Ref. Ares(2026)4248909 - 24/04/2026

EUROPEAN COMMISSION

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

FASTEST

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

Deliverable D2.3: Concepts for smart combination of physical and virtual testing

Primary Author [Felix Pischinger]

Organization [FEV]

Date: [23.04.2026]

Doc. Version: [V1.0]

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

restricted

Document Control Information	
Settings	Value
Work package:	WP2
Deliverable:	Concepts for smart combination of physical and virtual testing
Deliverable Type:	R – Document, report
Dissemination Level:	PU - Public
Due Date:	31.02.2026
Actual Submission Date:	23.04.2026
Pages:	< 31 >
Doc. Version:	V1.0
GA Number:	101103755
Project Coordinator:	Bruno Rodrigues │ ABEE (bruno.rodrigues@abeegroup.com)

Formal **Reviewers**		
Name	Organization	Date
Doniyor Urishov	**VTT**	13.04.2026
Koen Vanden Boer	**Flanders Make**	16.04.2026

Document History			
Version	Date	Description	Author
0.1	15.01.2026	First Draft	Felix Pischinger (FEV)
0.2	02.04.2026	Second Draft, Internal Review	Philipp Brendel (FHG) Felix Pischinger (FEV)
1.0	23.04.2026	Release	Philipp Brendel (FHG) Felix Pischinger (FEV)

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 Digital Twin architecture managing the information flows, and the TRL 6 proven and integrated prototype of the hybrid testing platform.

restricted

LIST OF ABBREVIATIONS, ACRONYMS AND DEFINITIONS

Acronym	Name
POC	Proof Of Concept
LIMS	Laboratory Inventory Management System
MQTT	Message Queuing Telemetry Transport
VPC	Virtual Private Cloud
DT	Digital Twin
DoE	Design of Experiment
UUT	Unit Under Test
UUID	Unit under test ID
SFTP	Secure File Transfer Protocol
IoT	Internet of things
UI	User Interface
XiL	X in the loop
IP	Internet Protocol
FIM	Fisher-Information-Matrix
SOC	State-of-Charge

restricted

LIST OF FIGURES

Figure 1. Schematic interpretation and representation of a FIM computed from 9 parameters........ 11
Figure 2 Smart DoE workflow for combination of physical and virtual testing. .. 12
Figure 3 LIMS Overview........ 14
Figure 4 Code of reading trigger message........ 17
Figure 5 Code of checking FimCore package........ 18
Figure 6 Code of processing the input data to the Doe algorithm 18
Figure 7 Test Bench Capabilities data in Azure Blob Storage........ 20
Figure 8 Test Bench capabilities attributes 20
Figure 9 Test data from json message........ 21
Figure 10 Code of parsing input from json message 22
Figure 11 Code of searching for test bench 22
Figure 12 Code of updating the json message 23
Figure 13 Code of client object initialization to access blob storage 23
Figure 14 Code of accessing and downloading test bench capabilities 23
Figure 15 Output of the smart DoE algorithm for a 1-hour CC-discharge profile applied in context of an LFP-based cathode........ 24
Figure 16 Log of Azure Function showing results from Fim calculations........ 26
Figure 17 Json message input data........ 28
Figure 18 Log of Test bench capabilities found in blob storage........ 28
Figure 19 Log of Test bench match for the input data 28
Figure 20 Log of automated deployment - part 1 29
Figure 21 Log of automated deployment - part 2 29

restricted

Table of Contents

1. EXECUTIVE SUMMARY 7
2. OBJECTIVES 8
3. INTRODUCTION 9
3.1 Purpose 9
3.2 Requirements 9
4. Smart DoE for optimal combination of physical and virtual testing 11
5. DoE Algorithm Integration 13
5.1 LIMS Overview 13
5.2 Integration 15
5.2.1 Azure Function App 15
5.2.2 Implementations & Deployment 17
6. Physical Testbench Selector 19
6.1 Integration 19
6.1.1 Implementation 21
7. RESULTS 23
7.1 Smart DoE for reduction of testing time and costs 24
7.2 Test DoE Algorithm in Azure Function 25
7.2.1 Test Description 25
7.2.2 Test Results and Evaluation 25
7.3 Test Physical Test Bench Selector in Azure Function 27
7.3.1 Test Description 27
7.3.2 Test Results and Evaluation 27
7.4 Deployment Test 29
8. CONCLUSION & NEXT STEPS 29
9. BIBLIOGRAPHY 31

restricted

1. EXECUTIVE SUMMARY

Deliverable D2.3 presents the concepts and technical realisation for a smart combination of physical and virtual testing, forming a core building block of the FASTEST hybrid testing platform. The overarching objective is to enable a fast-track, information-driven testing workflow in which physical and virtual experiments are systematically coordinated, reducing test effort while improving parameter quality and decision-making for battery system development.

To achieve this, the deliverable integrates the methodological foundations introduced in D2.1 and D2.2 — such as model-based Design of Experiments (DoE), Fisher-Information-Matrix (FIM)-based information assessment, and test-scenario specifications — into a cloud-based technical environment built around the Laboratory Information Management System (LIMS).

This deliverable documents the implementation, integration and stabilization of these components inside an Azure-based architecture, realized through the "DoE Function", a modular Azure Function that handles data exchange, triggers model evaluations, executes the DoE algorithm, and selects physical test benches based on capability datasets stored in Azure Blob Storage. The implemented system enables automated communication between LIMS, DoE algorithms and both test-bench types, ensuring consistent data handling, traceability and seamless workflow execution.

The result serves towards a fully interconnected hybrid testing pipeline in FASTEST. It demonstrates how physical and virtual testing can be combined intelligently to reduce redundant physical experiments, accelerate parametrization processes, and support robust battery R&D. The next steps will focus on validating the entire pipeline under realistic boundary conditions (T6.6), refining the integration with Digital Twin services, and ensuring readiness for TRL-6 demonstration.

restricted

2. OBJECTIVES

The FASTEST project aims to 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, virtualised benches, and DT data are incorporated into the daily facility operation to reach a new level of efficiency.

