Funded by the European Union

Call: HORIZON-CL5-2021-D5-01

Hyperconnected simulation ecosystem supporting probabilistic design and predictive manufacturing of next generation aircraft structures

CAELESTIS

Deliverable D2.2

CAELESTIS software and adaptative simulation workflow implementation

Work Package 2

HPC digital ecosystem and extended enterprise context

Document type	:	Other
Version	:	1.0
Date of issue	:	30/04/2023
Dissemination level	:	PUBLIC
Lead beneficiary	:	BSC

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or [EUROPEAN CLIMATE, INFRASTRUCTURE AND ENVIRONMENT EXECUTIVE AGENCY (CINEA)]. Neither the European Union nor the granting authority can be held responsible for them.

Funded by the European Union

Executive Summary

Abstract	This deliverable reports the results of the second phase of WP2. It includes the release of the software, the methodology to implement the CAELESTIS workflows and the implementation of the workflows implemented during the WP2.
Keywords	software, workflows

Revision history

Version	Author(s)	Changes	Date
0.1	Jorge Ejarque (BSC)	Table of Contents with assignments	11/03/2024
0.2	Pravin Luthada (ADD), Santiago Montagud (ESI), Riccardo Cecco (BSC), Jorge Ejarque (BSC)	Section contributions	6/04/2024
0.3	Jorge Ejarque (BSC)	Changes for internal review	9/04/2024
0.4	Verónica Mantecón(AIMEN)	Minor changes	09/04/2024
0.5	Guillaume Broggi (TUD), Francisco Serrano(ITA)	Internal review	16/04/2024
1.0	Pravin Luthada (ADD), Jorge Ejarque (BSC)	Address internal review comments	19/04/2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The CAELESTIS project aims to establish a digital simulation-driven ecosystem tailored for designing manufacturing strategies, with a focus on advancing the European aircraft industry. This initiative intends to facilitate the industry's evolution by enabling extensive exploration of design and manufacturing space to discover novel aerostructures and related systems.

The work package 2 of the CAELESTIS project has focused on the design and implementation of the CAELESTIS Interoperable Simulation Ecosystem (ISE) that integrates the Distributed Engineering Teams (DET) across aircraft industry with the High-Performance Computing (HPC) environments to enable the execution of complex product and process multi-scale and multi-physics simulation workflows. The first deliverable of the work package(D2.1), we provided the details on the design and implementation of the ISE as a software solution to manage the execution of simulation workflows that integrate different simulation software and data analytics techniques, and the cybersecurity analysis and recommendations to take into account for this ecosystem.

This deliverable releases the software components of the CAELESTIS ISE and the simulation workflows. The software components are released as beta version which are ready to be integrated with the demonstrator use case in WP7. The source code of these components is available in different Git repositories within the CAELESTIS Github organization. The URL of these repositories can be found in Table 1. Any change required due to the integration in WP7 or bug fix performed until the end of the project will be included in these repositories.

Table 1. Source code location for the implemented CAELESTIS ISE Components.

Component	Repository URL
AMLTool	https://github.com/CAELESTIS-Project-EU/AMLtool
HPC Simulation Service	https://github.com/CAELESTIS-Project-EU/Simulations_Service
External Software Executor	https://github.com/CAELESTIS-Project-EU/external_software_executor

Regarding the workflows, we have implemented several workflow templates which cover all the analysis algorithms used until now in the project. These templates define abstract phases that can be customized with different phase implementations which allow engineers to evaluate a single process and product simulations or different of them together without requiring to implement a full workflow every time. The workflow template, phases implementations and workflow description examples have also been stored in a repository in the CAELESTIS Github organization available in the following URL:

https://github.com/CAELESTIS-Project-EU/Workflows/

The rest of the document is divided in 3 main sections. Section 2 provides the details about how to install, configure the different components of the CAELESTIS ISE and how to use them to perform the workflow executions. Section 3 describes how the workflows' repository is organized and where we can find the different workflow templates and phases implementation as well as examples of how the different workflows are described. Finally, Section 4 describes how the system can be extended, to include new workflow templates and phases implementations.

2 WORKFLOWS EXECUTIONS USING THE CAELESTIS INTEROPERABLE SIMULATION ECOSYSTEM.

This section shows a brief guide about how to install, configure the CAELESTIS ISE Components and how to use them to execute a workflow. First, it provides the details about for the AML Tool, to define the Automation Markup Language (AutomationML/AML)[1] description of the workflow and the Hybrid Twin Platform (HTP) orchestration. Then, it shows how the External Execution Service is used to run software that is not able to run in the HPC sites. We have use as example the ADDPATH software, an Automated Fiber Placement (AFP) simulation software from Addcomposites. Finally, it describes how to install, configure and use the HPC simulation service.

2.1 Definition of the Workflows in AutomationML: AMLTool

The information required to launch a workflow on the HPC is contained into an AutomationML file. The AutomationML file consists in a template that needs to be filled for each particular workflow. To help non-trained users on filling in the information in the template, the AML tool has been developed. By using this tool, information like design of experiments (DoE) or required simulations are provided by the user, and some other information is stored automatically like the date, workflow identificatory, results folders, etc.

Installation and Configuration

The local desktop version is a python developed Graphical User Interface (GUI) that can be pulled from the CAELESTIS repository and used locally. The reported works in this deliverable are stored in the branch named '*Branch_Desktop_version'*, to be differentiated from future modifications. It can be done with the following command:

```
$ git clone https://github.com/CAELESTIS-Project-EU/AMLtool
$ cd AMLtool
[AMLtool]$ git checkout Branch_Desktop_version
```

There are no installation requirements rather than the python packages: *pyautomationml, lxml, os, json, datetime, xml* and *csv*. The application has been tested in Windows operating systems.

