MOdel based coNtrol framework for Site-wide OptimizatiON of data-intensive processes

D4.7–Final Multi-scale Model based Development Environment

Deliverable ID	**D4.7**
Deliverable Title	**Final Multi-scale Model based Development Environment**
Work Package	**WP4 – Cross-sectorial Data Lab**
Dissemination Level	**PUBLIC**
Version	**1.0**
Date	**01/08/2019**
Status	**Final**
Lead Editor	**CERTH**
Main Contributors	**TUK** **FIT** **LINKS** **CERTH**

Published by the MONSOON Consortium

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 723650.

Document History

Version	Date	Author(s)	Description
0.1	2019/04/10	ThanasisVafeiadis(CERTH)	First Draft with TOC
0.2	2019/05/25	Peter Bednar (TUK)	Added description of architecture and Apache Zeppelin and JupyterHub environments.
0.2	2019/06/26	Kostas Georgiadis (CERTH)	Contribution on Section 3 (Packaging, Relations with runtime container)
0.3	2019/07/05	ShreekanthaDevasya (FIT)	Updates on Section 3
0.4	2019/07/14	ThanasisVafeiadis (CERTH)	Addition on Section 4
0.5	2019/07/19	ThanasisVafeiadis (CERTH)	Introduction section added
0.6	2019/07/22	ThanasisVafeiadisKostas Georgiadis(CERTH)	Final version

Internal Review History

Version	Review Date	Reviewed by	Summary of comments
1.0	2019/07/30	Jose Antonio Jimenez (UNE)	Only minor editorial corrections suggested. No technical corrections.
1.0			

Table of Contents

1 Introduction

The present document is a deliverable of the "***Model based control framework for Site-wide OptimizatiON for data intensive processes***" - (MONSOON) project, funded by the European Commission's Directorate - General for Research and Innovation (D-G RTD) under Horizon 2020 Research and Innovation programme (H2020). The deliverable presents the final version of multi-scale model based development environment developed until M32 of the project.

The MONSOON project needs to develop a multi-scale development environment so as to support a variety of programming languages and tools that will cover the whole life cycle of the planning, implementation, and deployment of data analytics and predictive functions provided from WP5. *Task 4.3– Model based Development Environment*deals with this problem since the beginning of the project. The deliverable D4.7 - – *Final Multi-scale Model based Development Environment*provides the final architecture of the development environment of the MONSOON platform, tools for generation of models, predictive functions packaging and simulation and validation tools. This document is an update of the D4.6 – Initial Multi-scale Model Based Development Environment.

This report provides an update of the status of the architecture and tools mentioned before.The final version of theDevelopment environment is based on the combination of Apache Zeppelin and JupyterHub. Both notebook based tools provide similar interfaces and user experience, and both use various programming languages such as Python, Scala, or R. Also, an update on the pipeline of a predictive function inside a Docker container is provided. As for predictive functions, they use various file formats, such as JSON and CSV, as input and output and the output files are further stored in a database, namely, Kairosdb and Mongodb. Kairosdb stores the visualization data, while Mongodb stores also predictive function results.

The development approach in MONSOON project is iterative and incremental, including three prototyping cycles (ramp-up, period 1, period 2). The current document describes the situation of period 2.

2 Updates on architecture of the development environment

The main role of the development environment is to provide integrated tools and user interfaces for data scientists supporting whole life-cycle of predictive functions development. The development of predictive functions covers whole analytical process and consists of the phase of data understanding and pre-processing, modelling, validation and deployment. On one side, Development environment provides generic tools for software development (i.e. code editors, testing frameworks, source code repository, etc.) and on the other side, it provides tools specific for the data processing, machine learning and data visualization. The convenient, user-centric way how to combine these two perspectives is based on the usage of shared notebooks, i.e. multi-purpose documents which combine programming code of data processing scripts, textual documentation and graphical outputs of the visualization techniques.

The final version of the MONSOON Development environment is based on the combination of two notebook-based tools: Apache Zeppelin and JupyterHub. Both tools provide similar interfaces and user experience, and both have modular architecture which allow to create notebook document using various programming languages such as Python, Scala, or R) and running the code in various execution environments (local or distributed). MONSOON integrates Apache Zeppelin and HupyterHub to the common integrated environment, so data scientists can login to the tools using the same account, have the access to all data, and can easily share the results of data analysis between both tools. The following chapters provide more detailed description of both tools and how they are integrated with the rest of the MONSOON platform and Data Lab components.