During the project, FASTEST consortium aims to develop up to TRL6 for the platform and its components: the optimal DoE strategies according to three different use cases (automotive, stationary, and off-road); the development of a complete set of physics-based and data-driven models are able to substitute physical characterisation experiments; the overarching Digital Twin (DT) architecture managing the information flows, and the TRL6 proven and integrated prototype of the hybrid testing platform. The platform, aimed to become a flexible tool for any chemistry and application. One of which is, including the predictive maintenance algorithm, developed in WP3, which aims to estimate the remaining useful life (RUL) of a battery, taking into account various negative test scenarios. These scenarios are categorized into three main sections: mechanical, electrical, and thermal abuse. Specific conditions include battery casing penetration, internal and external short circuits, state-of-charge (SoC) calibration errors leading to rapid degradation from overcharge or over discharge, and thermal system failures resulting in internal and external heat exposure. By simulating these extreme operating conditions, the tests ensure that the algorithm can effectively handle rare battery failures. Based on the outcomes of these negative case tests, the system will determine whether a test should be conducted on the virtual test bench or the physical test bench, as part of WP2 and incorporation with LIMS.

The first objective is the investigation and development of intelligent algorithms for a smart combination of physical and virtual testing in line with the previously defined DoE specifications and methodologies (Deliverable D2.1) and use-case specific definition of battery system testing procedures for an automotive, off-road and stationary use case (Deliverable D2.2).

The second objective is the integration of the developed algorithms in the LIMS environment. This includes the evaluation of different integration approaches as well as the implementation of the selected approach.

restricted

3. INTRODUCTION

The following chapter focuses on laying the foundation of the project from a technical overview.

3.1 Purpose

The purpose of the automatic task distribution inside the DoE workflow is to decide whether a test demand should be performed on the physical test bench, effectively trading time and costs for new information about the unit under test, or rather be virtualized using the more efficient virtual models. In this way, the developed DoE algorithm aims to avoid the physical execution of experiments that have been performed before or that are likely to not produce new meaningful results regarding the unit under test (UUT).

The first part of this task is to integrate the DoE algorithm into the LIMS workflow. The implementation must operate seamlessly within the LIMS environment, ensuring accurate processing of user input data and correct delivery of output data. All data should be retrieved from a designated location and stored back in the same manner within the Data Storage System. The entire process must be continuously monitored and logged to maintain traceability and compliance.

Any required adaptors or custom code necessary to achieve this integration must be developed and implemented. Furthermore, an automated deployment procedure must be established to support ongoing modifications through a Continuous Integration and Continuous Deployment (CI/CD) pipeline.

The second part of this task is to enable the usage of the automated test bench selector algorithm within the LIMS workflow. A comprehensive list of test bench capabilities provided by partners must be available in the LIMS system. This list should be read and processed by the Azure Function. Based on the information in the list and the specified test parameters, the algorithm must select an appropriate test bench if one is available. The selected test bench must then be returned to LIMS. In cases where no suitable test bench is available, this outcome must also be communicated back to LIMS.

After the integration is completed successfully, the algorithms are run and tested and the results verified.

3.2 Requirements

To enable the implementation and validation of the smart DoE algorithm, the following requirements must be met:

restricted

- Availability of virtual models for the varying unit-under-tests covering different model hierarchies and cell types
- An initial "best-practice" model parametrization that serves as a baseline for both the development and the validation of the smart DoE algorithm combining physical and virtual testing
- Availability of tools and methods for assessing uniqueness of parameter identification via Fisher-Information-Matrices as identified in Deliverable D2.1.

To enable the integration of the procedure optimization algorithm and the automated testbench selector algorithm within the LIMS environment, several requirements must be met. The DoE Algorithm with the underlying FIM software logic must be available, the input parameters need to be defined, and an exchange platform on Azure Storage must be established and aligned. Additionally, a list of available test benches must be provided.

restricted

4. Smart DoE for optimal combination of physical and virtual testing

This chapter covers all developments related to the smart DoE workflow that aims to combine physical and virtual testing in an optimal way.

In Deliverable D2.1, a model-based Design of Experiments (DOE) framework was introduced to support fast and reliable parametrization of physics-based battery models as a cornerstone of the FASTEST hybrid testing platform. The central idea is to exploit Fisher-Information-Matrix (FIM)–based optimal design to maximize the information content of selected test profiles with respect to the model parameters, thereby reducing both experimental effort and parameter uncertainty.

While the reader is referred to D2.1 for a deeper discussion of the underlying concepts, we repeat the most important cornerstones for a better comprehensibility within this Deliverable. The FIM quantifies how sensitively a model output responds to changes in a given parameter vector θ under a specific excitation (e.g. current profile). For a model response $f(U,\theta)$ with Gaussian measurement noise, the FIM is defined as

$$\mathrm{FIM}(U,\theta) = \frac{\partial f(U,\theta)}{\partial \theta}^{\mathsf{T}} \frac{\partial f(U,\theta)}{\partial \theta}.$$

The diagonal entries of the FIM measure how well individual parameters are locally identifiable (higher values correspond to lower variance in the estimated parameter), while the off-diagonal entries characterize parameter correlations and potential identifiability issues. From the FIM, scalar "optimality" measures such as D-optimality (determinant), E-optimality (smallest eigenvalue) and T-optimality (trace) are derived to compare and optimize- experimental designs.

Figure 1. Schematic interpretation and representation of a FIM computed from 9 parameters.

Building on this definition, Deliverable D2.1 also sketched an initial workflow for optimal DoE but further work and discussions within the FASTEST consortium have led to an iterative refinement of that workflow that is visualized in Figure 2 and discussed in the following.

