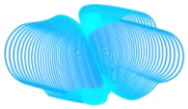
ICT-40-2020
H2020-ICT-2018-20



DataCloud

ENABLING THE BIG DATA PIPELINE LIFECYCLE ON THE COMPUTING CONTINUUM

D5.3: EVENT-DETECTION AND INFRASTRUCTURE AND DEPLOYMENT ADAPTATION

https://datacloudproject.eu/

Document Metadata

Work package	WP5
Date	24.10.2022
Deliverable editor	Dragi Kimovski (AAU)
Version	1.0
Contributors	Christian Bauer (AAU), Narges Mehran (AAU)
Reviewers	Andrea Marella (URO), Alexandre Ullises (MOG)
Keywords	Scheduling, Adaptation, Machine Learning, Utilization Prediction
Dissemination Level	Public

Document Revision History

Version	Date	Description of change	List of contributors
V0.1	10/01/2022	Structure of the deliverable	Dragi Kimovski (AAU)
V0.2	22/02/2022	Section 2	Narges Mehran (AAU)
V0.3	20/03/2022	Section 3	Narges Mehran (AAU)
V0.4	20/04/2022	Section 5	Dragi Kimovski (AAU)
V0.5	28/05/2022	Section 3, 4, 6	Christian Bauer (AAU)
V0.6	20/08/2022	Section 7	Christian Bauer (AAU)
V0.7	30/09/2022	Revision of the first completed draft	Dragi Kimovski (AAU)
V0.8	01/10/2022	Revision of the evaluation results	Christian Bauer (AAU)
V0.9	20/10/2022	Completed reviews	Dragi Kimovski (AAU)
V1.0	24/10/2022	Final formatting and layout	Brian Elvesæter (SI)

DISCLAIMER

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101016835.

Co-funded by the Horizon 2020 Framework Programme of the European Union

EXECUTIVE SUMMARY

Deliverable 5.3 describes the revised DataCloud scheduling approach and its extension with a dynamic adaptation algorithm named ADS, implemented as a software tool to lower the technological barriers to the management of Big Data pipelines over the Computing Continuum.

The described ADS scheduling and adaptation approach with event-detection improve the static scheduling with an adaptive run-time execution, empowering domain experts with little infrastructure and software knowledge to take an active part in the Big Data pipeline adaptation.

Additionally, Deliverable 5.3 describes a novel event-detection and infrastructure load algorithms for improvement of the scheduling and adaptation approach.

Co-funded by the Horizon 2020
Framework Programme of the European Union

TABLE OF CONTENTS

Co-funded by the Horizon 2020
Framework Programme of the European Union

LIST OF FIGURES

Co-funded by the Horizon 2020
Framework Programme of the European Union

ABBREVIATIONS

SAA	Scheduling and adaptation algorithm
IoT	Internet of things
LSTM	Long-short term memory
DNN	Deep neural network

Co-funded by the Horizon 2020
Framework Programme of the European Union

1 INTRODUCTION

Cloud computing is a disruptive paradigm that facilitates elastic on-demand resource-as-a-service provisioning for various Internet applications. The concurrent Internet applications encompass complex Big Data pipelines, which have a vast set of conflicting requirements, such as low execution time and communication latency. The requirements of the Big Data pipelines demand infrastructure design that pushes the services, traditionally bounded within centralized Cloud data centers, closer to their distributed data sources. The so-called Computing Continuum, which federates the Cloud services with emerging Edge and Fog resources, processes data closer to their sources, promising to reduce the network transfer latency and communication bottlenecks. However, eminent challenges in automating the management of Big Data pipelines across the Computing Continuum remain, including their effective scheduling and adaptation over heterogeneous resources from different providers [1].

Overall, the resource management and adaptation for Big Data pipelines across the continuum requires significant research effort, as the current data processing pipelines are dynamic. In contrast, traditional resource management strategies are static, leading to inefficient pipeline scheduling and overly complicated process deployment. Besides, with the advent of microservices architectures and containerization, the scheduling methods have to consider the proper data processing characteristics, facilitate networking connectivity, and provide optimized configuration of the Big Data pipeline workflows with guaranteed scalability from the execution performance perspective.

