TrustwOrthy model-awaRE Analytics Data platfORm

RIA - Grant No. 688797

TOREADOR Platform Validation for the Application Energy Production Data Analysis Pilot

Deliverable D10.4.

Project Title	TOREADOR – TrustwOrthy model-awaRE Analytics Data platfORm
Grant Number	688797
Project Type	RIA

Title of Deliverable	TOREADOR platform validation for the Application Energy Production Data Analysis pilot
Subtitle of Deliverable	
Deliverable Number	D10.4
Dissemination Level	Public
Internal Rev. No.	1.0
Contractual Delivery Date	December 31, 2018
Actual Delivery Date	January 10, 2019
Contributing partners	LIGHT, CITY
Editor(s)	Michal Poszumski (LIGHT)
Author(s)	Michal Poszumski, Menelaos Ioannidis (LIGHT)
Reviewer(s)	TAIGER, JOT

Executive Summary

The Platform Validation pilots are the practical evaluation of the capabilities of the TOREADOR methodology and platform implementation, through the lenses of industrial use-cases. D10.4 covers LIGHT use-case for the optimisation of the energy production of the solar and battery assets.

Main focus of the document is to present why the TOREADOR brings value to the existing and future LIGHT system solutions. It mainly compares how system is distributing produced energy, measured by means on MTBF algorithm (Mean Time Between Failures), where it highlights failure of sub-optimal charging pattern.

This document is a companion to the pilot itself, which shall be presented as a video demo file, as well as during the final review itself.

Contents

Chapter 1

Introduction

Application Energy Production Data Analysis Pilot is about providing detailed results of performance patterns for specific solar and battery hardware solutions in the residential sector. Installed solar system provides cheap and clean energy which end users is expecting to benefit from, however during the lack of presence at premises during midday the abundance of this energy is mostly exported to the grid. In order to address this, LIGHT has installed battery systems across the portfolio of its residential clients to allow them to use cheap and clean energy after there is not enough solar irradiation.
Battery charging policies and control system are subject to suboptimal operations cause by multiple factors, like: fast moving clouds, not optimised firmware, fast changing load profiles. These periods are classified by LIGHT as “failures”, to assess impact of specific system setup configurations MTBF (Mean Time Between Failure) algorithm is used.
Systems with higher MTBF are more likely to have better financial performance as then the other units. Therefore, any further engineering or business decision processes are directly affected by the results of TOREADOR methodology and toolset.
This deliverable describes how data from the LIGHT pilot is applied on the TOREADOR methodology and services. Firstly Section 2 covers the exact use-case for data source and nature of MTBF. Data ingest and execution is represented in Section 3, while Section 4 explains how LIGHT pilot is utilising TOREADOR code-based approach and its parallelisation capabilities.

Each TOREADOR pilot has its own specificities: LIGHT pilot is the main implementation of code-based approach which allows end user with utilisation of the existing analytical tools.

Chapter 2

Use Case

Our database includes two main categories of data points, electrical **power** and **energy**, tracked and registered for every installation in daily, weekly, monthly, annually and lifetime basis. More specifically, for the power points we developed a live flow chart which displays in real time, Figure 1:

- the condition of the generated power from the panels,
- the import/export power from and to the grid respectively,
- the consumption power from house's loads and finally battery's power circulation into the whole system.

This dynamic power flow is displayed below with the relevant explanation per icons:

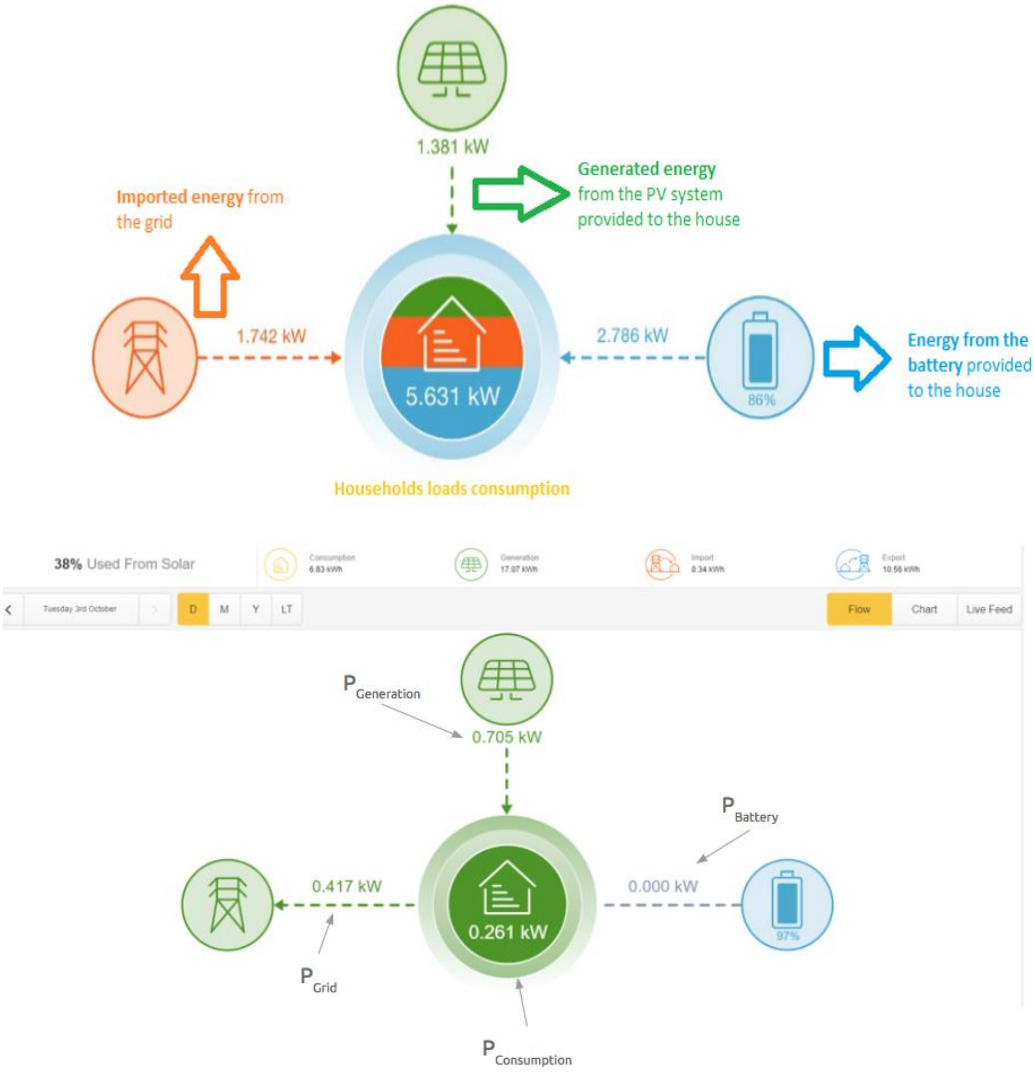

Figure 1: Dynamic power flow for the different key components and Power Values from a smart home installation