2.1 Apache Zeppelin environment

Apache Zeppelin provides web-based development interface, where the data scientists can create multi-purpose notebooks. Notebooks can contain data processing scripts in various languages and environments (e.g. Python, Spark) and present the result of the analysis using various data visualization techniques. Internally the Zeppelin's architecture consists of web-based frontend, core backend and interpreters, which integrates various execution environment and external tools for data analysis. The most important interpreters for the MONSOON platform are Python interpreter and Apache Spark interpreter.

2.1.1 Pythoninterpreter

The Python interpreter allows to implement and run data analytics scripts in the Python language (version 2 or 3).MONSOON implementation is based on the Dockerised environment, i.e. the data scientist can specify a Docker image which will be executed by the interpreter in the Docker container dedicated for the specific user session per notebook (i.e. user can work with multiple notebooks running in the separated Docker containers). After the starting of the container, the backend core of the Apache Zeppelin is communicating with the Docker environment using the proprietary script or using the IPython interpreter protocol over the network. IPython interpreter is then responsible to execute chapters of the notebook in interactive way (i.e. user can online edit the code, which is directly executed, and the results are presented in the web frontend in the real-time). A depiction of Apache Zeppelin Python Environment is given in Figure 1.

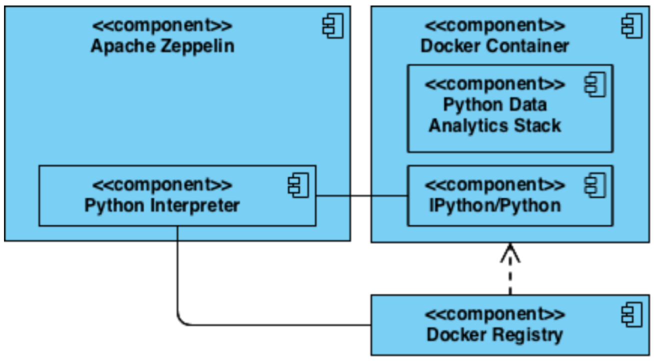

Figure 1–Apache Zeppelin Python interpreter.

It is possible to register multiple Docker images with the various configuration providing different data analysis tools such as stack based on the Numpy, Pandas and Scikit learn etc. Images are primary stored in the Data Lab function repository so the same software configuration can be shared for the development in the Data Lab environment or for the deployment on the site in the Runtime container in the operational environment. However, it is also possible to use external Docker image repositories with the configuration customized for the specific user or notebook.

Docker container is connected to the private network which is connected to the Distributed database in the Data Lab platform, so all data stored in the Data Lab environment are directly available in the Development environment. Data scientist is accessing data through the database client software library which is preconfigured in the Python environment. Data scientists can store data temporarily in the local container storage, or store data to the remote file system folder shared between all containers, so it is possible to directly share any kind of data artefacts generated during the data analysis process in different notebooks/environments.

2.1.2 Apache Sparkinterpreter (Optional)

Apache Spark is a parallel processing framework that enables users to run large-scale data analytics applications. Apache Spark provides API (Application Programming Interface) for batch and stream processing and SQL data access. Part of the Apache Spark are libraries of scalable Machine Learning (ML) and graph processing algorithms. For cluster management, Spark supports stand-alone installation or Apache YARN. For distributed storage, Spark supports various files systems and databases including HDFS.

Apache Spark is integrated with the Development environment using the Apache Zeppelin's Spark interpreter. Spark interpreter supports both stand-alone installation of the Apache Spark where the Spark job (packaged script code) is scheduled directly to the Spark Master node or YARN installation where the job is distributed by the YARN manager. The whole Spark cluster is dockerized and Spark interpreter is communicating with the Spark Master node or YARN through the private network. Storage is handled in the Spark using the distributed file system or shared NAS (Network Attached Storage) volume.