To address these needs, we propose in this deliverable an extension of the DataCloud scheduling approach with a dynamic adaptation algorithm named ADA-PIPE Scheduler (ADS), implemented as a software tool to lower the technological barriers to the management of Big Data pipelines over the Computing Continuum. ADS improves the static scheduling with an adaptive run-time execution and machine learning based resources utilization prediction, empowering domain experts with little infrastructure and software knowledge to take an active part in the Big Data pipeline adaptation. ADS covers three iterative phases that improve the scheduling, deployment, and execution of Big Data pipelines in the computing continuum:

- Initial static scheduling of Big Data pipelines to support their execution over heterogeneous resources.
- Prediction of resources utilization during Big Data pipeline execution due to the static scheduling.
- Adaptation of the run-time execution.

Therefore, the **Key Innovation Aspects** of the deliverable include:

- General model for Big Data pipeline scheduling and adaptation.
- Machine learning based approach for event detection during execution and prediction of the utilization rate of the resources across the computing continuum.
- Architecture design consisting of two interoperable subcomponents for scheduling and adaptation that jointly automate the Big Data pipelines lifecycle.
- Detailed performance evaluation of the utilization prediction algorithm used for improved adaptation.

Co-funded by the Horizon 2020
Framework Programme of the European Union

2 STATE-OF-THE-ART

This section reviews the state-of-the-art resource prediction methods for adaptation on the computing continuum.

Tuli et al. [2] proposed a novel prediction and mitigation method using an Encoder long-short term memory (LSTM) model for large-scale cloud computing infrastructure. This method aims at reducing the application response time while maintaining the service level agreement between the application owner and resource provider.

Ngo et al. [3] considered multiple anomaly detection deep neural network (DNN) models with varying complexity. Afterward, the authors explored selecting one of the models to perform autonomous detection at the most IoT, Edge, or Cloud layer. For the evaluations, the authors considered the devices such as NVIDIA Jetson-TX2 and NVIDIA Devbox with four GPU TitanX, respectively, as the Edge and Cloud server machines.

Thonglek et al. [4] designed a neural network model based on LSTM as a type of Recurrent Neural Network (RNN) to predict resource allocation based on historical data. This model has two LSTM layers each of which learns the relationship between: i) allocation and usage, and ii) CPU and memory. It aims to improve resource utilization in data centers by predicting the required resource for each data pipeline. Adaptation and event-detection in Fog and Edge

Tu et al. [5] proposed a method based on long short-term memory (LSTM) and deep reinforcement learning (DRL) to predict task dynamic information in real-time, based on the observed edge network condition and the server load. By predicting task requirements and edge devices' loads, their method offloads tasks to the optimal edge device. In the prediction model, their goal is to reduce latency and improve service quality.

Chen et al. [6] [7] proposed a learning-based method that generates resource allocation decisions with the goal of minimizing latency and power consumption called iRAF. iRAF's resource allocation action is predicted and obtained through self-supervised learning, where the training data is generated from the searching process of the Monte Carlo tree search (MCTS) algorithm.

Tan and Hu [8] used deep reinforcement learning to formulate the resource allocation optimization problem, where the parameters of caching, computing, and communication are optimized jointly.

Related methods model the Big Data scheduling and event-prediction as a machine learning problem that minimizes the data transmission and processing times but neglects the Big Data pipelines asynchronous exchange and device utilization in the computing continuum. Furthermore, they cannot properly predict the current or the future resources utilization rate.

We extend these methods by researching an event detection, scheduling and adaptation method based on a machine learning approach for event detection, utilization prediction and adaptation.

Co-funded by the Horizon 2020
Framework Programme of the European Union

3 MODEL

3.1 DATA PIPELINE MODEL

We model Big Data pipeline workflow received through the DEF-PIPE tool as $W = (M, E, Q, M_{SOURCE}, M_{SINK})$, where M is the set of Big Data pipeline tasks, E is the Big Data pipelines, Q is the data queue between the tasks, M_{SOURCE} is the data producers (sources) and M_{SINK} is the data consumers (sinks).

3.2 RESOURCE MODEL

We model the devices provided by the R-MARKET tool as a set of resources distributed over the computing continuum. We define the set of resources as $R = \{r_1, r_2, \dots, r_n\}$, where $n = |R|$ denotes the total number of resources. Thereafter, CPU_j^{util} shows a "non-idle" CPU of a resource r_j. The CPU_j^{util} is defined as: $CPU_j^{util} = CPU_j^{busy} + CPU_j^{wait}$. CPU_j^{busy} and CPU_j^{wait}, respectively, show the amount of CPU that is "busy" with processing and "waiting" for other processes to be finished. In addition, we define the utilized memory as $MEM_j^{util} = \frac{MEM_j busy}{MEM_j total}$.

3.3 NETWORK MODEL

We model the network channels between the computing resources as the round-trip latency LAT and network bandwidth BW between the resources r_i and r_j.