A configuration file is provided to set up some parameters, which can be find in the Github repository at *AMLtool/src/Config.json*:

- Predefined workflows
- Available outputs
- Communication parameters

Usage

The application must be launched by a python interpreter as shown in the following command:

```
[AMLtool]$ python AMLtool.py
```

Its use implies two steps:

1. The user must fill in the required information and upload the required files (see Figure 1). The current implementation requires:
    a. A csv file where the information related to the DoE is stored.
    b. To select a predefined workflow. This simplifies the development of a GUI and avoids invalid workflow generation by the user. For example, w3 consists of RTM and distortion simulations by PAM-RTM and PAM-DISTORTIONS solvers.
    c. To select the desired outputs. Launching several simulation tools in a large simulation campaign can generate a large amount of non-required information. To reduce this impact on power consumption, storage stress, etc., only required outputs will be stored.

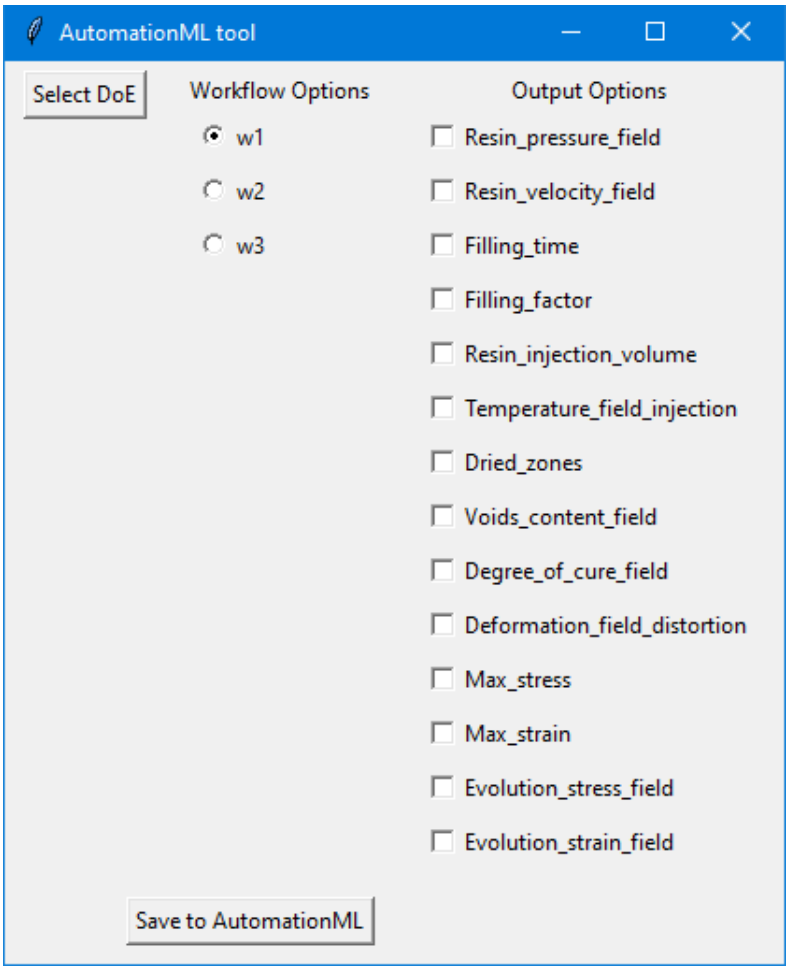

Figure 1: Desktop version of the AMLtool GUI

2. Save the workflow. A specific button saves the information into an AutomationML file. This is the file that contains the workflow information to be provided to the orchestrator for its execution.

2.2 Execution of the Workflow at HTP

The defined workflow requires an orchestrator, it is, a program that is able to read the required information from the AutomationML definition of the workflow, execute it and register its evolution and results. The orchestrator is also in charge of the communication with external dependencies in the CAELESTIS ecosystem (namely ADDPath software) and with the HPC.

The orchestrator is python-based and does not need any installation rather than the required python packages: pyautomationml, sys and subprocess. However, a creation of a user account

into the BSC simulation service is required with the correspondent security tokens, that need to be provided to the HTP orchestrator.

The orchestrator algorithm runs the following tasks, shown schematically in Figure 2:

1. Read the workflow definition from the AutomationML file. The workflow definition stores information about the user's requirement together with technical requirements. User requirements concern the inputs, the involved software and their outputs. Technical requirements concern communication protocols and other software needs.
2. Analyse data. Pre-computing checks are done on the extracted data. Some of them consists in data completeness or the requirements of external software requests.
3. External modules request. In the CAELESTIS ecosystem there are two software which are external to BSC infrastructure: ADDPath and JVN MecaMaster. The orchestration of this external modules depends on the HTP orchestrator.
4. Internal modules request. The most part of the CAELESTIS existing software or modules are installed in the HPC. The HTP orchestrator sends the required information to the BSC Simulation Service for its computation in the HPC.

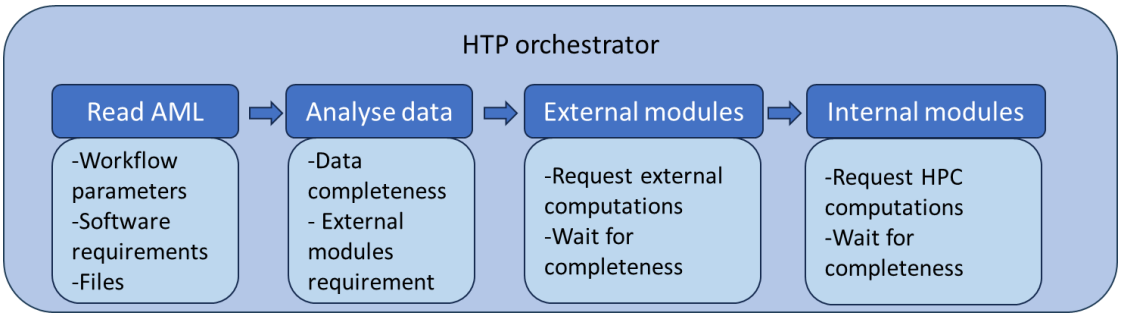

Figure 2 : Schematic representation of HTP orchestrator.

The execution of the orchestrator can be launched via the command:

```
$ python HTPorchestrator.py workflow_file.aml
```

The file *workflow_file.aml* must contain the information filled in by using the AMLtool.