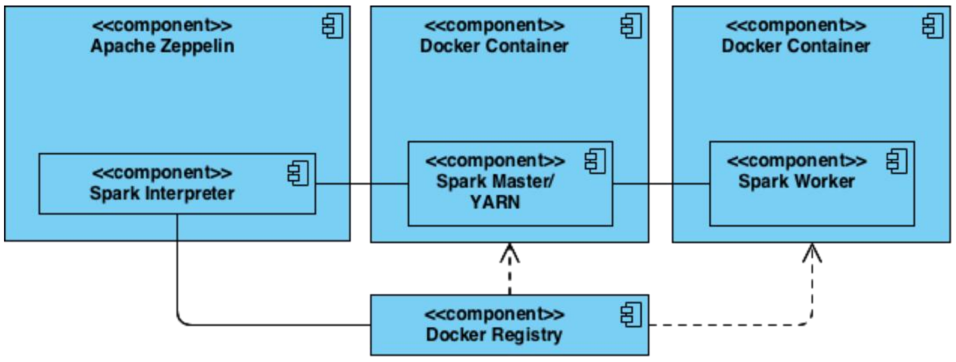

Figure 2 – Apache Zeppelin Spark interpreter.

2.2 Jupyter Hub environment

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more. Basic Jupyter Notebook installation support single user session only, but Jupyter Hub environment spawn multiple remote engines and support multi-user scenarios. User management of the Jupyter Hub is integrated with the MONSOON Data Lab user management infrastructure (based on LDAP), so data scientists can login using the same user account to all Development tools. Integration of the Jupyter notebooks is similar to the integration in Apache Zeppelin and it is based on the Dockerized Python environment. The same preconfigured images can be shared between the Apache Zeppelin and Jupyter Hub, the only disadvantage of the Jupyter Hub is that the user cannot dynamically change the image for the environment and all containers are based on the same configuration.

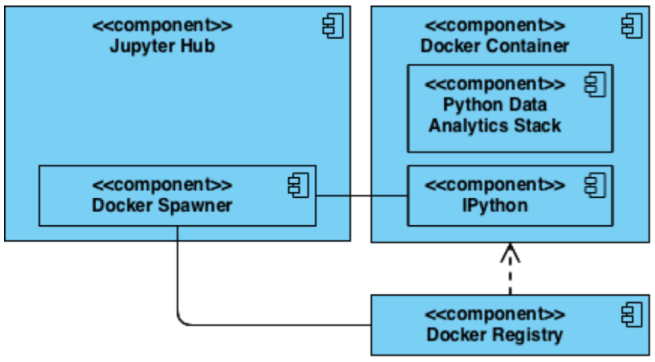

Figure 3 – Jupyter Hub environment.

The access to the data and storage is implemented in the same way like for the Apache Zeppelin, so the Jupyter Notebook has the access to the all data stored in the Data Lab and data scientist can store data locally or share them between multiple Jupyter notebooks or with the Apache Zeppelin environment.

3 Predictive function packaging and relation with the Runtime Container – Updates

3.1 Packaging

Predictive Functions are packaged in the development container as Docker images. The rationale behind the usage of Docker images to package Predictive Functions is described in the following (see the general docker architecture in Figure 4:

1. Package dependencies of prediction algorithms can be easily managed within a Docker container.
2. Easy rollback to previous models or predictive algorithms by using different versions of Docker containers.
3. Containers can run consistently on any server without modification.
4. Entire predictive function pipeline (including pre-processing and task specific evaluation) can be implemented within a Docker container.
5. Multiple containers can work together to achieve a bigger goal, hence, achieving component re-use.
6. Docker containers occupy less space and are very lightweight, thus there is no hardware performance delay.
7. Docker offers a wide variety of simple commands to build and run the containers, which is perfect during the developing phase.

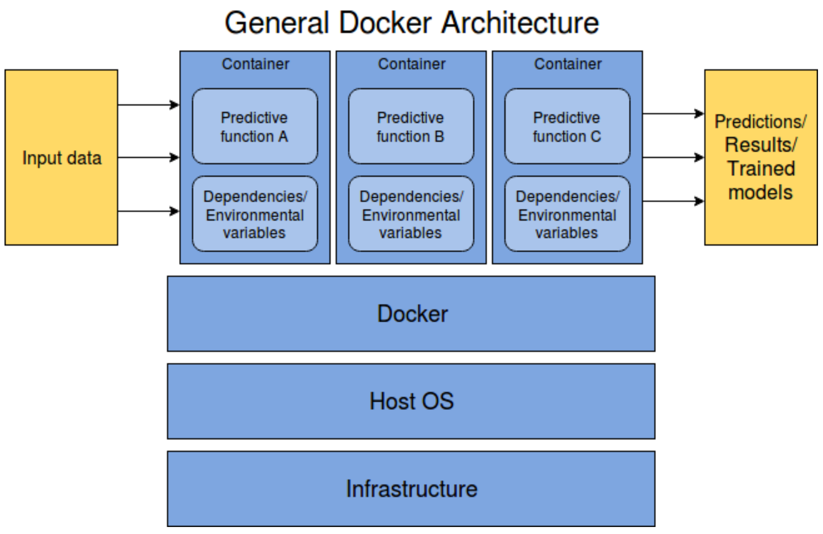
General Docker Architecture
Input data
Container
Predictive function A
Dependencies/ Environmental variables
Container
Predictive function B
Dependencies/ Environmental variables
Container
Predictive function C
Dependencies/ Environmental variables
Predictions/ Results/ Trained models
Docker
Host OS
Infrastructure