Afterward, we denote the number of data bytes/packets sent and received to/from resources as:

- $DATA_{sent} = (NET_{bytesSent}, NET_{packetsSent})$ denotes the number of packets sent $NET_{packetsSent}$ represented as bytes $NET_{bytesSent}$.
- $DATA_{recv} = (NET_{bytesRecv}, NET_{packetsRecv})$ denotes the number of packets (in bytes) received by the system from other resources.

3.4 MONITORING MODEL

The monitoring states S_t that is sent at time step t from the resources to the *Monitoring and Analysis component* consisting of all monitoring metrics collected from the computing continuum. Then, we define a capacity of a resource as $MON_r^{t,}$ denoting the monitored data of resource r at time step t. In this case, S_t is defined as $S_t = \{MON_{r1}^t, MON_{r2}^t, \dots, MON_{rn}^t\}$ as the combination of monitored data of a resource r_i at a time step t.

3.5 RESOURCE PREDICTION MODEL

The monitoring system on the computing continuum records the performance metrics, such as CPU, GPU, memory utilization and network usage, along with the runtime execution of tasks. We train an *LSTM-based* machine learning model on the monitored and historical data as depicted in Figure 1. Then, the output is obtained from the LSTM prediction model. In the case that the difference between the prediction and monitored data is less than a threshold value (e.g., 0.1), we obtain the predicted under-utilized resources for task scheduling. Otherwise, we

tune the training model's parameters or add more historical data to improve the prediction accuracy.

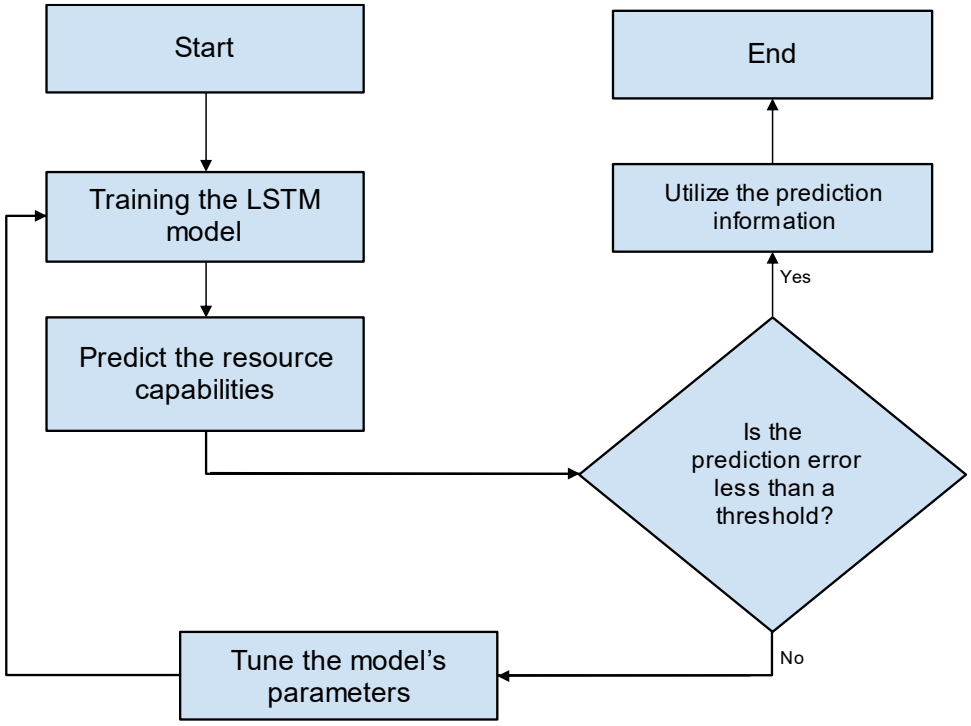


Figure 1: Flow chart of resource prediction.

3.6 ADAPTATION PROBLEM DEFINITION

The ADS adaptation approach is defined as a loop that cyclically updates every component contained in the loop. The adaptation loop is shown in Figure 2 and consists of the Big Data pipeline and the resources registered in the R-MARKET and monitored by the **Monitoring and Analysis** component, which is based on Prometheus. Both the Big Data pipeline and the resources send the gathered monitoring information of all pipelines/resources as a state at a time step P_t or S_t respectively to the Monitoring and Analysis component, where *t* denotes the current time step. The Monitoring and Analysis component then uses the states P_t and S_t, and calculates an action A_{t+1} as an update for the resources, where *t+1* denotes the next time step. This received action A_{t+1} is then used by the Adaptation component to reconfigure the resources if necessary based on different rules. The adaptation uses monitoring of tasks and resources to retrieve the states P_t, S_t . Monitoring component is necessary to obtain information about failures or performance fluctuations along with under-, over-utilization of resources.