LIGHT's main target by developing the smart home solution is to decrease the imported energy from the grid, which is much more expensive than the fixed priced solar generated energy from the panels, installed on every smart home's roof. So, both the solar generated power and the battery power data points play the most significant role in our income. When we monitor failures from the inverter equipment, which is directly linked

with the solar panels and the battery, it means that our revenue is significantly influenced.

Therefore, two scenarios will be exploited and will be included into the MTBF model, which are the below:

- ***Scenario 1: Battery charging failure***

The energy storage equipment, a residential battery, plays an important role in LIGHT's smart home project. This automatically means that the battery's functionality is fundamental, and a series of various activities must be completed, to keep the battery in the highest efficiency rates, for both our customers and LIGHT's benefits. In more detail, for the battery to start charging from the excess solar generated power, there are two main principles that must be followed:

1. The generated power from the solar panels is higher than the consumed power from the house loads (Pgen > Pcons)

2. The battery's state of charge (SOC) is below 100% , which means that the battery is not fully charged.

3. Summaries on the Figure 2.

$$P_{Consumption} < P_{Generation}$$

$$Battery\ Level\ < 100\%$$

$$P_{Battery} > 0$$

Figure 2: Condition of Power balance in the Solar+Battery system to reflects healthy charging pattern.

Both principles mentioned above are monitored in our live system and are supervised from us via the inverter which gives the relevant commands for this power flow circulation to be completed successfully. In case that everything mentioned previously is followed accordingly, under these two main rules, and we don't see the battery charging (Pbat<0) then automatically an alert is flagged which means that we have a failure in our system (for various reasons), so this is where the MTBF model can assist us to reduce this kind of situations. Please see below a simulation of the MTBF for this scenario:

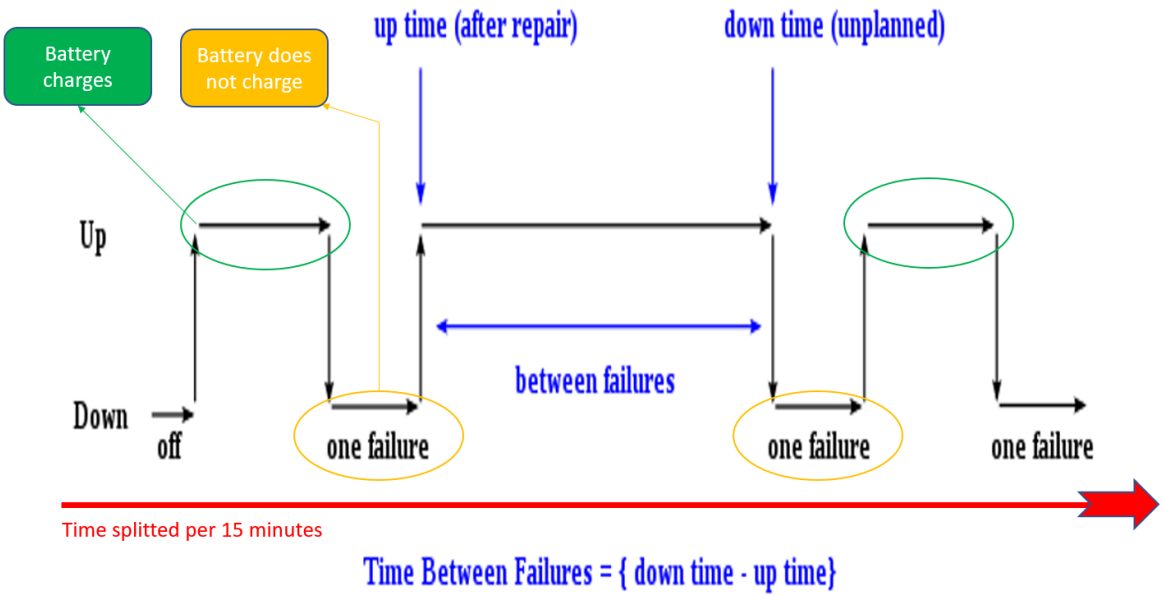

Figure 3: MTBF for the battery charging failure scenario

- ***Scenario 2: No generated power from the solar panels***

The key component of the smart home project is the solar panels on every smart home's roof. The ultimate target of the solar power, generated from the panels, is to cover most of the household needs and this power flow is controlled and monitored by the inverter, as in the scenario 1. LIGHT's income is totally based on the generated power from the panels towards the household, under a fixed price per kW, so any failures coming from this system influence our revenue. Our monitoring system has the ability to track the activity of the solar power value, during daytime when it will available, so if we see this value to be zero (Pgen=0) at 12.00 o'clock every day for one installation, this automatically means that there is fault that has to be resolved as soon as possible. Please see below the relevant MTBF model for this scenario:

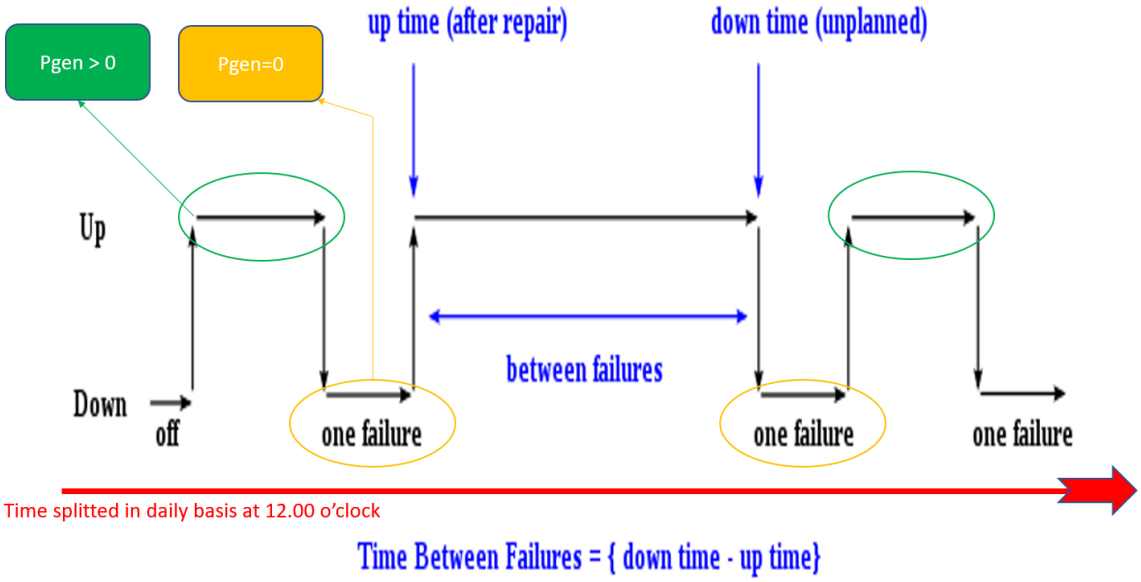

Figure 4: MTBF for the no generated power from solar panels scenario

Scenarios Results

By using the MTBF methodology for the two scenarios for several installations, we could produce a series of results using the MTFB formula, as it's displayed below:

$$\text{MTBF} = \frac{\sum(\text{start of downtime} - \text{start of uptime})}{\text{number of failures}}.$$

Figure 5: Summary of the MTBF formula, to be reflected in minutes)

When MTBF algorithm, written in Python is applied on the 4 data sensors it produces following example result for one house:

```
Fetching data...
Analysing data...
[Battery not charging] Pcons: {'time': '2017-06-04T09:29:00Z', 'value': 0.115}, Pbatt: {'time': '2017-06-04T09:29:00Z', 'value': -0.011}, Pdc: {'time': '2017-06-04T09:29:00Z', 'value': 0.732}
[Battery not charging] Pcons: {'time': '2017-06-04T17:15:00Z', 'value': 0.302}, Pbatt: {'time': '2017-06-04T17:15:00Z', 'value': -0.099}, Pdc: {'time': '2017-06-04T17:15:00Z', 'value': 0.721}
[Battery not charging] Pcons: {'time': '2017-06-04T17:22:00Z', 'value': 0.357}, Pbatt: {'time': '2017-06-04T17:22:00Z', 'value': -1.579}, Pdc: {'time': '2017-06-04T17:22:00Z', 'value': 0.495}
[Battery not charging] Pcons: {'time': '2017-06-04T17:31:00Z', 'value': 0.329}, Pbatt: {'time': '2017-06-04T17:31:00Z', 'value': -0.171}, Pdc: {'time': '2017-06-04T17:31:00Z', 'value': 0.364}
For Gateway UTXb98803 between 2017-06-04 00:00:00 and 2017-06-04 23:59:59
- MTBF for Battery not charging: 119.75 minutes
- MTBF for System not producing Solar: 0.0 minutes
```

Battery not charging:
- total minutes between failures: 479
- total number of failures: 4
- MTBF: 479/4 = 119.75

Figure 6: Typical house representing four events where battery wasn't working in an optimal way.

Chapter 3

Pilot Architecture

Application Energy Production Data Analysis pilot consists of LIGHT technology stack, where each house is equipped with solar, battery, power meters and datalogging controller with the Internet upload capability. Each minute data is sent and stored on the dedicated IoT broker and time-series data base.

Dataset is available via RESTful API, to which calls are made straight from the MTBF algorithm, where parameters for period (START *timestamp* and END *timestamp*) and specific datalogger(s) IDs.

In return user is presented with the data files which either can be injected back to LIGHT backend systems (also RESTful endpoints) for business decision making. Overall architecture is reflected on Figure 7.