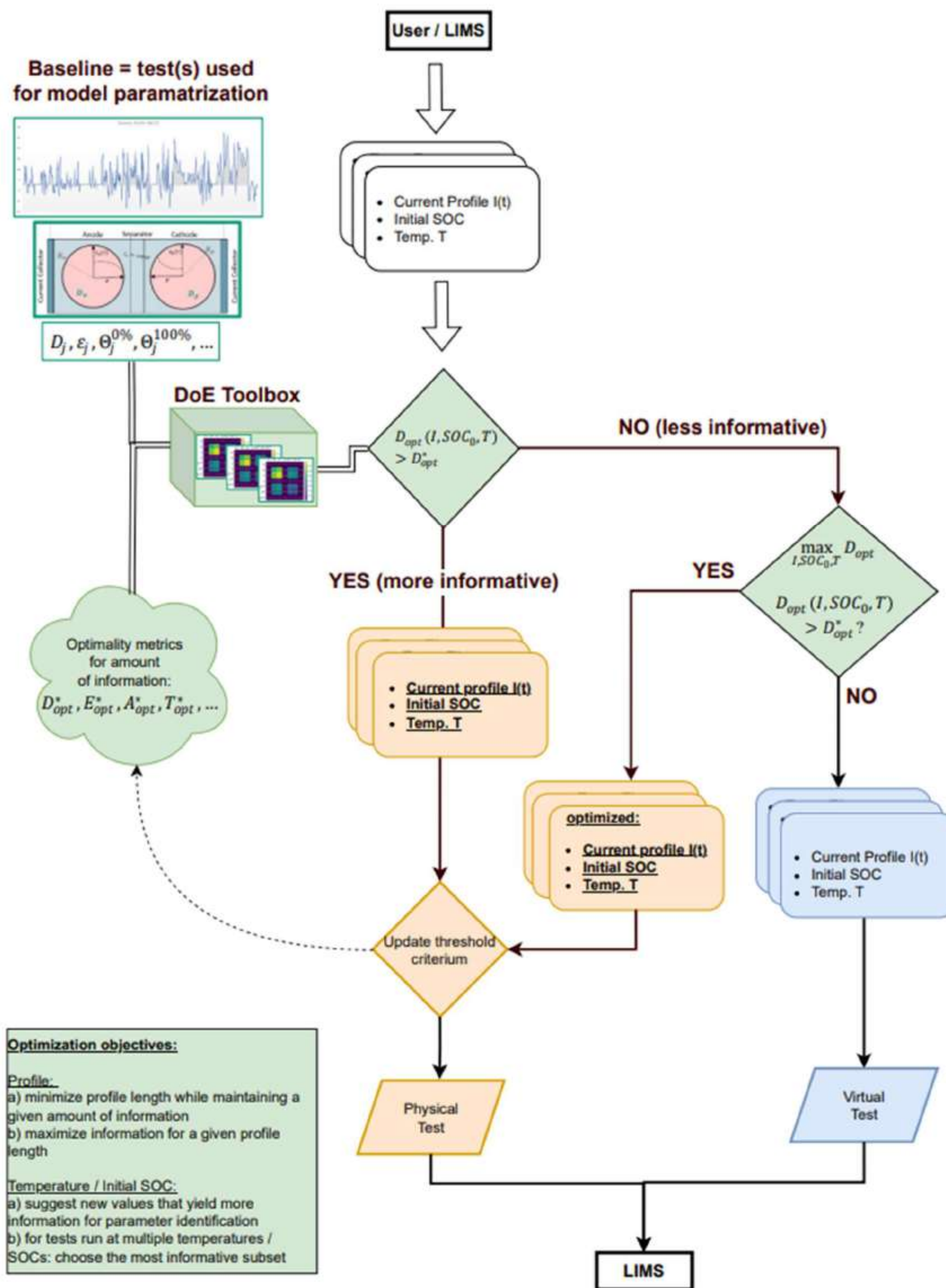

Figure 2 Smart DoE workflow for combination of physical and virtual testing.

At the core, the system connects three main elements: the parametrized model, a DoE toolbox driven by the concept of FIMs, and the user/LIMS interface for evaluating and optimizing a certain test demand. Some preliminary test procedure

restricted

– defined by a current profile I(t), an initial SOC, and a temperature T – is treated as a baseline that has been used to parametrize the respective surrogate model. Based on the DoE toolbox and the concept of FIMs, optimality metrics for the "amount of information" are computed as baseline threshold criteria. These metrics quantify how well a given combination of profile P, C-rate C, temperature T, and initial SOC supports parameter identification and can be chosen based on the specific scope and objectives of the test configuration being evaluated.

The DoE toolbox evaluates the candidate test configuration by calculating the optimality metric (e.g. $D_{\text{opt}}(I, SOC_0, T)$) and comparing it to a reference value (D^*_{opt}) that is determined from a single or multiple baseline profiles used for model parametrization. If the candidate configuration yields a higher optimality metric than the current reference, the test is to be executed physically in order to provide the expected additional information to the user. If a test yields less information than the reference threshold, however, its outcome likely won't improve the virtual model, and hence is deemed less useful in the scope of battery test virtualization. The smart DoE algorithm than attempts to optimize this given test design by changing its degrees of freedom (e.g., profile shape, C-rate, Temperature or initial SOC configuration) until its optimality metric suggests a higher value and it becomes useful for physical execution. The resulting optimized test definitions are then passed via LIMS to be executed as physical or virtual tests, closing the loop and continuously improving the test design with respect to information content and efficiency.

In order to successfully implement the algorithm in Figure 2, the model-based DoE toolbox had to be aligned with model-developments of WP3 of the FASTEST project. More specifically, this process required an intensive stretch of reviewing, debugging and improvement of the underlying software library that computes FIM based on WP3-related models. These models are exported and evaluated as Functional-Mock-up Units (FMU) for their final and seemless integration into the FASTEST platform. The following Section 5 gives a deeper insight into the work related to that integration.

5. DoE Algorithm Integration

This chapter covers all activities related to the integration related to the DoE Algorithm implementations, described in the previous chapter.

This chapter contains a short recap of how the LIMS environment is designed and then specifies how the DoE algorithm is integrated.

5.1 LIMS Overview

restricted

In the picture below, an overview of the LIMS workflow is shown. The integration of the algorithms of this delivery will be packaged in one Azure Function called "DoE function".