A master AutomationML file and the executed workflows conform the so-called *digital thread.* The 'Master.aml' file references the existing executed workflows, working as a list of workflows

with meta-data for their quick identification. The executed workflows are stored in different AutomationML files containing all relevant information.

2.3 Execution of External Deployed Software: ADDPath Execution

To enable the execution of Software that cannot be executed in the HPC, we have developed a simple web Application Programming Interface (API) following the Representational State Transfer (REST) specification . to enable the remote execution of a software. This REST API is developed with Flask , a Python library which facilitates the implementation of this software. The server where we deploy the API must have the software we remotely call installed and must be accessible from the HTP with a public IP and allow the incoming request in the TCP where service is listening.

The following paragraphs show how install configure and use this API to remotely execute external software. We use the execution of ADDPath as an example of a software that we run in the CAELESTIS workflows, but the same procedure can be used for other software.

Installation and Configuration

The REST API for enabling the external software executor must be downloaded from the Github repository using the following command:

```
$ git clone https://github.com/CAELESTIS-Project-EU/external_software_executor.git
```

After downloading the software, the used python modules must be installed using the following command:

```
$ cd external_software_executor
[external_software_executor]$ pip install requirements.txt
```

Finally, the service database must be created invoking the following python script.

```
[external_software_executor]$ python reload_db.py
```

The external software API must be configured to allow the execution of the desired software. It is done internally by the service administrator modifying the *available_software.json file* located in the *config* folder. This file must contain a line where the software name in the API request is map to a command template or python function which describes how to run the desired software. Note that with due to this configuration, the external user will be only able to run the software allowed by the administrator, providing the software name not the commands to avoid any potential security risk. The mentioned configuration file looks like the following one.

```
{
        “<request_software_name>” : {
                “type” : “command”,
                “command“: “</path/to/software/executable> {<request_param>}”
        }
}
```

To enable the execution of ADDPath in a remote machine, the ADDPath software must be installed in this machine. Follow the instructions in this link (Link) to install ADDPath. This installation deploys an internal service which listen the simulation request and run the simulations.

The REST API mentioned above has been modified to include the call to the ADDPath internal service to perform the simulations requested through the REST API. This modification can be found in the *addpath* branch in the Github repository. Use the following command to change to the *addpat*h branch in the cloned repository.

```
$ git checkout addpath
```

To start the REST API to allow the remote executions, run the following command.

```
$ python app.py
```

Usage

The user need to follow these steps, these steps are also demonstrated in a video here.

- **Step 1:** In order to provide ply and laminate information without opening ADDPath, a simple configurator is created that can be accessed here. By double clicking on the ConfigGen.exe it opens an interface to input all the key details regarding the ply and laminate shown in Figure 3. Based on user requirement they can input the ply and laminate data. Once completed, click on Generate button and this should create the *config.json* file. Make sure to save it in the folder where the experiment is run.

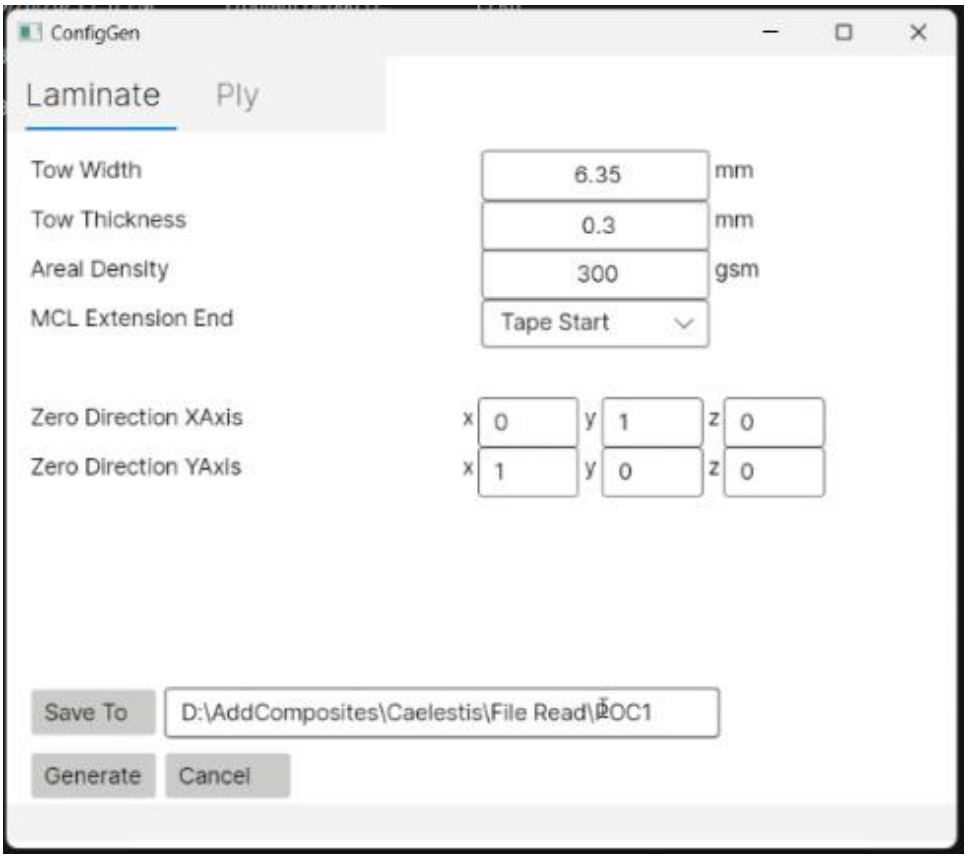

Figure 3 Configuration file generator for AddPath

- **Step 2:** Prepare layup and boundary surface files in *.stp* format, as shown in the Figure 4. The layup surface defined as the area onto which the intended fibre placement can be carried out by definition. The boundary surface is defined as surface that is a continuous surface adjacent to layup surface and allows for any unintended or overflowing fibre placement to be carried out. Both the surface files are a onetime input into the simulation and are provided by the end user as a definition of the layup. An example from POC2 is shown below. Sample files for the same can also be found here.