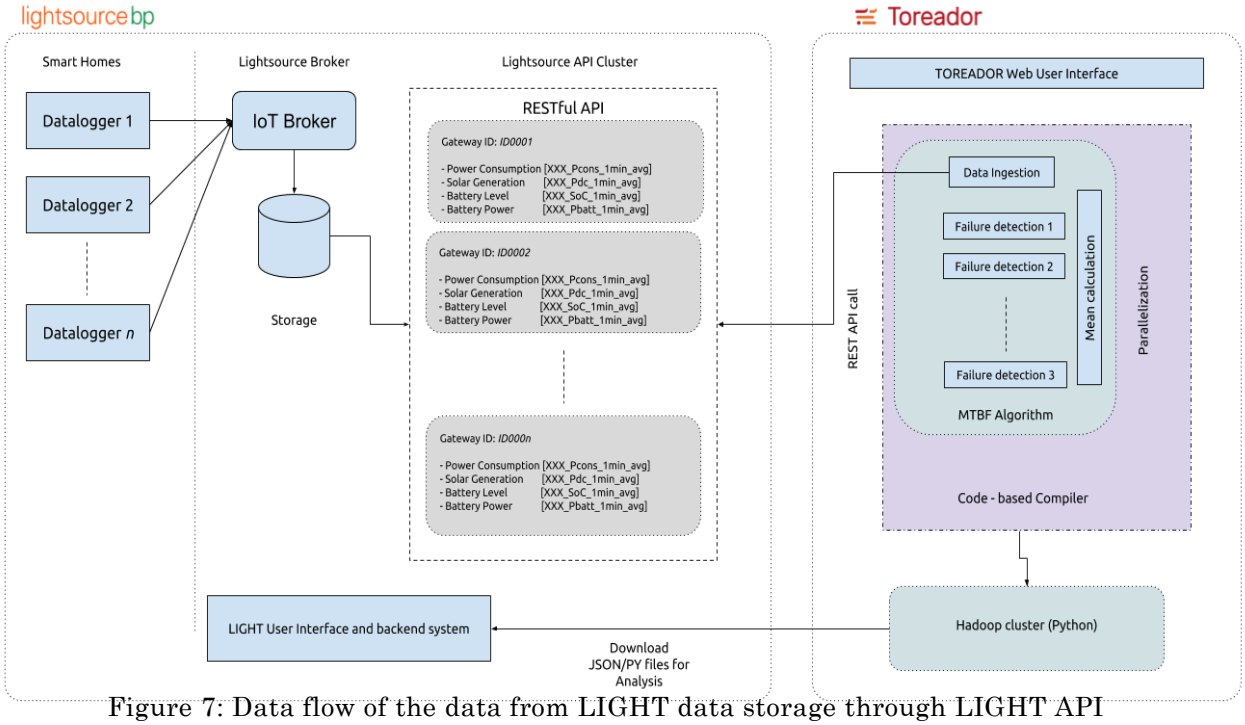

Figure 7: Data flow of the data from LIGHT data storage through LIGHT API endpoint for specific house(s).

Chapter 4

TOREADOR methodology

Application Energy Production Data Analysis pilot is utilising Code-Base approach of the TOREADOR. This part of the deliverable cover parallelisation of the MTBF algorithm and its flow through the TOREADOR User Interface (Figure 8).

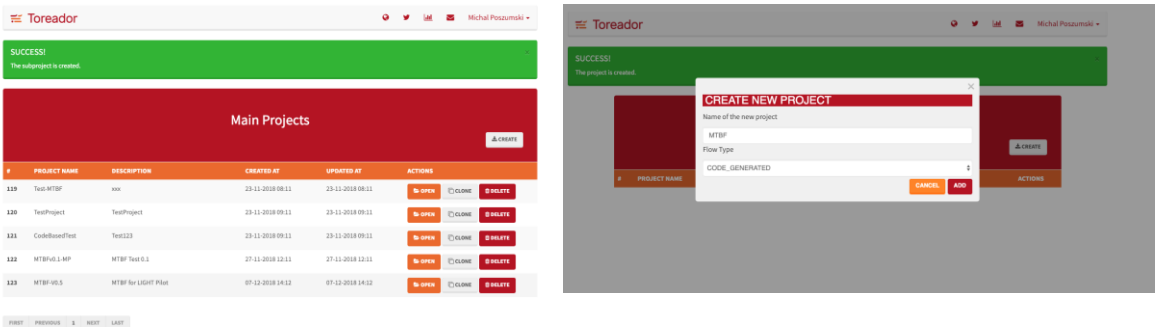

Figure 8: Code based approach selection on the TOREADOR User Interface.

4.1 The Code-Based approach: recap

The Code-Based Approach developed within the Toreador project takes as input a sequential code, annotated with parallelization primitives which have been defined within the very same project, and provides a distributed version of it. The distributed version consists of a set of instantiations of Parallel Skeletons, filled by following a Parallel Computational Pattern, and selected according to the primitives used to annotate the original code and to the information derived from the Declarative Model (see Deliverables D1.2, D2.1, and D3.3)
The main objective of the Code-Based Approach is to express the parallel computation of a coded algorithm in terms of parallel primitives and distribute it among computational nodes hosted by different platforms/technologies in multi-platform Big Data and Cloud environments. using State of the Art orchestrators. The approach requires that the user annotates her source code with a set of Parallel Primitives, which are then examined by the compiler. Such primitives are modeled

following a set of Parallel Patterns and guide a Skeleton Compiler in the selection of the best suiting Skeleton to fill to obtain the desired parallelization. Within the Toreador approach, Parallel Primitives are constructs with a specific meaning, which are used to augment a standard and widespread language, i.e. Python, to guide the compiler in the parallelization.
Skeletons' filling is obtained by manipulating the Abstract Syntax Tree of the source code, following a set of transformation rules which are determined by the adopted Parallel Primitives and corresponding Patterns. Deployment Skeletons are also produced, according to the target platform selected by the user.
The Code-Based Approach can be exploited in two different moments of the parallelization process: at the very beginning, if the user wants to start with an only-code approach, thus she requires support in the application of the proposed parallelization primitives; during a tuning phase, when the code has already been produced but optimization is still required.
Figure 9 provides an overview of the complete workflow with the three main phases composing it.

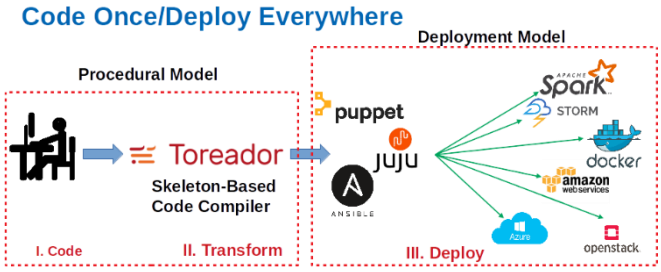

Figure 9: Interface of the Web Compiler

1. Code: in this phase the user, which is supposed to be an expert programmer and to own a good knowledge of the sequential algorithm to be parallelized, annotates the original codes with the provided Parallel Primitives.
2. Transform: During this phase a Skeleton-Based Code Compiler (Source to source Transformer) transforms the sequential code augmented with primitives into parallel versions specific to different platforms/technologies, according to a 3-steps sub-workflow.
    - Parallel Pattern Selection: selection of the Parallel Paradigm (based on Declarative Model and utilized primitives)
    - Incarnation of agnostic Skeletons: transformation of the algorithm coded utilizing the aforementioned primitives in

a parallel agnostic version incarnating predefined code Skeletons
- Production of technology dependent Skeletons: specialization of the agnostic Skeletons in multiple versions of the code to be executed, but specific to the different target platforms/technologies

3. Deployment: Production of Deployment Scripts with parameters inferred from the Declarative Model (e.g. number of nodes, node's characteristics, memory).