The state, therefore, provides feedback if a resource is able to handle additional load, and thus, more tasks will potentially be mapped to it for execution. In case that the resource is not capable of handling the current load,less demanding tasks will be mapped to that resource in future scheduling cycles and it will be reconfigured accordingly. A monitored resource information consists of the CPU_j, memory MEM_j, and storage utilization, in addition to network bandwidth usage. The Big Data pipeline and resources send their states P_t or S_t respectively to the Monitoring and Analysis component. Afterward, given those states, the analyzer will determine the next action A_{t+1} of the resources. The monitoring feedback will be periodically retrieved from all registered resources and provided to the analyzer. In case of a resource side

Co-funded by the Horizon 2020 Framework Programme of the European Union

failure, the monitoring mechanism will be alerted of the occurred anomaly. The analyzer uses this monitoring data to decide if any actions $a \epsilon A$ on the current scheduling plan have to be applied.

The following actions A are considered in the adaptation approach:

- Resource load within expected parameters: When tasks on resources are well mapped, and no action has to be taken for these resources, this is denoted in the action set At+1 as such. This state is achieved when a resource is running within expected parameters, the estimated time of completing a task is not exceeded, or other anomalies occur.
- Resource load under expected parameters: When the monitoring component detects under-utilization of a resource, tasks of the pipeline that were deemed well-suited for the resource are mapped to it to be efficiently utilized.
- Resource load over expected parameters: There are multiple options if a resource is over-utilized. If it is detected that horizontal scaling solves the over-utilization and the resource instance is capable of being scaled up, it will be reconfigured to be within expected utilization levels. Furthermore, if an ill-defined task is detected and is mapped to a resource that isn't capable of fulfilling the computation within a reasonable time frame, this task will be migrated to another resource.

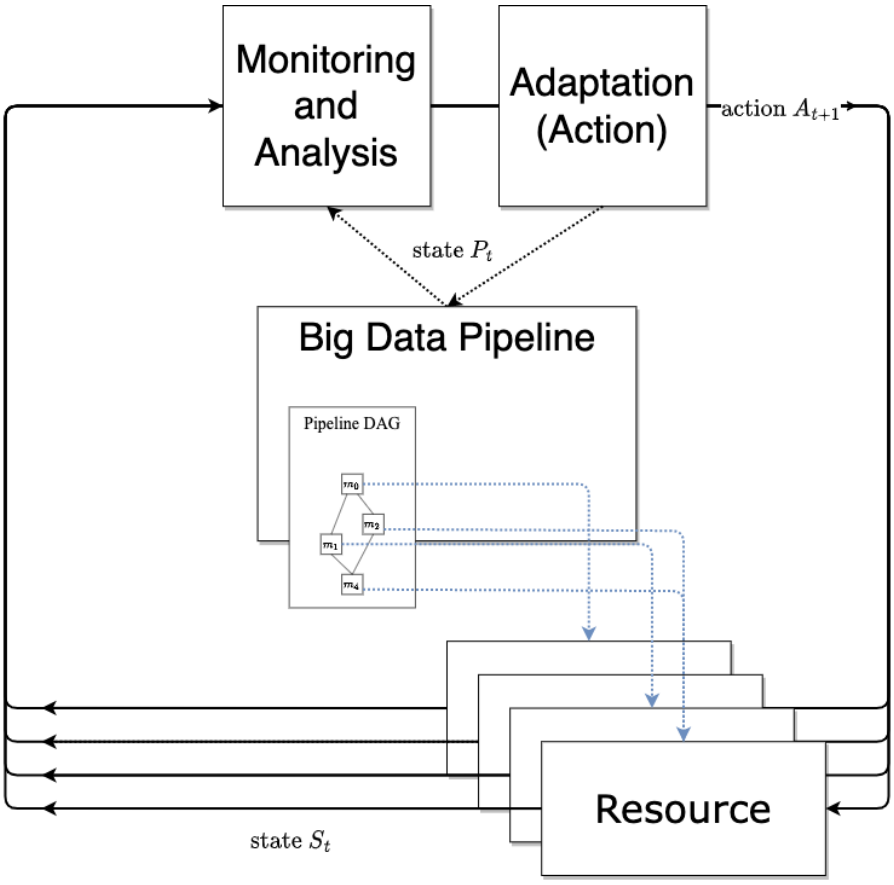


Figure 2: Adaptation loop.