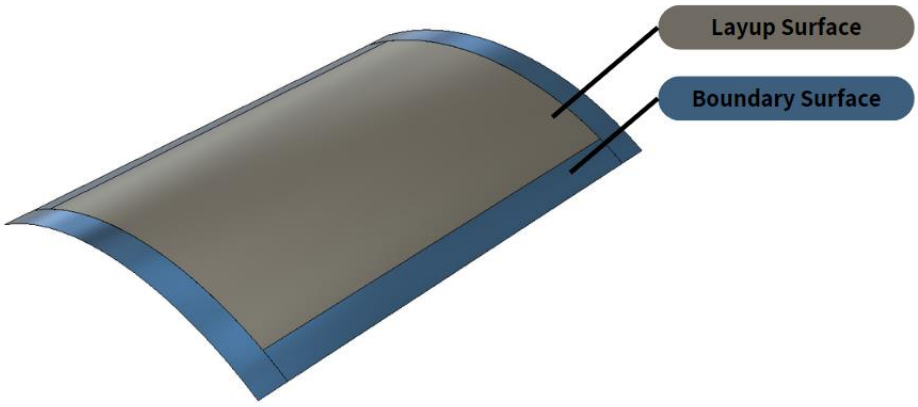

Figure 4 Layup and boundary surface files to be provided to AddPath

- **Step 3**: Provide a point cloud of the centroids of the PAM-RTM mesh in the .txt format. The file name should be **centroid.txt** and should be placed in the same folder as other files. This file is provided by the user or generated by the workflow depending on the use case.
- **Step 4:** Upload the files in a folder in the FTP Storage service of the CAELESTIS platform following this structure "*/P_CAELESTIS_SHARE/addpath/Input/<ExperimentNumber>*". The FTP access must be requested to AIMEN which is administrating the service.
- **Step 5:** Once the files are uploaded, the user sends an HTTP request to the external service to initiate the simulation. An example of this request is shown below.

```
$ curl -u <username>:<password|token> \
      -X POST -H "Content-Type: application/json" \
      -d '{"software": "addpath", "parameters": { \
            "dir": "/P_CAELESTIS_SHARE/addpath/Input/<ExperimentNumber> "} }' \
     http://<AddPath_IP_Address>:<Port_Number>/run
```

Users must browse to http://<AddPath_IP_Address>:<Port_Number>/signup in order to get a username and password (or generate an API token) for accessing the external software executor.

- **Step 6:** The simulation generates the layup with suggested parameters and outputs a .txt files with updated parameters which can be fetched from the specific directory of the FTP server *"/P_CAELESTIS_SHARE/addpath/output/<ExperimentNumber>*".

2.4 Execution of workflow at HPC: Simulation Service

The CAELESTIS HPC Simulation service is a web service that provide a Graphical User Interface and a REST API to manage the execution of simulation workflows in the HPC infrastructures. The HPC Simulation Service is implemented using Django[2], a python library for developing web services with python. This service requires a PostgreSQL[3] database to store the users, machines and simulations metadata. The implemented Django service must be deployed in a Linux Server using NGINX[4] as web server and Gunicorn[5]. NGINX receives the user's web request and redirects to Gunicorn which manages the Django application execution inside the server. In the following paragraph, we provide the details about how to deploy the HPC Simulation Service in a server. Some commands of the following instructions are only valid for Ubuntu distributions. Similar commands can be used for other distributions.

Installation Instructions

To install the service in a production server you have the run the following steps:

- **STEP 1: Downloading the source code.**

The CAELESTIS HPC Simulation Service can be download from its Github repository using the *git clone* command as shown in the following command.

```
$ export SIM_SERVICE_DIR=/path/to/simulations_service/
$ git clone https://github.com/CAELESTIS-Project-EU/Simulations_Service.git \
              $SIM_SERVICE_DIR
```

- **STEP 2: Installing the Software requirements from OS packages (Ubuntu)**

After downloading the code, we need to install the required software dependencies from the OS packages as depicted below.

```
$ sudo apt install python3-venv python3-dev libpq-dev postgresql \
                    postgresql-contrib nginx curl
```

- **STEP 3: Creating a Python Virtual Environment for your installation**

Finally, you can create a new Python virtual environment inside the simulation service directory to install the required python modules.

```
$ python3 -m venv myprojectenv
$ source myprojectenv/bin/activate
(myprojectenv) $ pip install django gunicorn psycopg2-binarysudo
```

Configuration Instructions

Once we have installed the required software and Python modules, we need to configure the database and the Django framework and setup the Gunicorn and Nginx services to serve the simulation service.

- **STEP 1: Creating the PostgreSQL Database**

In the first step we need to generate the database and create a user to interact to this database from the simulation. To do it run the following commands, replacing *{myprojectuser}* and *{password}* with the chosen username and password for your user.

```
$> sudo -u postgres psql
postgres> CREATE DATABASE caelestis_db;
postgres> CREATE USER {myprojectuser} WITH PASSWORD '{password}';
postgres> ALTER ROLE {myprojectuser} SET client_encoding TO 'utf8';
postgres> ALTER ROLE {myprojectuser} SET \
                  default_transaction_isolation TO 'read committed';
postgres> ALTER ROLE {myprojectuser} SET timezone TO 'UTC';
postgres> GRANT ALL PRIVILEGES ON DATABASE caelestis_db TO {myprojectuser};
Postgres> \q
```

- **STEP 2: Configuring Django.**

To configure Django for the current deployment, we must link the Django configuration to the database created in the previous step. To do it Within, navigate to the DATABASES section the *settings.py* file and update the *NAME* property to '*caelestis_db*', as well as replace USER and PASSWORD with the corresponding '*{myprojectuser}*' and '*{password}*' of the user created in the previous step.

```
(myprojectenv) $ nano $SIM_SERVICE_DIR/login_register_project/settings.py
DATABASES = {
      'default' : {
            'ENGINE' : 'django.db.backends.postgresql_psycopg2',
            'NAME' : 'caelestis_db',
            'USER' : '{myprojectuser}',
            'PASSWORD : '{password}',
            'HOST' : 'localhost',
            'PORT' : ''
      }
}
```

After linking the database, execute the following commands to complete the Django setup for this simulation service deployment.

```
(myprojectenv) $ $SIM_SERVICE_DIR/manage.py makemigrations
(myprojectenv) $ $SIM_SERVICE_DIR/manage.py migrate
(myprojectenv) $ $SIM_SERVICE_DIR/manage.py createsuperuser
(myprojectenv) $ $SIM_SERVICE_DIR/manage.py collectstatic
```

At this point, a development instance of the Simulations service can be started using the following command.

```
(myprojectenv) $ $SIM_SERVICE_DIR/manage.py runserver 0.0.0.0:8000
```

Then, the service can be internally accessed in the following URL http://127.0.0.1:8000. This deployment is useful to test everything is correctly setup or for development purposes. For production deployments, follow the next steps to serve the Django service using NGINX and Gunicorn.