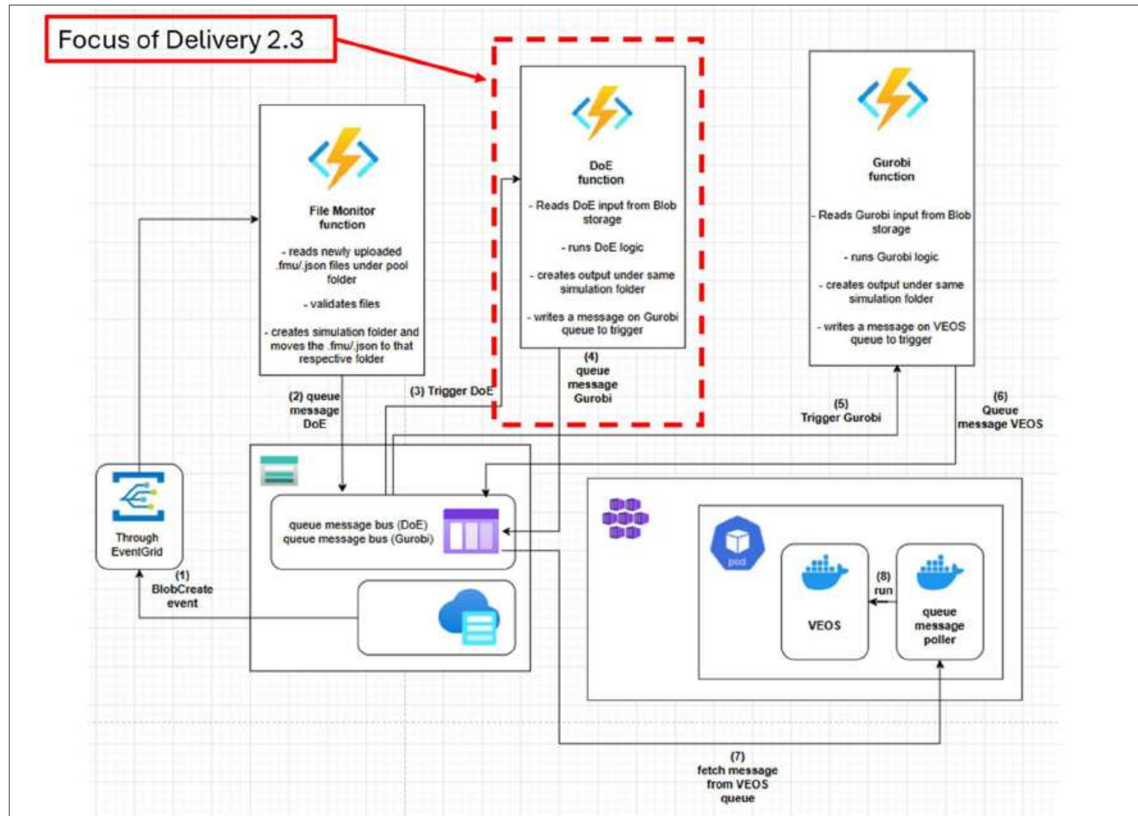

Figure 3 LIMS Overview

In this chapter and chapter 6, the detailed content of DoE function will be described.

In order to understand the background if LIMS, in this subchapter a short recap will be provided.

Starting with LIMS, we decided to use Azure App Service (Microsoft, App Service - Build and Host Web Pages, 2024) to host it as a web application that is published as a Linux docker container for portability purposes. The accessibility is controlled by an application gateway (Microsoft, Application Gateway, 2024) and optionally whitelisting the public IPs of users when needed. The application is connected to a database instance hosted on Azure SQL Database service (Microsoft, Azure SQL Database, 2024). This database instance allows only specific set of IPs for security purposes.

The virtual test benches are deployed in the form of a docker container running a co-simulation software. This docker container is hosted on Azure Kubernetes Pods. Just before the start of each virtual test, the respective model is fetched from our blob storage which in turn receives these models from our partner in at the DT side. As for the physical test benches, the same metrics will be pushed from it to

the respective clients through our broker, but the actual deployment of these test benches remains under the responsibility of the partner's industrial plant.

The physical test benches are provided by multiple partners. Depending on the test demands and configurations and the available test bench capabilities, LIMS will return information on a suitable test bench to roll out a physical test.

5.2 Integration

As stated in the overview, the integration of the algorithms will be done in an Azure Function called "DoE function". This DoE Function will include all software from this delivery; the DoE algorithm and the physical test bench selector, as well as the additional implementations required for the integration.

The specific rules and boundaries we have to consider using Azure functions make it necessary to give a more detailed description about how they work.

5.2.1 Azure Function App

Due to the expected short runtime cycles and the necessity of being triggered easily by LIMS through HTTP calls, Azure Function App is the chosen medium of deployment for DoE. We will be able to assign the appropriate development environment, variables and libraries inside the function according to the programming language used to develop DoE. Using Azure functions will help us connect the processes between LIMS and DoE with minimal deployment delays, real-time and secure communication within our virtual network.

In order to handle multiple functions within Azure, the Azure Function type "Durable Function" is used in this project.

5.2.1.1 Durable Functions

Durable Functions let you build stateful, long-running, and reliable workflows in serverless apps, all managed inside your Function App in the Azure Portal. ((Microsoft, Durable Functions, n.d.))

Key capabilities:

- Orchestrations (deterministic workflows): Define the control flow for calling other functions, waiting, retrying, and aggregating results—without managing state yourself.
- Long-running operations: Pause and resume across hours/days with durable timers, while paying only for actual execution time.

restricted

- Event-driven interactions: Wait for external events (e.g., HTTP callbacks, messages) to continue a workflow—great for human approval steps.
- Fan-out/fan-in: Run many activities in parallel and aggregate results efficiently.
- Sub-orchestrations & modularity: Compose complex workflows from smaller orchestrations.
- Durable Entities: Lightweight, stateful objects for fine-grained state and counters without full workflows.
- Reliability features: Built-in retries, exactly-once orchestration progress, and automatic state persistence via the Durable Task framework.
- Multi-language support: .NET/C#, JavaScript/TypeScript, Python, PowerShell.

In the Azure Portal you can:

- Create and configure the Function App (storage, scale, app settings).
- Deploy functions and view invocations and logs (via Application Insights).
- Monitor performance and query orchestration/instance status (through logs, durable APIs, and Azure Monitor).
- Manage lifecycle actions (e.g., start, terminate, purge instances—typically via APIs/CLI, surfaced through Portal-connected tools).

Benefits of using Activity triggers (i.e., Activity Functions; Activity functions are the worker steps called by orchestrations.)

- Separation of concerns: Keep orchestration logic deterministic and pure, while doing I/O or CPU-heavy work in activities.
- Reliability with retries: Orchestrators can call activities with automatic retry policies and compensation if needed.
- Scalability: Activities run independently and can scale out, enabling parallelization and high throughput.
- Fault isolation: Failures are contained to the activity; orchestrator can catch, retry, or take alternate paths.
- Testability & reuse: Activities are small, focused functions that are easy to unit test and reuse across workflows.
- Non-deterministic friendly: Activities can safely perform non-deterministic operations (e.g., HTTP calls, random numbers) that orchestrators must avoid.
- Observability: Each activity invocation is logged, making it easier to trace and measure steps of your process.

restricted

5.2.2 Implementations & Deployment

This chapter gives a detailed technical description of the additional steps to enable the integration of the DoE algorithm.

The model-based DoE algorithm is based on models developed in WP3 and a FIM-library developed in T2.1 and T2.2 that servers as backend for FIM calculations.

As described in the previous chapter, the DoE algorithm will be integrated in LIMS by using Azure Functions. Therefore, two major steps are required:

- Implement additional SW to handle input and output data with the LIMS file system -> Logical communication between LIMS and DoE Algorithm/Developing interface adapters
- Implement SW to enable the proper deployment of the FIM core and the required dependencies

5.2.2.1 Implementation of interface adapters

In order to interact with LIMS, a test-specific json exchange file is used to read and write the data (See Delivery 6.6 for more details on the json schema).

Therefore, adapters need to be implemented to handle the communication and pass the data.