4.2 Definition - Application of the Code-Based approach to the MTBF algorithm

Since the MTBF algorithm gathers data from different sensors and then analyses their operational data to assess their possible failures, it seems a good candidate for parallelization. In particular, the assessment of failures for each sensor is obtained through the execution of the fault detection function, which is called iteratively over the rows of a data frame, as it can be seen in the Code Excerpt 1.

```
for index, row in data.iterrows():
    # Append dates depending on state of battery and solar
    fault = fault_detection(row.time, row.pcons, row.pbatt, row.pdc, row.soc)
    if fault[0] != 0:
      mtbf_battery_dates.append(fault[0])
    if fault[1] != 0:
      mtbf_solar_dates.append(fault[1])
```

Code 1: Fault calculation for each sensor

The parallelization approach applied to the MTBF algorithm is composed of two different levels

1. A first level parallelization, applied to each of the specific gateways to analyse. This section of code has been restructured using the ***data_parallel_region*** primitive, as described in the following section. The elements which are analysed in this first level of parallelization are gateways: for each of them failures of solar-based and battery-based powering are calculated, and the result will be elaborated in the second level.
2. A second level of parallelization, consisting in a data parallel section where the ***producer_consumer*** primitive is used to calculate failure, and then two different reductions can be executed at the same time by applying the ***bag_of_task*** primitive. This specific combination of primitives has been chosen to minimize

code modifications and to obtain an implementation where consumers can be reused in the reduction phase.

Code Excerpt 2 and 3 show the complete transformation operated on the MTBF code, by using parallelization primitives to annotate specific portions of the original program.

```
def calculate_mtbf(data):
    #initialise dates
    mtbf_battery_dates = pd.DataFrame(data={'Timestamp':[]})
    mtbf_solar_dates = pd.DataFrame(data={'Timestamp':[]})

    # (aggregate) all measurement points and loop through
    print("Analysing data...")
    for index, row in data.iterrows():
        # Append dates depending on state of battery and solar
        mtbf_battery_dates, mtbf_solar_dates = fault_detection(row, mtbf_battery_dates, mtbf_solar_dates)

    # sort the dates
    if mtbf_battery_dates.shape[0]:
        mtbf_battery_dates.sort_values(by = ['Timestamp'], inplace=True)
    if mtbf_solar_dates.shape[0]:
        mtbf_solar_dates.sort_values(by = ['Timestamp'], inplace=True)

    # Calculations for Battery and Solar
    mtbf_battery_minutes_between_failures, mtbf_battery_failures  =
calculate_minutes_and_failures(mtbf_battery_dates)
    mtbf_solar_minutes_between_failures, mtbf_solar_failures  = calculate_minutes_and_failures(mtbf_solar_dates)

    # Round down mtbf_battery minutes
    mtbf_battery = int(mtbf_battery_minutes_between_failures / (mtbf_battery_failures if mtbf_battery_failures !=
0 else 1))
    mtbf_solar = int(mtbf_solar_minutes_between_failures / (mtbf_solar_failures if mtbf_solar_failures != 0 else
1))

    print("For Battery: minutes between failures {} and number of failures {}".format(
        mtbf_battery_minutes_between_failures, mtbf_battery_failures))
    print("For Solar Power: minutes between failures {} and number of failures {}".format(
        mtbf_solar_minutes_between_failures, mtbf_solar_failures))

    single_json = json_calculation(mtbf_battery_minutes_between_failures, mtbf_battery_failures,
                                   mtbf_solar_minutes_between_failures, mtbf_solar_failures)

    return mtbf_battery, mtbf_solar, single_json

for gateway in gateway_names:
    # API Calls
    # 1st, 2 API Calls, Get gateway_id, Get sensors_id's

    sensor_id_Pcons, sensor_id_Pbatt, sensor_id_Pdc, sensor_id_SoC = retrieve_sensors(gateway)
    # 2-6, 4 API Calls, Get Data for each sensor_id
    dataPcons = retrieve_sensor_data(date_start, date_end, sensor_id_Pcons)
    dataPbatt = retrieve_sensor_data(date_start, date_end, sensor_id_Pbatt)
    dataPdc = retrieve_sensor_data(date_start, date_end, sensor_id_Pdc)
    dataSoC = retrieve_sensor_data(date_start, date_end, sensor_id_SoC)

    # Renaming and Merging Dataframes so we can iterate them later
    # If we have different time they won't be merged
    dataPcons = pd.DataFrame({'time':dataPcons['time'], 'pcons':dataPcons['val']})
    dataPbatt = pd.DataFrame({'time':dataPbatt['time'], 'pbatt':dataPbatt['val']})
    dataPdc = pd.DataFrame({'time':dataPdc['time'], 'pdc':dataPdc['val']})
    dataSoC = pd.DataFrame({'time':dataSoC['time'], 'soc':dataSoC['val']})
    data = pd.merge(dataPcons, dataPbatt)
    data = pd.merge(data, dataPdc)
    data = pd.merge(data, dataSoC)

    # Calculate mtbf
    mtbf_battery, mtbf_solar, single_json = calculate_mtbf(data)
    msg = (
        "For Gateway {} between {} and {}\n"
        "- MTBF for Battery not charging: {} minutes\n"
        "- MTBF for System not producing Solar: {} minutes"
    ).format(gateway, date_start, date_end, mtbf_battery, mtbf_solar)
    print(msg)
    json_file.append(single_json)
```