Figure 4 – General Docker architecture.

Figure 5 shows the pipeline of a predictive function inside a Docker container. The Docker image is created in the development container by a data scientist. The data scientist, trains one or more machine learning models, using the historical data available in Datalab and evaluates them for better accuracy. Once the model performs well, the data scientist decides to export it to the Runtime Container. This decision can also be automatically taken, based on accuracy measures, for example, whenever the accuracy is better than a certain threshold. This model is then stored in a Docker volume, which is a directory outside the Docker container filesystem, used for persisting data generated by the Docker containers.

The Docker image mainly contains a Prediction Algorithm that performs machine learning tasks, such as classification, regression or clustering, using pre-trained models. These algorithms expect the data to be in a developer-defined format. Moreover, the model has some predefined assumptions over the data, failing of which can end up in malfunctioning of the prediction tasks. Therefore, it is very important to validate and pre-process the incoming data before feeding them to the prediction algorithm. Hence, the Docker image also packages a pre-processing logic that validates, cleans and manipulates the incoming data. It also converts the data to a format required by the Predictive Function.

Another useful feature offered by Docker is the utilization of a parent image to build multiple subsequent images from it. More specifically, the Docker image of a predictive function, can be used as a parent to build other images. These derived images modify the usage of the parent image, for example run the predictive function for a specific period, or add a new feature to it, for example utilize cron to run the predictive function with a fixed frequency.

The Docker image, once created, will be stored in a Docker registry. This image will be deployed in the Runtime container for real-time usage. Even after a version of the Predictive Function is stored in Docker registry, new versions of the model are continuously trained with the new set of data arriving from the sites. Therefore, the model kept updated updating with time and new versions of Docker images are created for every updated model.

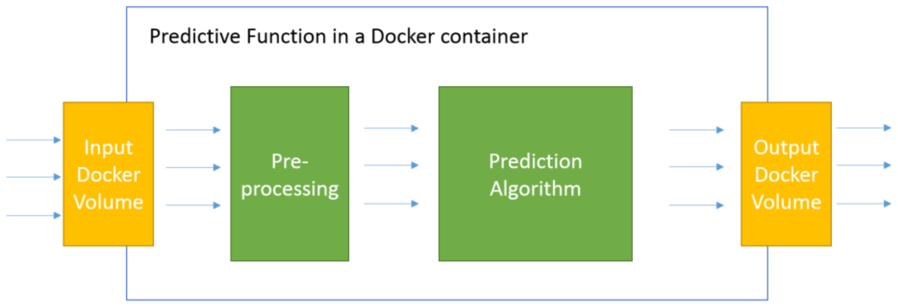

Figure 5 - Predictive Function inside a Docker container.

3.2 Relations with Runtime Container

The latest versions of the Predictive Functions, packaged as Docker images, are pulled from the Docker registry of the development container and are executed in the Runtime Container (see Figure 6 below).
In addition to the pre-processing logics of predictive functions, the data ingestion module (i.e. Apache NIFI) of the Runtime Container will provide initial set of pre-processing logic, which filters the data before they are fed to the predictive functions. These filters are used by multiple Predictive Functions and provide a uniform input format among multiple predictive functions, running in the Runtime Container. The behavior of filtering logics in the data ingestion module is governed by a configuration file exposed by the Predictive Functions. For example, the configuration file of a predictive function might contain the subset of attributes to be used for further processing.

