Python based FMUs can be integrated into the virtual test benches of the co-simulation environment. Limitations are given by the VEOS simulator as well as by the tools that generate the FMU out of Python code. To reduce interoperability risks, it is recommended for these FMUs to be self-contained in terms of the libraries a Python application depends on. A systematic analysis and a platform limitation guideline are to be prepared to address this topic and define a workflow to be followed.

A complementary enhancement is to trigger the co-simulation workflow automatically by monitoring files as they are uploaded to Azure Blob Storage. By watching specific containers or directory paths for new or updated artifacts, such as FMUs, configuration files, or test definitions, the system can initiate the appropriate processing steps without manual intervention. This event-driven approach shortens turnaround times, reduces operational overhead, and ensures that uploaded assets immediately enter the testing pipeline in a controlled and reproducible manner.

Potential future implementations should place emphasis on a clear, responsive user experience. This is characterized by immediate, actionable feedback on validation failures, contextual help, and streamlined steps to accelerate onboarding.

A robust co-simulation workflow depends not only on successful automation, but also on clear diagnostics when something goes wrong. To support this, future environment implementations optionally should generate meaningful, structured logfiles whenever an automated step fails, such as FMU generation, simulation loading, or model initialization. These logs could capture the logs from the VEOS tools and relevant context, error sources, and system state so that issues could be traced. As a potential next step this procedure should be specified in more detail.

During the automated build process and simulation start up various errors might occur due to:

- erroneous FMUs that are incompatible with the VEOS simulation engine and cannot be built or loaded,
- incomplete or malformed FMUs,
- false or mismatching interconnection description files,
- contradictive simulation parameters.

If an error would occur during the workflow, in future implementations it could turn out to be important to provide the users with troubleshooting information with the aim of making debugging and troubleshooting easier. Therefore, all log files that are available and contain helpful information might need to be uploaded to the result area of Blob Storage.

8. BIBLIOGRAPHY

[1] A. P. Passaro, "FASTEST Deliverable D5.3: Integration plan for Digital Twin on the platform," 31 03 2025. [Online]. Available: https://fastestproject.eu/wp-content/uploads/2025/07/D5.3-Integration-plan-for-Digital-Twin-on-the_compressed.pdf. [Accessed 11 02 2026].

[2] B. Rodrigues, "FASTEST Deliverable D2.2: Definition of battery system testing for automotive, off-road and stationary use cases," 29 08 2025. [Online]. Available: https://fastestproject.eu/wp-content/uploads/2025/09/D2.2-Definition-of-battery-system-testing-for-automotive-off-road-and-stationary-use-cases.pdf. [Accessed 11 02 2026].

[3] S. Liu, "FASTEST Deliverable D6.5: Real-time connection of physical and virtual bench," 19 11 2024. [Online]. Available: https://fastestproject.eu/wp-content/uploads/2024/12/D6.5-Real-time-connection-of-physical-and-virtual-bench.pdf. [Accessed 8 12 2025].

[4] Microsoft, "Get to know Azure," 8 12 2025. [Online]. Available: https://azure.microsoft.com/en-us/explore/.

[5] Kubernetes, "Overview," 3 12 2025. [Online]. Available: https://kubernetes.io/docs/concepts/overview/. [Accessed 8 12 2025].

[6] Microsoft, "What is Azure Kubernetes Service (AKS)," 8 12 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/aks/what-is-aks.

[7] Microsoft, "Azure Kubernetes Service (AKS)," 8 12 2025. [Online]. Available: https://azure.microsoft.com/en-us/products/kubernetes-service/.

[8] Modelica Association, "Functional Mock-up Interface Specification, Version 3.0," 10 5 2022. [Online]. Available: https://fmi-standard.org/docs/3.0/. [Accessed 8 12 2025].

[9] C. Bertsch, M. Blesken, T. Blochwitz, A. Junghanns, P. R. Mai, B. Menne, K. Reim, M. Süvern, K. Schuch, T. Sommer and P. Täuber, "Beyond FMI - Towards New Applications with Layered Standards," 9-11 10 2023. [Online]. Available: https://www.dspace.com/shared/data/pdf/2024/dSPACE_FMI-New-Applications_Paper_2023.pdf. [Accessed 8 12 2025].