Code 2: MTBF code before the annotation

```
def calculate_mtbf(gateways):
   # API Calls
   # 1st, 2 API Calls, Get gateway_id, Get sensors_id's
   sensor_id_Pcons, sensor_id_Pbatt, sensor_id_Pdc, sensor_id_SoC = retrieve_sensors(gateway)
   # 2-6, 4 API Calls, Get Data for each sensor_id
   dataPcons = retrieve_sensor_data(date_start, date_end, sensor_id_Pcons)
   dataPbatt = retrieve_sensor_data(date_start, date_end, sensor_id_Pbatt)
   dataPdc = retrieve_sensor_data(date_start, date_end, sensor_id_Pdc)
   dataSoC = retrieve_sensor_data(date_start, date_end, sensor_id_SoC)

   # Renaming and Merging Dataframes so we can iterate them later
   # If we have different time they won't be merged
   dataPcons = pd.DataFrame({'time':dataPcons['time'], 'pcons':dataPcons['val']})
   dataPbatt = pd.DataFrame({'time':dataPbatt['time'], 'pbatt':dataPbatt['val']})
   dataPdc = pd.DataFrame({'time':dataPdc['time'], 'pdc':dataPdc['val']})
   dataSoC = pd.DataFrame({'time':dataSoC['time'], 'soc':dataSoC['val']})
   data = pd.merge(dataPcons, dataPbatt)
   data = pd.merge(data, dataPdc)
   data = pd.merge(data, dataSoC)

   #initialise dates
   mtbf_battery_dates = pd.DataFrame(data={'Timestamp':[]})
   mtbf_solar_dates = pd.DataFrame(data={'Timestamp':[]})

   # (aggregate) all measurement points and loop through
   print("Analysing data...")
    for index, row in data.iterrows():
       # Append dates depending on state of battery and solar
       failure_dates = fault_detection(row, mtbf_battery_dates, mtbf_solar_dates)

   failure_dates = producer_consumer(data.iterrows(), fault_detection, mtbf_battery_dates, mtbf_solar_dates)
   batt_dates = pd.DataFrame(data={'Timestamp':[]}). solar_dates = pd.DataFrame(data={'Timestamp':[]})
   for el in failure_dates:
       if not el[0].empty:
           batt_dates = batt_dates.append({'Timestamp': el[0].Timestamp.values[0]} , ignore_index=True)
       if not el[1].empty:
           solar_dates = solar_dates.append({'Timestamp': el[1].Timestamp.values[0]} , ignore_index=True)

   mtbf_battery_dates = pd.DataFrame(batt_dates)
   print(mtbf_battery_dates.head())
   mtbf_solar_dates = pd.DataFrame(solar_dates)

   # sort the dates
   if mtbf_battery_dates.shape[0]:
       mtbf_battery_dates.sort_values(by = ['Timestamp'], inplace=True)
   if mtbf_solar_dates.shape[0]:
       mtbf_solar_dates.sort_values(by = ['Timestamp'], inplace=True)

   # Calculations for Battery and Solar
   minutes,failures =
bag_of_task([mtbf_battery_dates,mtbf_solar_dates],[calculate_minutes_and_failures,calculate_minutes_and_failures])

   # Round down mtbf_battery minutes
   mtbf_battery = int(mtbf_battery_minutes_between_failures / (mtbf_battery_failures if mtbf_battery_failures != 0 else
1))
   mtbf_solar = int(mtbf_solar_minutes_between_failures / (mtbf_solar_failures if mtbf_solar_failures != 0 else 1))

   print("For Battery: minutes between failures {} and number of failures {}".format(
       mtbf_battery_minutes_between_failures, mtbf_battery_failures))
   print("For Solar Power: minutes between failures {} and number of failures {}".format(
       mtbf_solar_minutes_between_failures, mtbf_solar_failures))

   single_json = json_calculation(mtbf_battery_minutes_between_failures, mtbf_battery_failures,
                                  mtbf_solar_minutes_between_failures, mtbf_solar_failures)
     return mtbf_battery, mtbf_solar, single_json
mtbf_battery, mtbf_solar, single_json
 = data_parallel_region(gateway_names, calculate_mtbf)
```

Code 3: MTBF code after the annotation

4.3 Parallelization and deployment using the *Map-Reduce* Parallel Pattern

By applying the data_parallel_region primitive, it is indeed possible to simplify the code and prepare it for parallelisation via the Toreador Code-Based approach, by exploiting an online web-compiler. In particular, in Code Excerpt 4 the application of the primitive is shown.

```
def data_parallel_region(distr, func, *repl):
  return [func(x, *repl) for x in distr]

fault = toreador.data_parallel_region(data, fault_detection)
```

Code 4: Definition of the data_parallel_region primitive and its application

The primitive simply substitutes the for loop in Code box 1 (the definition has been reported as reference, the complete list of available parallelization primitives has been reported in deliverable D2.2). When the modified source code is presented to the web compiler, it is analysed and transformed, according to the specific Parallel Pattern which has been chosen for the parallelization. As it can be seen from Figure 10, the code can be pasted in the central area, while the Pattern and the Figure 10.

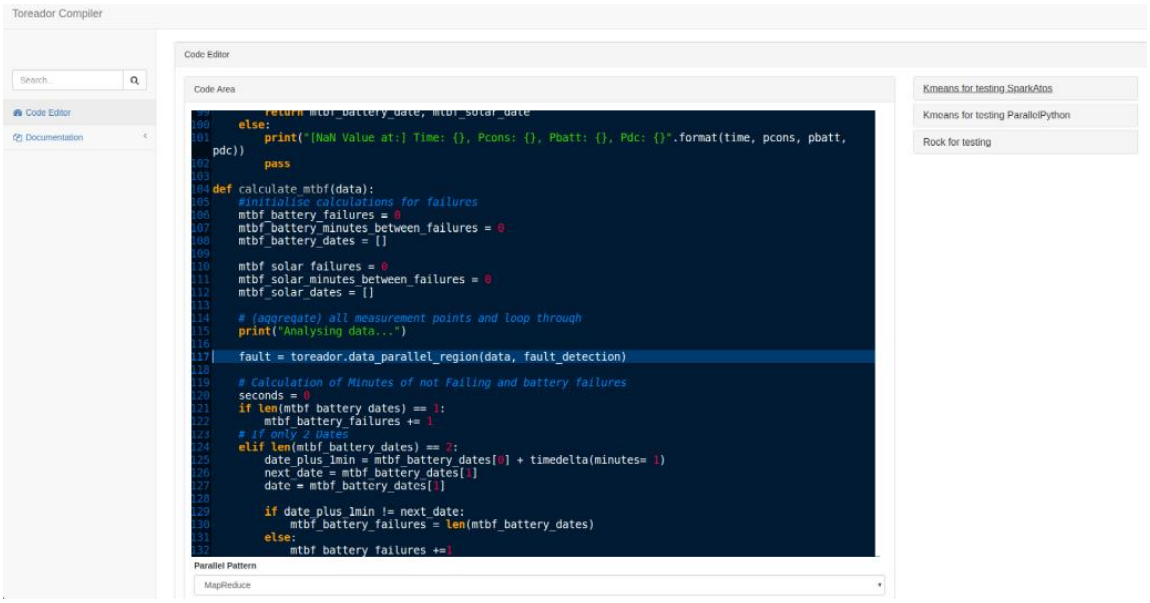

Figure 10: Interface of the Web Compiler

Deployment Platform can be selected from the menu just below the code area, as it can be seen in Figure 11.