Reading input data from trigger message and passing to the DoE algorithm:

```
message_body = azqueue.get_body().decode('utf-8')
logging.info('%s trigger processed a message: %s', QUEUE_NAME, message_body)

message_body_updated = run_fim(message_body)

logging.info("Updated queue message: %s", message_body_updated)
```

Figure 4 Code of reading trigger message

Checking availability of FIM library package in Azure Function:

```
def run_fim(message_body):
    'fim logic goes here'
    try:
        # Kritisches Paket erst hier importieren:
        from fim_core_src.fim_core import main_fim_core
        logging.info("FimCore imported successfully")
    except Exception as e:
        logging.exception("Error while importing fim_core_src")
        excepttion = traceback.format_exc()
        logging.exception(excepttion)
        # return func.HttpResponse(f"IMPORT-ERROR: {e}", status_code=500)
        return        Chat Ctrl+L  Edit Ctrl+I


    res = main_fim_core(message_body)
```

Figure 5 Code of checking FimCore package

Reading, processing and returning the new calculated parameters:

```
def run_process(self, test_config_json_path):
    inputData = json.loads(test_config_json_path)

    # Initial set of model parameters [m, k, b]
    initial_parameters = [inputData.get("m"), inputData.get("k"), inputData.get("b")]

    # Simulation settings (time and initial conditions)
    # user_data: (n, t0, t1, A0, phi)
    #   n   = number of time samples
    #   t0  = start time
    #   t1  = end time
    #   A0  = initial amplitude
    #   phi = initial phase

    n = inputData.get("n")
    t0 = inputData.get("t0")
    t1 = inputData.get("t1")
    A0 = inputData.get("A0")
    phi = inputData.get("phi")
    user_data = (n, t0, t1, A0, phi)


    # Configure FIM options
    n_slices= inputData.get("n_slices")                       # number of time slices for FIM calculation
    accuracy_order= inputData.get("accuracy_order" )               # finite-difference derivative accuracy (2, 4, 6, 8, or 10)
    dtheta_relative= inputData.get("dtheta_relative" )             # relative perturbation of the model parameters
    use_noise_normalization= inputData.get("use_noise_normalization")      # use noise normalization

    # Run Fimcore logic
    result = self.run_fim(initial_parameters, user_data, n_slices, accuracy_order, dtheta_relative, use_noise_normalization)

    # uptdate output data
    outputdata = self.writeOutput(inputData, result)

    # update json file
    test_config_json_path = json.dumps(outputdata)

    return test_config_json_path
```

Figure 6 Code of processing the input data to the Doe algorithm

restricted

5.2.2.2 Implementation of Deployment pipeline

The FIM library package is provided as a Python wheel file. Since this package is not available from the standard package-manager (pip) or the python standard library, we must ensure that this package is deployed properly. Furthermore, we must ensure, that deploying becomes mostly automated to allow fixes and modifications to be deployed without any additional work and effort.

To enable the deployment, an extra deployment script is developed, which automatically executes all the required steps:

- Disconnection VNET integration
- Creating fresh zip archive
- Run AZ deploy command
- Refresh Azure Function
- Adding VNET integration

6. Physical Testbench Selector

This chapter contains the steps needed to integrate and implement the Physical Testbench Selector. As stated in the requirements in chapter 3.2, the available testbenches must be provided by the partners. The testbench capabilities will be stored in the Azure blob storage as excel file. From there it can be easily accessed from the Azure Function

6.1 Integration

Similar to the integration of the DoE algorithm, the Physical Testbench Selector will be part of “DoE Function”. For details see chapter 5.2

In order to achieve a successful integration, additional code needs to be implemented. The following steps describe the information and data that needs to be integrated.

The data is stored as an excel file in Azure Storage and can be constantly updated here:

restricted

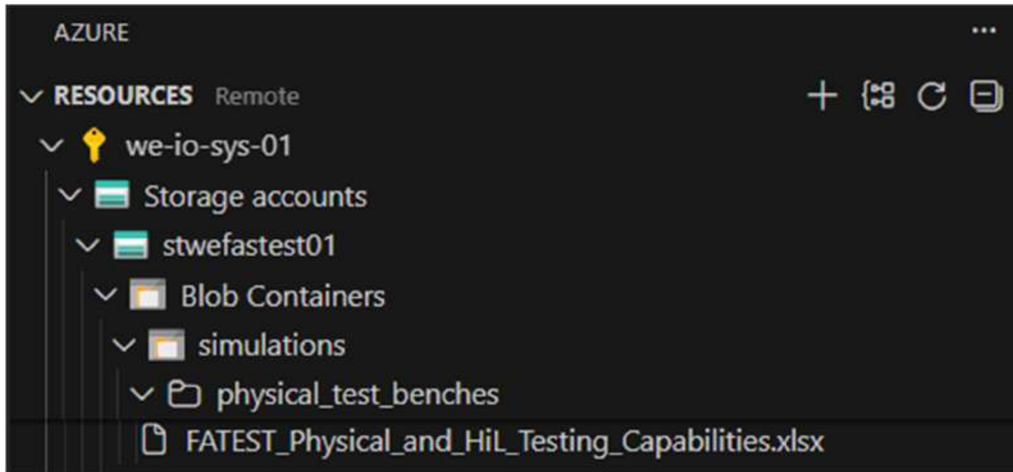

Figure 7 Test Bench Capabilities data in Azure Blob Storage

The table contains the required information about the existing test bench capabilities:

	A	B	C	D	H
1	Test bench	Test bench type	Use case	UUT ID	Test name
2	FLANDERS Make	Physical	Stationary	Gen3b - Cell	Aging and Performance - Preconditioning Test
3	FLANDERS Make	Physical	Stationary	Gen3b - Cell	Aging and Performance - Capacity Test
4	FLANDERS Make	Physical	Stationary	Gen3b - Cell	Aging and Performance - HPPC Test
5	FLANDERS Make	Physical	Stationary	Gen3b - Cell	Thermal - Thermal Cycling
6	FLANDERS Make	Physical	Stationary	Gen3b - Cell	Aging and Performance - Cycling Tests
7	FLANDERS Make	Physical	Stationary	Gen4 - Cell	Aging and Performance - Preconditioning Test
8	FLANDERS Make	Physical	Stationary	Gen4 - Cell	Aging and Performance - Capacity Test
9	FLANDERS Make	Physical	Stationary	Gen4 - Cell	Aging and Performance - OCV Test
10	FLANDERS Make	Physical	Stationary	Gen4 - Cell	Aging and Performance - HPPC Test
11	FLANDERS Make	Physical	Stationary	Gen4 - Cell	Thermal - Thermal Cycling
12	FLANDERS Make	Physical	Stationary	Gen4 - Cell	Aging and Performance - Cycling Tests
13	Flash Battery	Physical	Off-Road	Gen3b - Module	Aging and Performance - Cycling Tests
14	Flash Battery	Physical	Off-Road	Gen3b - Pack	Aging and Performance - Cycling Tests
15	Flash Battery	Physical	Off-Road	Gen4 - Module	Aging and Performance - Cycling Tests
16	Flash Battery	Physical	Off-Road	Gen4 - Pack	Aging and Performance - Cycling Tests
17	Ikerlan	HiL	Stationary	Gen3b - Module	Aging and Performance - Preconditioning Test
18	Ikerlan	HiL	Stationary	Gen4 - Module	Aging and Performance - Preconditioning Test

Figure 8 Test Bench capabilities attributes

From here the test bench capabilities data is extracted.