[10] C. Gomes, M. Najafi, T. Sommer, M. Blesken, I. Zacharias, O. Kotte, P. R. Mai, K. Schuch, K. Wernersson, C. Bertsch, T. Blochwitz and A. Junghanns, "The FMI 3.0 Standard Interface for Clocked and Scheduled Simulations," 20-24 9 2021. [Online]. Available: https://www.dspace.com/shared/data/pdf/2021/FMI-3_ClockedAndScheduledSimulations_Modelica2021.pdf. [Accessed 8 12 2025].

[11] A. Junghanns, T. Blochwitz, C. Bertsch, T. Sommer, K. Wernersson, A. Pillekeit, I. Zacharias, M. Blesken, P. R. Mai, K. Schuch, C. Schulze, C. Gomes and M. Najafi, "The Functional Mock-up Interface 3.0 - New Features Enabling New Applications," 20-24 9 2021. [Online]. Available: https://www.dspace.com/shared/data/pdf/2021/FMI-3_NewFeaturesEnablingNewApplications_Modelica2021.pdf. [Accessed 8 12 2025].

[12] The MathWorks Inc., "MATLAB version: 24.2.0 (R2024b)," 2024. [Online]. Available: https://www.mathworks.com. [Accessed 8 12 2025].

[13] dSPACE, "VEOS," 9 7 2025. [Online]. Available: https://www.dspace.com/en/inc/home/products/sw/simulation_software/veos.cfm. [Accessed 8 12 2025].

[14] Kubernetes, "Containers," 25 10 2024. [Online]. Available: https://kubernetes.io/docs/concepts/containers/. [Accessed 8 12 2025].

[15] Kubernetes, "Images," 20 5 2025. [Online]. Available: https://kubernetes.io/docs/concepts/containers/images/. [Accessed 8 12 2025].

[16] Kubernetes, "Pods," 28 10 2025. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/. [Accessed 8 12 2025].

[17] Kubernetes, "Sidecar Containers," 21 5 2025. [Online]. Available: https://kubernetes.io/docs/concepts/workloads/pods/sidecar-containers/. [Accessed 8 12 2025].

[18] Microsoft, "Use Azure Blob storage Container Storage Interface (CSI) driver," 1 8 2024. [Online]. Available: https://learn.microsoft.com/en-us/azure/aks/azure-blob-csi. [Accessed 8 12 2025].

APPENDIX A. Script Support for Building and Interconnecting FMUs

The Python script `BuildFmuAndConnect.py` is derived from VEOS demo scripts. It builds and combines all FMU given in a sub-folder and creates a single OSA file with all the FMU interconnected according to a specification. The subfolder must contain a JSON file `connections.json` which comprises the interconnection specification of the FMUs.

E.g., a sub-folder `CoSimulation` contains the entries:

```
connections.json
FMU_Input_Handler.fmu
FMU_Battery_Model.fmu
FMU_Output_Handler.fmu
```

The file `connections.json` contains interconnections in the following format (extract):

```
[
    {
        "InSignalReference" : "/FMU_Battery_Model/Initial_SoC_In/Initial_SoC_In",
        "OutSignalReference" : "/FMU_Input_Handler/Initial_SoC_Out/Initial_SoC_Out"
    },
    {
        "InSignalReference" : "/FMU_Battery_Model/Initial_Temperature_In/Initial_Temperature_In",
        "OutSignalReference" : "/FMU_Input_Handler/Initial_Temperature_Out/Initial_Temperature_Out"
    },
    {
        "InSignalReference" : "/FMU_Output_Handler/Current_In/Current_In",
        "OutSignalReference" : "/FMU_Battery_Model/Current_Out/Current_Out"
    },
    {
        "InSignalReference" : "/FMU_Output_Handler/Cell_Voltage_In/Cell_Voltage_In",
        "OutSignalReference" : "/FMU_Battery_Model/Cell_Voltage_Out/Cell_Voltage_Out"
    },
    {
        "InSignalReference" : "/FMU_Output_Handler/SoC_In/SoC_In",
        "OutSignalReference" : "/FMU_Battery_Model/SoC_Out/SoC_Out"
    },
    {
        "InSignalReference" : "/FMU_Output_Handler/SoH_In/SoH_In",
        "OutSignalReference" : "/FMU_Battery_Model/SoH_Out/SoH_Out"
    },
    ...
```

Calling the build script

```
python3 BuildFmuAndConnect.py CoSimulation
```

will result in an OSA output file

```
CoSimulation.osa
```

This file contains the simulation application which consists of the FMUs listed above being interconnected. It can be loaded and started in VEOS.