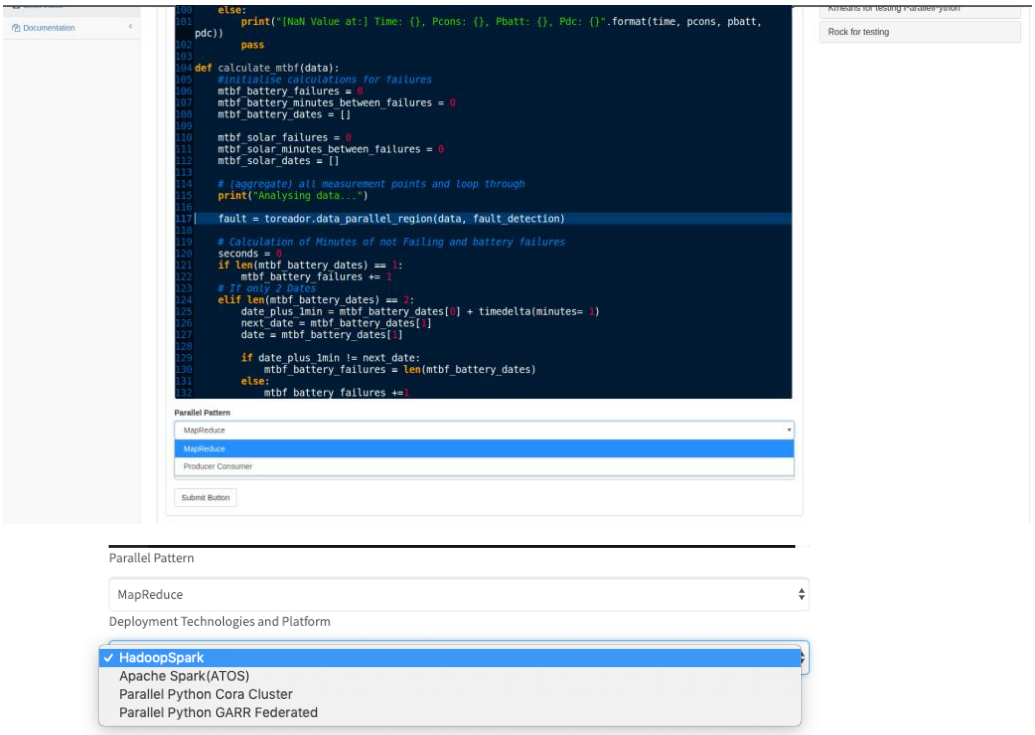

Figure 11. Web Compiler: Selection of the Parallel Pattern

If the ***Map Reduce*** Parallel Pattern is selected, and the "Submit" button is clicked, then three different files are produced:

- A Main.py file in charge of creating the files needed by Map Reduce to run and to set-up the environment: input data files, output directories and so on.
- A Mapper.py file that will contain the Map code
- A Reducer.py file that will contain the Reducer code.

If **Hadoop cluster** using **Python** is selected as *Deployment Platform*, then communications between mappers and reduces are performed via STDIN and STDOUT and the Mapper and Reducer code is produced, as reported in Code Boxes 5 and 6.

```
for row in sys.stdin:
   row = row.strip()
   row=row.split('\t');
    fault = fault_detection(row[0], row[1], row[2], row[3], row[4])
     print '%s\t%s' % (fault[0] fault[1])
```

Code SEQ Code 5: The Mapper File

```
for fault in sys.stdin:
   # remove leading and trailing whitespace
   fault = fault.strip()
   fault = fault.split('\t')
if fault[0] != 0:
   print 'mtbf_battery_dates\t%s' % (fault[0])
      if fault[1] != 0:

   print 'mtbf_solar_dates\t%s' % (fault[1])
```

Code SEQ Code 6: The Reducer File

The distribution of the data to the Mappers is part of the responsibilities of the Main file, which elaborates the input and translates it to a suitable file for the Mapper to read. Also, the Main file collects the single results printed by the Reducer file and produces the output requested by the original algorithm.
The Compiler allows the user to download and inspect the produced code, and also to visualize the output of the execution, as shown in Figure 12:

Figure 12: Downloadable Python files and Output box

4.4 Two-level Parallelization with Producer-Consumer and Bag-of-Tasks Patterns, and Deployment on Spark and Docker

The previous example focuses on a Hadoop cluster as a target deployment platform.
However, the approach is flexible enough to allow the users to deploy their application on any other supported target platform, without significant changes.
Indeed, the web compiler supports, among other platforms, **Apache Spark** and **Docker** for the deployment and execution of the parallelized code. In particular, for our experiments we have exploited the Docker environment provided by **ATOS**.
The steps to follow are exactly the same as in the previous example:

1) The code annotated with the Parallel Primitives is copy-pasted in the compiler window
2) The desired Pattern is chosen. For a Spark or Docker deployment options, different Parallel Patterns such as ***Producer Consumer*** and ***Bag of Tasks*** can be selected, in addition to the *Map Reduce* one.
3) The user selects the target platform and submits the selections: the Code-compiler working in background produces the execution and deployment files, by filling suitable Skeletons.

As in the previous case, the creation of the files and the final deployment and execution are completely transparent to the user.

If the *Producer Consumer* Pattern is selected, after clicking the "Submit" button three different files are produced:

- A Main.py file in charge of creating the files needed by the Spark/Docker target to run and execute the code correctly
- A Producer.py file that will contain the Producer code, implementing a queue for the distribution of the data
- A Consumer.py file that will contain the Consumer code, which will be executed in parallel.