The test parameters are passed via the json trigger message and have the following dedicated structure:

```
"properties": {
  "test_name": {
    "description": "Name of the battery tests, lower and upper case sensitive",
    "type": "string",
    "enum": ["Capacity 15°C"]
  },
  "test_UUT": {
    "description": "In FASTEST, only following 6 UUT IDs have been defined",
    "type": "string",
    "enum": ["Gen3b-Cell"]
  },
  "test_bench": {
    "type": "string",
    "description": "The test bench information",
    "enum": ["Physical"]
  },
  "use_case": {
    "type": "string",
    "enum": ["Stationary"]
  },
```

Figure 9 Test data from json message

From this data the relevant information test_name, test_UUT, test_bench and use_case is taken and processed and compared to the available test bench capabilities found the the excel sheet.

The findings are returned by updating and returning the json file as message to the next function in the pipeline in LIMS.

6.1.1 Implementation

The implementation involves the complete software logic to access the test bench data and the test parameters, as well as implementing the logic to read the input data, check for a suitable testbench configuration and finally write the result into the output.

Implementation of function to read and parse the .json data:

```python
def get_test_details(self, json_data: dict):
    json_properties = json_data.get("properties")
    json_properties_use_case = json_properties.get("use_case")
    json_properties_test_UUT = json_properties.get("test_UUT")
    json_properties_test_name = json_properties.get("test_name")
    json_properties_test_bench = json_properties.get("test_bench")

    tc_type = json_properties_test_bench.get("enum") # e.g. "Physical"
    tc_usecase = json_properties_use_case.get("enum") # e.g. "Stationary"
    tc_UUT = json_properties_test_UUT.get("enum") # e.g. "Gen3b - Cell"
    tc_test_name = json_properties_test_name.get("enum") # e.g. "Electrical - Overcharge"

    test_details = {TEST_PARAMETERS[0]: tc_type[0],
                    TEST_PARAMETERS[1]: tc_usecase[0],
                    TEST_PARAMETERS[2]: tc_UUT[0],
                    TEST_PARAMETERS[3]: tc_test_name[0]
                    }
    return test_details
```

Figure 10 Code of parsing input from json message

The details which are taken into account are Test Bench Type, Use Case, UUT ID and Test name.

Implementation of function to look up the table for matching test utilities:

```python
def find_tests_in_excel(self, excel_rows, json_data, tol=1e-6):
    '''Checks excel and json input and returns matches if found'''
    test_details = self.get_test_details(json_data)
    matches = []
    for row in excel_rows:
        match = True

        for key in TEST_PARAMETERS:
            excel_value = row.get(key)
            json_value = test_details.get(key)

            if excel_value != json_value:
                match = False
                break
        if match:
            matches.append(row)
    return matches
```

Figure 11 Code of searching for test bench

Updating .json content:

```
def create_new_json_data(self, json_data: dict, new_test_bench_name):
    json_properties = json_data.get("properties")
    json_properties_test_bench = json_properties.get("test_bench")

    json_properties_test_bench["enum"] = [new_test_bench_name]

    return json_data
```

Figure 12 Code of updating the json message

Azure Blob Storage:

The data for the test bench capabilities is stored in the azure blob storage container. In order to access the container, we need to use the proper access methods.

Implementation of client object to access the storage:

```
credential = ManagedIdentityCredential()
blob_service_client = BlobServiceClient(
    f"https://{storage_account_name}.blob.core.windows.net",
    credential=credential
)
```

Figure 13 Code of client object initialization to access blob storage

```
container_name = "simulations"
excel_folder = "physical_test_benches"
excel_filename = "FATEST_Physical_and_HiL_Testing_Capabilities.xlsx"

simulations_container_client = blob_service_client_.get_container_client(container_name)
excel_blob_path = f"{excel_folder}/{excel_filename}"
excel_blob = simulations_container_client.get_blob_client(excel_blob_path)

blob_data = excel_blob.download_blob().readall()
logging.info(f"Downloaded {len(blob_data)} bytes")
```

Figure 14 Code of accessing and downloading test bench capabilities

The deployment process is the same as for the Doe algorithm integration. Both algorithms are contained in one deployment for doe functions.

7. RESULTS

7.1 Smart DoE for reduction of testing time and costs

The smart DoE algorithm visualized in Figure 2 has been implemented in Python combining models developed in WP3 with the FIM library that has been iteratively refined throughout the project (see also Deliverables D2.1 and D2.2). The implementation follows a modular architecture in which model evaluation, sensitivity analysis, and optimality assessment are decoupled but consistently interfaced. In particular, the integration of the Functional Mock-up Units (FMUs) enables a standardized and scalable execution of model simulations, while the FIM-library provides a unified framework for quantifying information content across varying test configurations.

From an implementation perspective, special emphasis was placed on robustness and flexibility. This includes handling different model structures, ensuring numerical stability in the computation of sensitivities, and enabling the efficient evaluation of multiple candidate test designs. The optimization routine operates on a defined set of degrees of freedom (e.g., current profile characteristics, C-rate, temperature, and initial SOC) and iteratively adjusts these parameters based on the selected optimality criterion. The algorithm thereby establishes an automated workflow that links model-based evaluation with test design adaptation.