The Predictive Functions use various file formats, such as JSON and CSV, as input and output. The Docker containers work with Docker volumes to store and share files with the host machine. The Predictive Functions share two volumes with the host machine in order to share input and output files. The ingestion module puts the input files in the shared location reserved for inputs (input/left orange block on the figure). The Predictive Function reads the files and uses it for further processing. The output of the Predictive Function is then stored in the shared location reserved for outputs (output/right orange block on the figure). The output files arefurther stored in a database, namely, Kairosdb and Mongodb. Kairosdb stores the visualization data. Mongodb stores predictive function results.

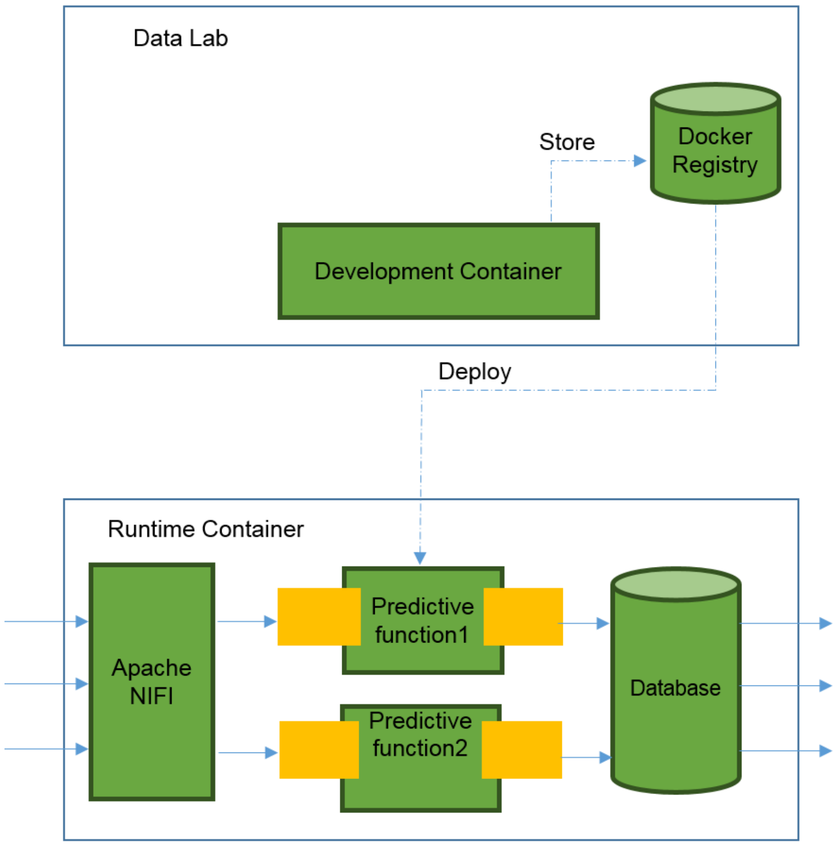

Figure 6 – Position of predictive function in Runtime container. The dotted lines indicate the deployment flow. The solid lines indicate data flow.

4 Simulation and Validation tools

Validation is a procedure used to check whether a predictive function meets predefined requirements and specifications and fulfils its intended purpose. In other words, validation checks whether the specification captures the end user needs and helps to fill the gaps between the expectations and the current results.

In MONSOON project, the validation will be focused on the functions created to will support:

- quantitative evaluation of the predictive functions on the validation datasets using the various metrics identified with the data scientists (e.g. approximation/prediction error, rate of false positive alerts, etc.)
- methods for sensitivity analysis how the function outputs are influenced by the changes in the inputs and how robust is the prediction when the input data are influenced by the noise or during monitoring failures.

The Simulation framework is in charge to run the predictive function using the data specified by data scientist. It is the component that allows the validation approach proposed above. The simulation framework has three sources of data:

- Historical data, so data scientist prepares validation datasets and store them in distributed file system;
- Real-time operation data from site, in this way, it is possible to monitor predictive function in Data Lab before it is deployed in run-time environment;
- Perturbed data from a data generator component.

The validation methodology along with the simulation framework of MONSOON project were decided since the beginning of the project and there are no further updates since then. Their description is provided in deliverable D4.6 – Initial Multi-scale Model based Development Environment.

5 Conclusion

In this document, the finalviews of the Multi-scale Model Based Development Environment have been described.This document provides the final architecture of the development environment of the MONSOON platform, updates on tools for generation of models, predictive functions packaging and simulation and validation tools.

List of Figures

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 723650.