- **STEP 3: Setting up Gunicorn**

Gunicorn was installed as a Python module in the virtual environment in Step 3 of the Installation instructions. In this section, we are going show how to set up the Gunicorn as an OS system service which will serve the Simulations Service and will be started at boot time. The part of this configuration step is writing the *gunicorn.socket* file as described below.

```
(myprojectenv) $ nano /etc/systemd/system/gunicorn.socket
[Unit]
Description=gunicorn socket
[Socket]
ListenStream=/run/gunicorn.sock
[Install]
WantedBy=sockets.target
```

After defining the socket we have to define the Gunicorn system service writing the *gunicorn.service* file as described below. Replace *<sim_service_dir>* with the path where you downloaded the simulation service in Step 1 of the Installation Instructions, *<username>* by your username in the server, and <procs> by the number of workers you want to deploy to serve the Django service. The optimal number of workers depends on the expected load of users and the number of CPU cores available in your server. If your server is just dedicated to the service, set the number of workers equal to the number of available CPU cores.

```
(myprojectenv) $ nano /etc/systemd/system/gunicorn.service
[Unit]
Description=gunicorn daemon
Requires=gunicorn.socket
After=network.target
[Service]
User=<username>
Group=www-data
WorkingDirectory=<sim_service_dir>
ExecStart==<sim_service_dir>/myprojectenv/bin/gunicorn –access-logfile - \
             --workers <procs> –bind unix:/run/gunicorn.sock \
             login_register_project.wsgi:application
[Install]
WantedBy=multi-user.target
```

To finalize the Gunicorn configuration, the socket and service must be enabled to be started at server boot. It can be done with the following commands.

```
$ sudo systemctl enable gunicorn.socket
$ sudo systemctl enable gunicorn
```

Finally, the Gunicorn socket and service can be started with the following commands.

```
$ sudo systemctl start gunicorn.socket
$ sudo systemctl daemon-reload
$ sudo systemctl start gunicorn
```

- **STEP 5: Configure Nginx to Proxy Pass to Gunicorn**

If we want to serve the Simulation Service in the standard HTTP ports (80 or 443) we have to serve Gunicorn through the Nginx HTTP server. To do it, you need to configure Nginx Proxy Pass to forward the HTTP network packages to the Gunicorn socket creating a new configuration file in the */etc/nginx/sites-available/* folder. This file must look like the one below, replace *<sim_service_dir>* with the path where you downloaded the simulation service in Step 1 of the Installation Instructions and <sever_domain> by the fully qualified domain name of your server.

```
$ nano /etc/nginx/sites-available/sims_service
server {
      listen 80;
      server_name <server_domain>;
      keepalive_timeout 300;
      location / {
            include proxy_params;
            proxy_read_timeout 600s;
            proxy_connect_timeout 600s;
            proxy_apss http://unix:/run/gunicorn.sock;
      }
      location /static/ {
            autoindex on;
            alias <sim_service_dir>/static/;

      }
```

To activate the new proxy pass, you have to run the following commands:

```
$ sudo ln -s /etc/nginx/sites-available/sims_service /etc/nginx/sites-enabled
$ sudo nginx -t
$ sudo systemctl restart nginx
```

Now, users must be able to access the service at the URL http://<server_domain_or_IP>

If the service is not available in the URL, a frequent cause is because you do not have the HTTP ports open in your firewall. In Ubuntu distributions, you can open it by running the following command.

```
$ sudo ufw allow 'Nginx Full'
```

Usage Instructions

The user guide including the usage instructions and the REST API documentation has been created as an online documentation using the ReadTheDocs platform[6]. This platform allows developers to maintain up to date documentation for different versions of software. This documentation can be found in the following link:

https://caelestis-project-eu-simulations-service.readthedocs.io

3 IMPLEMENTED HPC WORKFLOWS

To reduce the developments required to implement the simulation workflows in CAELESTIS we have follow an approach of customizable workflow templates and phases. A workflow template that implements the algorithm of an analysis that contain abstract phases. These abstract phases are part that can be plug-in with different phase implementations. It allows engineers to run analysis for different parts of the manufacturing process without requiring to implement a new workflow from scratch. To define a workflow to run, engineers must select a workflow template and the implementations for each of the abstract phases in the workflow template. This workflow definition can be simplified with the AMLTool as explained in Section 1. This tool offers predefined workflows description where the template and phases are already selected, and the final user only need to select the inputs and outputs for the execution.

Workflow templates, phases implementations and workflow definition examples are stored in the following GIT repository:

https://github.com/CAELESTIS-Project-EU/Workflows/

This repository contains 3 main folder:

- *WORKLFOWS* folder: It stores the implementation of the available workflow templates.
- *PHASES* folder: It stores the implemented phases for the different abstract phases defined in the workflow templates.
- *Examples* folder: It stores the different workflow definitions used to test the different analysis for different process and product simulations.

3.1 Workflow Templates

In the WORKFLOWS folder we can find different subfolders that stores the implementation of a workflow template. Until now, we have implemented the following workflow templates:

- SENSITIVITY ANALYSIS: Sensitivity analysis is a systematic process used to understand how changes or variations in the values of input variables of a model affect the results or outcomes produced by that model. Sensitivity analysis is particularly valuable when dealing with complex models or when input data includes uncertainties. It helps decision-makers understand the reliability and robustness of their models and supports informed decisions by providing insights into the potential consequences of different parameter variations.