Furthermore, interfaces to external systems, particularly the LIMS environment, have been prepared to allow seamless data exchange and execution of selected test configurations. This ensures that the algorithm is not only a standalone analytical tool but can be embedded into a broader experimental and data management infrastructure. Figure 15 presents a simplified and illustrative snapshot of the tool's functionality in the form of a terminal-based output.

```
Computing FIM for initial design...
D-optimality: 12.10
Test has a lower optimality: 12.10 <= 51.48
 --> optimizing experiment.
Running FIM-Analysis at 298.15 K and 90 % initial SOC...
D-optimality: 12.20
Found better D-optimality: 12.20 > 12.10. Updating experimental design.
Running FIM-Analysis at 298.15 K and 70 % initial SOC...
D-optimality: 11.77
Running FIM-Analysis at 298.15 K and 50 % initial SOC...
D-optimality: 15.49
Found better D-optimality: 15.49 > 12.20. Updating experimental design.
Running FIM-Analysis at 298.15 K and 30 % initial SOC...
D-optimality: 15.46
Running FIM-Analysis at 298.15 K and 10 % initial SOC...
D-optimality: 45.71
Found better D-optimality: 45.71 > 15.49. Updating experimental design.
Optimized test has a lower optimality than baseline threshold: 45.71 <= 51.48
 --> Run experiment virtually.
Saving optimized experimental design...
```

Figure 15 Output of the smart DoE algorithm for a 1-hour CC-discharge profile applied in context of an LFP-based cathode.

restricted

In this case, a standard 1-hour CC discharge profile is being analyzed and optimized in terms of the initial SOC over a set of pre-defined values (90%, 70%, 50%, 30%, 10%). The FIM-library interacts with the corresponding model that is based on a LFP cathode in this example and determines the optimal experimental design at the lowest initial SOC value of 10%. This is in line with the generally steeper voltage curve of LFP-based LIB cells at lower SOC ranges which causes the CC discharge at this initial SOC to be more informative in terms of the underlying model parameters. The result is finally compared with a previously computed threshold (corresponding to model parametrization) and - as it is lower - the DoE algorithm concludes that the physical execution of this test would not yield more information and the virtual test execution is triggered.

7.2 Test DoE Algorithm in Azure Function

After integration is finished, run the DoE function with test input data and document the output in Azure logs.

7.2.1 Test Description

Partners	**System Components**	**Date, Place**	**Status**
FEV	LIMS, DoE		Passed
Objectives of the Test			
• Test if the expected results have been returned to LIMS			
Test Description			
A predefined set of input parameters are sent to the Azure function and the output is monitored			
Open Issues			
• None			

7.2.2 Test Results and Evaluation

The test data are being processed in Azure as expected and the results are shown in the Azure Logs:

restricted

```
2026-01-09T07:32:12   [Information]   FimCore imported successfully
2026-01-09T07:32:12   [Information]   FIM: [[ 1755.31202029 -1735.43378995   -19.8782303 ]
 [-1735.43378995  1759.6834463    -24.2496564 ]
 [  -19.8782303    -24.2496564     44.1278867 ]]
2026-01-09T07:32:12   [Information]   log10(abs(FIM)): [3.24435433 3.23940805 1.29837772 3.24543455 1.38470559 1.64471313]
2026-01-09T07:32:12   [Information]   eig(FIM): [ 3.49293571e+03  6.61876476e+01 -1.26391438e-12]
2026-01-09T07:32:12   [Information]   singular values(FIM): [3.49293571e+03 6.61876476e+01 1.11698561e-12]
2026-01-09T07:32:12   [Information]   D-opt: -2.922033517449191e-07
2026-01-09T07:32:12   [Information]   E-opt: -1.2639143810067555e-12
2026-01-09T07:32:12   [Information]   T-opt: 3559.1233532919873
2026-01-09T07:32:12   [Information]
FimCore calculation finished successfully.
```

Figure 16 Log of Azure Function showing results from Fim calculations

Test Physical Test Bench Selector in Azure Function

After integration is finished, run the DoE function with test input data and document the output in Azure logs.

7.2.3 Test Description

Partners	**System Components**	**Date, Place**	**Status**
FEV	LIMS		Passed
Objectives of the Test			
• Test if the expected physical test bench has been selected			
Test Description			
N/A			
Open Issues			
• None			

7.2.4 Test Results and Evaluation

The following json file is taken as input file for the test:

```
"properties": {
  "test_name": {
    "description": "Name of the battery tests, lower and upper case sensitive",
    "type": "string",
    "enum": [
      "Capacity 15°C"
    ]
  },
  "test_type": {···
  },
  "test_UUID": {···
  },
  "test_UUT": {
    "description": "In FASTEST, only following 6 UUT IDs have been defined",
    "type": "string",
    "enum": [
      "Gen3b-Cell"
    ]
  },
  "test_bench": {
    "type": "string",
    "description": "The test bench information",
    "enum": [
      "Physical"
    ]
  },
  "use_case": {
    "type": "string",
    "enum": [
      "Stationary"
    ]
  },
```

Figure 17 Json message input data

The excel file is successfully downloaded and parsed from the blob storage:

```
2026-01-09T07:32:14   [Information]   Data found in execl: [{'Test bench': 'FLANDERS Make', 'Test Bench Type': 'Physical', 'Use Case': 'Stationary', 'UUT ID': 'Gen3b-
Cell', 'Cell nr': 'Cell 1', 'Test category': 'Beginning of Life', 'Test type': 'Preconditioning', 'Test name': 'Preconditioning 25°C', 'T_ambient': 25.0, 'Notes':
None}, {'Test bench': 'FLANDERS Make', 'Test Bench Type': 'Physical', 'Use Case': 'Stationary', 'UUT ID': 'Gen3b-Cell', 'Cell nr': 'Cell 2', 'Test category': 'Beginning
of Life', 'Test type': 'Preconditioning', 'Test name': 'Preconditioning 25°C', 'T_ambient': 25.0, 'Notes': None}, {'Test bench': 'FLANDERS Make', 'Test Bench Type':
'Physical', 'Use Case': 'Stationary', 'UUT ID': 'Gen3b-Cell', 'Cell nr': 'Cell 3', 'Test category': 'Beginning of Life', 'Test type': 'Preconditioning', 'Test name':
'Preconditioning 25°C', 'T_ambient': 25.0, 'Notes': None}, {'Test bench': 'FLANDERS Make', 'Test Bench Type': 'Physical', 'Use Case': 'Stationary', 'UUT ID': 'Gen3b-
Cell', 'Cell nr': 'Cell 4', 'Test category': 'Beginning of Life', 'Test type': 'Preconditioning', 'Test name': 'Preconditioning 25°C', 'T_ambient': 25.0, 'Notes':
None}, {'Test bench': 'FLANDERS Make', 'Test Bench Type': 'Physical', 'Use Case': 'Stationary', 'UUT ID': 'Gen3b-Cell', 'Cell nr': 'Cell 5', 'Test category': 'Beginning
of Life', 'Test type': 'Preconditioning', 'Test name': 'Preconditioning 25°C', 'T_ambient': 25.0, 'Notes': None}, {'Test bench': 'FLANDERS Make', 'Test Bench Type':
'Physical', 'Use Case': 'Stationary', 'UUT ID': 'Gen3b-Cell', 'Cell nr': 'Cell 6', 'Test category': 'Beginning of Life', 'Test type': 'Preconditioning', 'Test name':
```