The management of the different nodes is handled differently according to the target deployment environment:
In **Spark**, the environment takes care of the nodes' management, and the compiler only produces one deployment Script. However, by selecting Spark only, it would be impossible to exploit the two different parallelization levels. If we want to exploit the two parallelization levels, the first one must be implemented through Docker, by instantiating a

Spark cluster with its Master and Slave nodes for each gateway. A deployment file for each of the clusters would be produced.

With **Docker**, the Main file will contain the instructions required to correctly instantiate the execution nodes and build a shared volume, while each of the Docker containers will receive a Python file containing the micro-functions to execute.
Regarding the Docker implementation, two different strategies are currently being considered:

- A centralized approach, in which the Docker containers are managed locally, within a self-contained platform which instantiates and manages them;
- A distributed approach, in which containers can run on remote devices, which are selected by a central offloading algorithm according to the execution schema. This second approach can be considered an instance of **Edge Computing** paradigm, where edge computing nodes are smartphones or other smart devices near to the data sources. In the LIGHT scenario, such devices can be represented by the gateway themselves, located in each household, and each gateway would compute only the data coming from the sources belonging to the household where it is located.

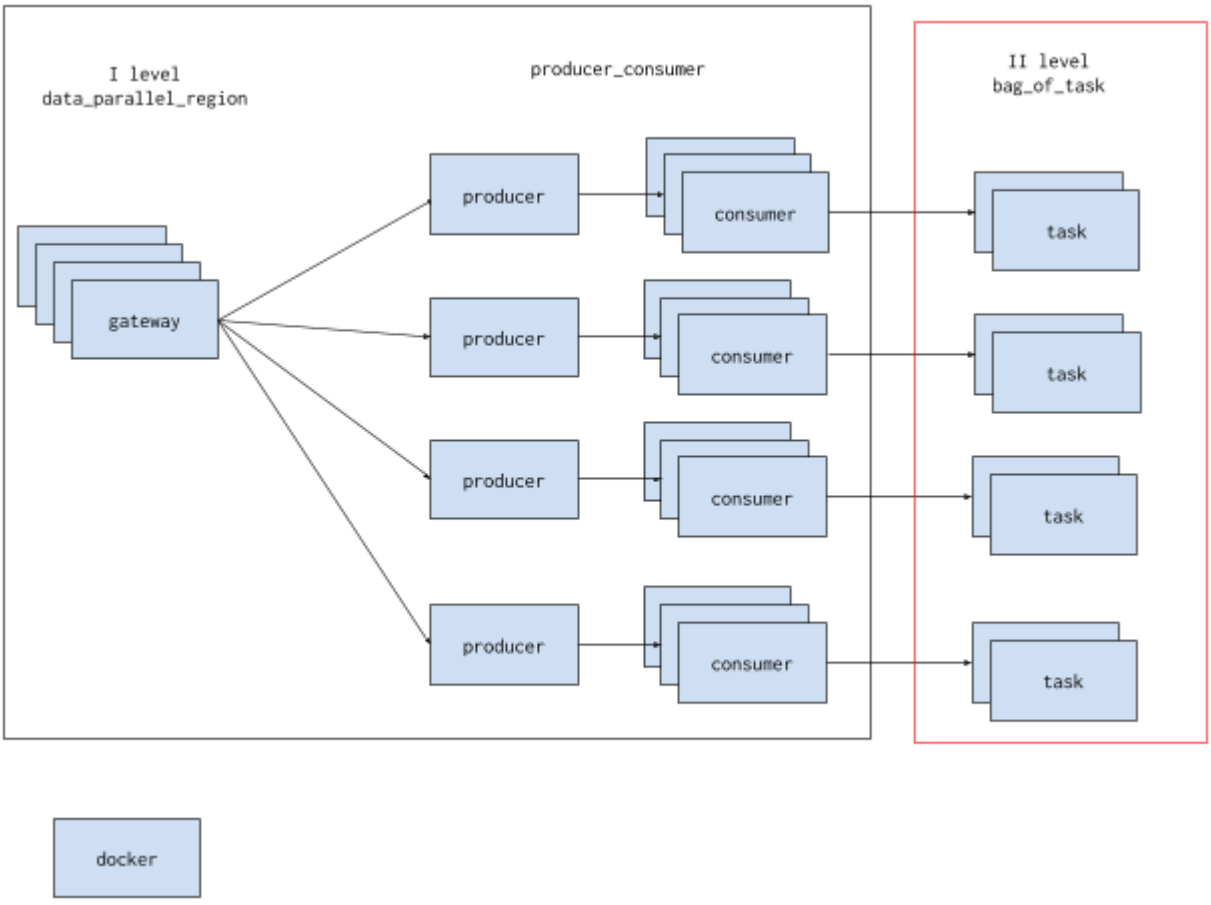

Figure 13: Docker Implementation with the two parallelization levels

All of the parallelization and deployment strategies reported until now, have been applied to a **Batch** processing of information.
However, data **Streaming** has also been considered as a possible input: as of now, solutions based on Apache Kafka are being considered and studied to manage stream analysis and overcome the limitations of batch elaborations. The basic idea is to create different Kafka topics for each of the gateways and to proceed with a Spark elaboration, thus reusing part of the parallelization strategies we have defined so far.

Required conversion process is deployed to create JSON format suitable for LIGHT User Interface.

Chapter 5

Results

As output of the Deliverable end user obtains JSON file or direct upload to LIGHT backend system which presents data on the chart or tables, based on this Engineering departments: Firmware creation, Control systems and Procurement departments are able to assess which particular product combinations are most profitable for the client or asset owner.

LIGH Pilot implementation example (Figure 14) across number of systems demonstrates that installation ID000005 generates highest MTBF factor, which is reflected also in the lowest number of suboptimal events. Sites like this are then used as a benchmark for business and equipment indicators.

gateway	battery__minutes	battery__failures	mtbf_battery
ID000001	18499	168	110.1
ID000002	18075	140	129.1
ID000003	17109	170	100.6
ID000004	18002	210	85.7
ID000005	18691	96	194.7
Mean of All	18075.2	156.8	124.1

Figure 14. Example of the MTBF results for 5 systems operated in 6 months of 2017.