  This workflow defines a first *"sampler"* phase to perform a sampling of the design space to obtain the cases to simulate. For each sample case, three phases are defined: a "prepare_data" phase where the simulation input configuration is generated from the sample, then it has a "simulation" phase, where the simulation software is invoked and the post_process phase, where the output of the simulation is processed to obtain the output values to evaluate. Finally, all the outputs are provided to the sensitivity phase where the sensitivity analysis is performed, and results are written to a file or folder.

- MONTECARLO: A Monte Carlo simulation is a computational technique used to estimate complex mathematical results by using random sampling and probability distributions. It's particularly useful when deterministic methods are impractical or impossible to apply due to the complexity of a problem, the involvement of numerous variables, or the presence of uncertainty.

  This workflow has the same initial part as the sensitivity analysis workflow, with a sampler phase to generate the DoE cases and for each case it also has the "prepare_data", "simulation" and "post_process" phases to extract quantities of interest. Then it has a "uncertainty_quatification" phase which computes the uncertainty quatification statistics.

- MODEL TRAINING: This workflow addresses the challenges posed by time and resource-intensive complex simulations within this project. It entails the creation and training of an artificial intelligence model capable of generating results based on predetermined

parameters, eliminating the need for running simulations. The model is continuously trained and updated using input data and results from previous runs. It starts with the same simulation part as in the other workflows but in this case it performs a model selection algorithm where we can define as phases the search algorithm, the ML kernels and parameters to evaluate. The best model is serialized to a file in order to be downloaded and used at operation time.

Other subfolders store other workflow templates that we have used during implementation to test the integration of new workflows but, at the end, we have been able to integrate them in the common templates mentioned before.

3.2 Workflows Phases

The PHASES folder stores the implementations of the different abstract phases of the workflow templates. It is organized in the following subfolders:

- SAMPLERS: In this folder, we store the implementations of the sampler phases. This phase is usually used as the first step of the workflows, and it creates the sample set using different techniques for sensitivity analysis and Design of Experiments. There are also samplers that basically read the sample cases from a CSV file or similar and return them in the expected format. The inputs of this phase can have as whatever input parameters and returns a 2D array where each row is a sample to evaluate.

- BEFORE SIMULATION: This folder stores the implementations of the data pre-process algorithms. It gets the sample for each case and converts them to the expected inputs for the simulator. This step receives the working folder for the case to evaluate, the case number, the sample values of the case in a python dictionary format as well as other optional parameters defined for each implementation of this phase.

- SIMULATION: This folder stores the python modules that run the simulation software. These modules receive the input generated by the before simulation phase and the working case folder as well as other parameters defined in the workflow description. The results are stored in the working folder.

- POST SIMULATION: This folder stores the implementation of the post-processing phases. These implementations are used to collect the results generated by the simulations run in parallel, post-process them and compute an array of values required by other phases of the workflows (Sensitivity, UQ, Model Training, …).

- SENSITIVITY: This folder stores the implementations of the sensitivity analysis methods. This phase receives as input the sampling and results arrays and performs a sensitivity analysis whose outcome provides the impact of the input variables on the result.

- MODEL TRAINING: This folder stores the implementation of the python modules used in the different phases of the model training. It also receives as input the sampling and results arrays which are converted to a training and validation dataset to perform the model selection and training.

3.3 Workflow descriptions

In the Examples folder we store the descriptions of the workflows. In this folder, we can see examples defined in AutomationML format and YAML format. The AutomationML format is recommended for industry 4.0 and this is the format supported at HTP and HPC levels. The YAML format is easier to integrate with python because it has a direct mapping to python data structures. We used YAML for the initial implementations of the HPC simulation service, so we have kept the support to this format in the HPC part, but it is not supported at HTP level. From the perspective of the description of the workflow to be executed, both formats contain the

same information and are equivalent. The YAML format is more human readable due to a lower level of verbosity, therefore the examples shown in the document will be in YAML format.

A workflow description looks like the following YAML file where the user has to define the template selected for the workflows in the *workflow_type* section, the implementation selected for each of the workflow phase in the *phases* section, the external inputs and outputs data movements required in the workflow execution are specified in the *inputs* and *outputs* section, other parameters used in the workflow are specified in the *parameters* section, and environment variables to be set during the execution can be also specified in the *environment* section in a key-value .

```
workflow_type: WORKFLOWS.SENSITIVITY_ANALYSIS.workflow.execution
phases:
  sampler:
    …
  prepare_data:
    …
  sim:
    …
  post_process:
    …
  sensitivity:
    …
inputs:
  …
outputs:
  …
parameters:
  -   problem:
      - num_vars: 21
      - variables-sampler:
            - E11: {mean: 171420.0, cov: 1.39}
      …
environment:
  -   ALYA_PROCS: 64
```

Each phase is specified as shown in the following example. User has to specify the implementation of the phase in the *type* field and, in the *arguments* field, the extra arguments required to run the phase implementation. We can refer the value of a phase argument to workflow parameter, input/output data path or internal variables of the workflow. In this example, the *problem* argument defined in the *sampler* phase is linked to the value of *problem* parameter defined in the *parameters* section of the workflow description.

```
workflow_type: WORKFLOWS.SENSITIVITY_ANALYSIS.workflow.execution
phases:
  sampler:
    type: PHASES.SAMPLERS.morris.sampling
    arguments:
      - r: 20
      - p: 16
      - problem: ${parameters.problem}
…
```

To describe the input and output data transfers required by the workflow, users have to specify the path where to store the data in the computing site and the URL where to store the data in the Storage server. They can also specify a flag to overwrite if the file already exists. An example of these descriptions is shown below.

```
…
inputs:
  mesh:
    - server: "ftp://nas.aimen.es/P_CAELESTIS_SHARE/input/meshes/OHT_Validation_D2"
    - path: "meshes/OHT_Validation_D2"
outputs:
  sesitivity_report:
    - path: results.txt
    - server: "ftp://nas.aimen.es/P_CAELESTIS_SHARE/outputs/test2J"
    - overwrite: true
…
```

4 EXTENDING THE INTEROPERABLE SIMULATION ECOSYSTEM

Developers can extend the CAELESTIS ISE can be extended three ways: including new workflow templates, including new phases to be used in the workflow templates, and adding new simulation software. The next paragraphs show how to proceed in each of the cases.