Figure 18 Log of Test bench capabilities found in blob storage

From the given input file, a suitable test was found and returned:

```
✅ Test found: {'Test bench': 'FLANDERS Make', 'Test Bench Type': 'Physical', 'Use Case': 'Stationary', 'UUT ID':
 'Gen3b-Cell', 'Cell nr': 'Cell 6', 'Test category': 'Beginning of Life', 'Test type': 'Capacity', 'Test name': '
Capacity 15°C', 'T_ambient': 15.0, 'Notes': None}
```

Figure 19 Log of Test bench match for the input data

7.3 Deployment Test

This test's purpose is to check if the automatic deployment is running successfully.

Below in the screenshot is a log of the script while running. As we can see every step is executed automatically and the deployment finished successfully.

```
PS C:\Projects\FASTEST\GIT_Repository\IF_FASTEST\DoE> & "C:/Program Files/Python312/python.exe" c:/Projects/FASTEST/GIT_Repository/IF_FASTEST/DoE/doe_functions/auto_de
ploy.py

Disconnecting VNET integration...


Restarting Function App...


Starting main deployment process...

Using Azure CLI at: C:\Program Files\Microsoft SDKs\Azure\CLI2\wbin\az.CMD
Skipping requirements installation.
Creating ZIP file: c:\Projects\FASTEST\GIT_Repository\IF_FASTEST\DoE\doe_functions\deployment.zip
Added: .funcignore
Added: function_app.py
Added: host.json
Added: requirements.txt
Skipped (not found): fimcore-0.1.0-cp38-cp38-linux_x86_64.whl
Added folder: physical_bench_selector_src
Added folder: fim_core_src
Added folder: wheelhouse
ZIP file created successfully.
Starting deployment using cli command
'['C:\\Program Files\\Microsoft SDKs\\Azure\\CLI2\\wbin\\az.CMD', 'functionapp', 'deployment', 'source', 'config-zip', '--src', 'c:\\Projects\\FASTEST\\GIT_Repository\
\IF_FASTEST\\DoE\\doe_functions\\deployment.zip', '--name', 'func-we-fastest-doe-01', '--resource-group', 'rg-we-dev-01', '--build-remote', 'true']'...
Deployment completed successfully!
```

Figure 20 Log of automated deployment - part 1

```
{
  "active": true,
  "author": "N/A",
  "author_email": "N/A",
  "complete": true,
  "deployer": "az_cli_functions",
  "end_time": "2026-01-09T07:24:54.5081346Z",
  "id": "a7785db3-bc29-43b6-937f-cdfcba92e9b4",
  "is_readonly": true,
  "is_temp": false,
  "last_success_end_time": "2026-01-09T07:24:54.5081346Z",
  "log_url": "https://func-we-fastest-doe-01-b6c9ckeyd6e5f3e4.scm.westeurope-01.azurewebsites.net/api/deployments/a7785db3-bc29-43b6-937f-cdfcba92e9b4/log",
  "message": "Created via a push deployment",
  "progress": "",
  "received_time": "2026-01-09T07:24:22.2570433Z",
  "site_name": "func-we-fastest-doe-01",
  "start_time": "2026-01-09T07:24:23.8248388Z",
  "status": 4,
  "status_text": "",
  "url": "https://func-we-fastest-doe-01-b6c9ckeyd6e5f3e4.scm.westeurope-01.azurewebsites.net/api/deployments/a7785db3-bc29-43b6-937f-cdfcba92e9b4"
}


Adding VNET integration...

..Finished.
PS C:\Projects\FASTEST\GIT_Repository\IF_FASTEST\DoE>
```

Figure 21 Log of automated deployment - part 2

8. CONCLUSION & NEXT STEPS

All integration was implemented successfully and as planned. The algorithms can be used and triggered in Azure. The next steps are to run the whole pipeline including the DoE function and verify the majority of the process.

The DoE workflow integrates both the FIM-library developed earlier in WP2 as well as representative models for virtualization developed in WP3 and was tested successfully. Running and disseminating this workflow will be a continued process until the end of the FASTEST project in order to explore its full potential for

restricted

reducing testing time and costs via smart combination of physical and virtual testing.

Further adaptions during the complete integration in T6.6. are expected and can lead to minor modifications on the implementation regarding the DoE function integration and interaction with the other Azure services.

restricted

9. BIBLIOGRAPHY

HiveMQ. (2024, 10 30). *www.hivemq.com*. Retrieved from www.hivemq.com: https://www.hivemq.com/solutions/the-best-mqtt-broker-for-azure/

Microsoft. (2024, 10 30). *App Service - Build and Host Web Pages*. Retrieved from Microsoft Azure: https://azure.microsoft.com/en-us/products/app-service

Microsoft. (2024, 10 30). *Application Gateway*. Retrieved from Microsoft Azure: https://azure.microsoft.com/en-us/products/application-gateway/?msockid=333b0623c62160732cc912b7c75961f7

Microsoft. (2024, 10 30). *Azure Container Apps*. Retrieved from Microsoft Azure: https://azure.microsoft.com/de-de/products/container-apps/?msockid=333b0623c62160732cc912b7c75961f7

Microsoft. (2024, 10 30). *Azure Functions*. Retrieved from Microsoft Azure: https://azure.microsoft.com/en-us/products/functions/?msockid=333b0623c62160732cc912b7c75961f7

Microsoft. (2024, 10 30). *Azure SQL Database*. Retrieved from Microsoft Azure cloud: https://azure.microsoft.com/en-us/products/azure-sql/database/?msockid=333b0623c62160732cc912b7c75961f7#Features

Microsoft. (2024, 10 30). *Enable or disable SSH File Transfer Protocol (SFTP) support in Azure Blob Storage*. Retrieved from Microsoft Azure: https://learn.microsoft.com/en-us/azure/storage/blobs/secure-file-transfer-protocol-support-how-to?tabs=azure-portal

Microsoft. (n.d.). *Durable Functions*. Retrieved from https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview

Sidna, J., Amine, B., Abdallah, N., & El Alami, H. (2020). Analysis and evaluation of communication Protocols for IoT Applications. *Proceedings of the 13th international conference on intelligent systems: theories and applications*, (pp. 1-6).

restricted