4.1 Including new workflows templates

To include a new workflow template, developers must include a new python module in the WORKFLOWS folder of the Github repository. To do it, run the following command to clone the repository and create a new branch to establish a new version of the CAELESTIS workflows.

```
$ git clone https://github.com/CAELESTIS-Project-EU/Workflows.git
$ cd Workflows
~/Workflows/$ git checkout -b my_new_version
```

Then create a new sub-directory within the WORKFLOWS directory and create an empty *__init__.py* file inside the new folder to indicate that it is a Python module.

```
~/Workflows/$ cd WORKFLOWS
~/Workflows/WORKFLOWS/$ mkdir <NEW_WORKFLOW_NAME>
~/Workflows/WORKFLOWS/$ cd <NEW_WORKFLOW_NAME>
~/Workflows/WORKFLOWS/<NEW_WORKFLOW_NAME>/$ touch __init__.py
```

Then, create a Python script file where you will implement the code workflow template. This script must have the shape shown in the following example. First, developers must import the *PHASE.utils* module to run the phases defined in the workflow description. Then, they must include the function that implements the workflows behaviour. This function must include the calls to the different phases using the *phase.run* method included utils module imported at the beginning of the file. The interface of the workflow function must include the following arguments:

- **execution_folder**: it's the path to the directory created to run de workflow.
- **data_folder**: it's the path where the input data is stored when a relative path is defined.
- **phases**: it's the description of the workflow's phases (obtained from the workflow description)
- **inputs**: Key-value map where the key is the input name and the value is path to the inputs of the workflow (obtained from the workflow description)
- **outputs:** Key-value map where the key is the output name and the value is the path where outputs must be stored at the end of the workflow execution (obtained from the workflow description)
- **parameters:** Key-value map storing the parameters of the workflow (obtained from the workflow description)

```
~/Workflows/WORKFLOWS/<NEW_WORKFLOW_NAME>/$ nano my_new_workflow.py
from PHASES.utils import phase
from pycompss.api.api import compss_wait_on
…
def execution(execution_folder, data_folder, phases, inputs, outputs, parameters):
   …
   phases_output = phase.run(phases.get("{name_phase}"), inputs, outputs, parameters, \
                             data_folder, locals())
   phase_output = compss_wait_on(phase_out)
   …
```

CAELESTIS Workflows are implemented using the PyCOMPSs programming model [7] where computations are executed in remote and asynchronous way to allow to exploit the inherent parallelism of the workflow. Due to this fact, if we want to inspect the values of the data generated in a phase in the main function of the workflow (*execution* method in the example), developers need to use the *compss_wait_on* method passing the data that we want to synchronize. This method waits until the phase that generates the data and synchronize its

value. After the *compss_wait_on,* data can be accessed as a normal python data. More information about how to develop PyCOMPSs workflows can be found in the following link.

https://compss.readthedocs.io/en/stable/Sections/02_App_Development/02_Python.html

As commented above, developers must call the *phase.run* function to run a phase defined in the workflow registry. This method receives the phase description (obtained with *phases.get("{name_phase}"),* and all the values of inputs/outputs/parameters from the workflow description and the local variables defined in the workflow code. This function will execute the function indicated in the *type* tag of the phase definition with the defined arguments, substituting the references to the input, output, parameters and variables by their corresponding values.

For the phase description shown below, the *phase.run* method will execute the *PHASES.BEFORESIMULATION.alya.prepare_data* function, whose arguments' values will be obtained from the inputs, parameters sections of the workflow description or the variables defined in the workflow code.

```
phases:
  prepare_data:
    type: PHASES.BEFORESIMULATION.alya.prepare_data
    arguments:
      - mesh: $inputs.mesh
      - template_sld: $inputs.template_sld
      - template_dom: $inputs.template_dom
      - problem: $parameters.problem
      - simulation_wdir: $variables.simulation_wdir
      - values: $variables.values
      - name_sim: $variables.name_sim
      - original_name_sim: $variables.original_name_sim:
```

Once the developer has implemented the workflow, it can be pushed to the git repository with the following command.

```
~/Workflows/$ git pull origin my_new_version
```

The implementation of the new workflow template can be tested indicating the branch when submitting the workflow in the HPC service. Once the development is correct, it can be merged to the main branch by creating a pull request in the Github repository.

4.2 Including new phases

To include new phases, developers have to follow a similar approach than for including a new workflow template. They must clone the repository and create a new branch as indicated with the following commands:

```
$ git clone https://github.com/CAELESTIS-Project-EU/Workflows.git
$ cd Workflows
~/Workflows/$ git checkout -b my_new_version
```

Then, they also must create a new python module within the PHASES folder.

```
~/Workflows/$ cd PHASES
~/Workflows/PHASES/$ mkdir <NEW_PHASE_NAME>
~/Workflows/PHASES/$ cd <NEW_PHASE_NAME>
~/Workflows/PHASES/<NEW_PHASE_NAME>/$ touch __init__.py
```

As commented in the previous section, CAELESTIS workflows are implemented using the PyCOMPSs programming model. We support two main types of phases: Single task phases, which includes a single computation; and subworkflow phases which include small algorithm

that can execute several parallel computations. For single task phases, phases which are just python code or executions of simulation software that we will explain in the next section.

The creation of single python computation phase, we just need to import the PyCOMPSs decorators and parameter types and create a python function which implements the computation and annotate it with PyCOMPSs *@task* decorator as shown in the next example.

```
~/Workflows/PHASES/<NEW_PHASE_NAME>/$ nano my_new_phase.py
from pycompss.api.task import task
from pycompss.api.parameter import *
…
@task(path_A=DIRECTORY_IN, path_B=FILE_OUT, returns=2)
def my_phase_funtion(path_A, path_B, **kwargs):
   # phase functionality implementation as standard python code.
   return a, b
```

The task decorator is used to indicate that the functions is going to be considered as a synchronous execution and developers must indicate the type and direction about how the phase parameters are used inside the phase. By default, all arguments are considered IN objects that are just read. If one of the function parameters is a path. Developers must indicate if this path is a file or directory, to avoid an incorrect interpretation by the runtime. More details and options about how to define tasks in PyCOMPSs is defined in the following link.

To include this phase in a workflow, the user has to define the phase invocation in the workflow description as shown in the following example.

```
phases:
  <phase_name>:
    type: PHASES.<NEW_PHASE_NAME>.<my_new_phase>.<my_phase_function>
    arguments:
      - path_A: …
      - path_B: …
      …
```

To create subworkflow phases, developers have to define a function annotates as task for each type of asynchronous invocation that want to include in the phase and another function which implements the subworkflow behaviour without the task decorator. An example of these definitions is depicted in the following code snippet.

```
~/Workflows/PHASES/<NEW_PHASE_NAME>/$ nano my_new_phase.py
from pycompss.api.task import task
from pycompss.api.parameter import *
…
@task(input_file=FILE_IN, returns=1)
def process(input_file):
   # phase functionality implementation as standard python code.
   return out
@task(accum=INOUT)
def merge(accum, new_data):
   accum.update(new_data)

def my_phase_function(files, **kwargs):
   accum = Acum(0)
   for file in files:
       out = process(file)
       merge(accum, out)
    return compss_wait_on(accum)
```

Once the developer has implemented the new phase, it can be pushed to the git repository with the following command.

```
~/Workflows/$ git pull origin my_new_version
```

The implementation of the new phase can be tested indicating the branch when submitting the workflow in the HPC service. Once the development of the phase has been validated, it can be merged to the main branch by creating a pull request in the Github repository.

4.3 Including new Simulation Software

In the previous section, we have shown how to create phases from standard python code, which is very useful in sampling and data processing phases. However, workflow phases we can also include class to Simulation Software that can use different processors and nodes of an HPC system. In this section, we describe how to define this type of phases. The initial procedure will be the same as for the other type of phases. Developers must clone the repository, create a new branch and python module. The main difference is the way that the phase function is defined depending on the type of execution required by the Simulation Software. We support simulation software that runs in different cores and different nodes (using MPI or other approaches) as well as Software distributed in containers. For each of these cases, we have to add some decorators on top of the task definition. Table 2 shows the decorators to add in each of the mentioned cases.

Table 2. Decorator to be included in the phase definition depending on the Simulation Software execution.

Execution type	Executable binary	Multiple CPU cores	Multiple Nodes		Container
			MPI	Other	
Decorator	@binary	@constraints	@mpi	@multinode	@container

The following example shows how to define an execution of a multi-core executable with containers and an execution of an MPI application.

```
~/Workflows/PHASES/<NEW_PHASE_NAME>/$ nano my_simulation_software.py
from pycompss.api.task import task
from pycompss.api.constraint import constraint
from pycompss.api.binary import binary
from pycompss.api.container import container
from pycompss.api.mpi import mpi
from pycompss.api.parameter import *
…
@constraint(computing_units=4)
@container(engine='SINGULARITY', image="$SOFTWARE_CONTAINER_IMAGE")
@binary(binary="software.bin", args="-i {{input_file}} -o {{output_file}}"
@task(input_file=FILE_IN, output_file=FILE_OUT)
def binary_software(input_file):
   # No implementation required
   pass

@mpi(binary="Alya.x", processes="$ALYA_PROCS", working_dir="{{simulation_dir}}")
@task(simulation_dir=DIRECTORY_INOUT)
def mpi_software(simulation_dir):
   pass
```

More details about how to use these decorators are available at the following link.

https://compss.readthedocs.io/en/stable/Sections/02_App_Development/02_Python/01_1_Task_definition/Sections/06_Other_task_types.html

5 CONCLUSION AND FUTURE WORK

Deliverable D2.2 releases the beta version of the CAELESTIS ISE components as well as the workflow templates and phases implemented identified and implemented during WP2 to perform the analysis and surrogate models required for the project. The source code of the components, workflow templates and phases are stored in the CAELESTIS Github organization.
This document has provided the details about how to install, configure and use the component. Regarding the workflows, it has described the implemented workflow templates and phases and how they can be combined to define different simulation workflows. Apart from that we have also presented how to extend the current system to include more workflow templates, phases and simulation software.

The components, workflow templates and phases implemented until now, provide the functionalities which have been identified until this point of the project. They are ready to be integrated with the use case in WP7. We have also foreseen some improvements that can be performed during WP7 which will be included in the components Github repositories in the next months. For instance, the AML Tool is a desktop application, but it could be integrated to HPC simulation service. It will unify the user interface and it will avoid the user installing software in their computers. Another foreseen improvement is the installation of PAM-OPT, which is a tool particularly performant for optimization purposes, this implementation could include external software under optimization loops, improving the capacities of the CAELESTIS Ecosystem.

ABBREVIATIONS

AFP Automated Fiber Placement

AML/AutomationML Automation Markup Language

API Application Programming Interface

CPU Central Processing Unit

DET Distributed Engineering Teams

DoE Design of Experiments

FTP File Transfer Protocol

GUI Graphical User Interface

HPC High-Performance Computing

HTP Hybrid Twin Platform

HTTP Hypertext Transfer Protocol

ISE Interoperable Simulation Ecosystem

MPI Message Passing Interface

POC Proof of Concept

REST Representational State Transfer

URL Uniform Resource Locator

WP Work Package

YAML- Yet Another Markup Language

REFERENCES

[1] Rainer Drath, "*AutomationML, A Practical Guide"*, (2021), https://www.automationml.org

[2] The Django Framework Web site, https://www.djangoproject.com/

[3] Postgre SQL Database Web Site, https://www.postgresql.org/

[4] NGIX Server Web Site, https://nginx.org/

[5] Gunicorn Python WSGI HTTP Server Web Site, https://gunicorn.org/

[6] Read The Docs Website, https://about.readthedocs.com/

[7] Badia, Rosa M., et al. "Pycompss as an instrument for translational computer science." Computing in Science & Engineering 24.2 (2022): 79-84.