Co-funded by the Horizon 2020 Framework Programme of the European Union

4 ADAPTATION, MONITORING AND ANALYSIS

This section provides information on the core component of the ADS adaptation and analysis tool namely, the machine learning based resources utilization prediction.

4.1 PREPROCESSING OF DATA

The data of monitored resources are stored in *comma-separated values (CSV)* files.

The manipulation of the data is done in **Python 3** using the data analysis module **Pandas** in so-called *dataframes*.

Monitored data is sorted based on the *start date* of the task from earliest to latest.

The first step of the data preprocessing step is the filtering of this monitored data to only include necessary values for the analysis. The first filter omits all tasks that didn't successfully terminate. Next, for each datapoint, only required columns will be kept.

Categorical data like *task_name* is then transformed to *one-hot encoded* [9] data. This is done to improve the prediction of the resource utilization.

Then, the data gets prepared to be included in multiple modules of the machine learning framework **Pytorch**. For this, a custom implemented Pytorch class called *GPUDataset* is used that transforms the data further to be able to analyze it.

Afterward, the dataset gets split into a *feature set* called *X* and a *label set* called *y*.

In both datasets, the values are then scaled with the module **scikit-learn** [11]. The values of the feature set are transformed to unit variance scale using standard deviation. The values of the label set are transformed with *min-max scaling* that translates each value to a given range.

Standardization of a dataset is a common requirement for machine learning estimators.

In the last step of the data preprocessing, the values will be stored as a **Pytorch Tensor** data type.

4.2 PREPARING FOR TRAINING

To observe the performance of a machine learning regression model, we use the following the evaluation metrics:

- Root Mean Squared Error (RMSE)
  - The RMSE of an estimator $\underline{\theta}$ is defined as the square root of the *mean square error* using an estimated parameter θ.
  - $RMSE(\underline{\theta}) = \sqrt{MSE(\underline{\theta})} = \sqrt{E((\underline{\theta} - \theta)^2)}$
- Mean Absolute Error (MAE)
  - The MAE is a measure of errors between observed y_i and predicted x_i values in a dataset containing of n observed-predicted value pair.
  - $MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$

Co-funded by the Horizon 2020 Framework Programme of the European Union

The closer the results of these metrics are to zero, the more accurate is the set of predicted data points.

The performance and metrics of each training run are tracked and compared with help of the framework *Weights & Biases* (in short*:* **wandb**)*.*

After this, the dataset will be loaded into memory with the aforementioned class *GPUDataset* transforming it into a fitting structure. Based on this dataset, hyperparameters for the machine learning (ML) model are generated, such as the *input-, hidden size,* and the *number of classes* that the output will take into account. Other hyperparameters are the *learning rate* and *number of epochs*. Those hyperparameters are then logged with **wandb** and these hyperparameters changes along with the resulting performance of the ML model can directly be compared to past ML models and their respective hyperparameters.

Next, the ML model gets instantiated. For the prediction of the resource utilization, we use a *Long-Short Term Memory* model, which is a further development of *Recurrent Neural Networks (RNN)*. Next, the corresponding loss function and optimizer are created based on the ML model. As a loss function, the **Pytorch** class **MSELoss** (MSE = Mean Square Error) is used, and the **Pytorch** optimizer implementation of the popular **AdamW** algorithm [10] was chosen.

Figure 3 shows the ML model architecture. As observed, the input data is first sent to an **init_layer**, which is a fully connected linear layer. This layer presents the *initial hidden state* and *internal state* of the LSTM module. The output of the **init_layer** then is sent to the LSTM module, which then transforms the data with multiple operations seen, that are displayed as the small blocks between the **Transpose** and **LSTM** block in Figure 3. The transformed data is then passed through the LSTM layer and its corresponding weights:

- $W: model\ weights,$
- $R: recurrent\ weights,$
- $B: bias\ weights.$

Once the data has passed through the LSTM layer and its weights, the data is sent to multiple fully connected layers after another. Each fully connected layer gets smaller in size compared to the previous fully connected layer. At the final block denoted as **abs** the absolute of the **fc_3** output value is calculated and returned as the prediction of the model for a given input.

Then, the ML model and loss function will be sent to the desired hardware to be trained on. This hardware device can be a dedicated graphics card able to compute the training via the parallel computing platform and programming model of **Compute Unified Device Architecture (CUDA)** provided by NVIDIA corporation. This is done to reduce the training time. If no hardware on the device is able to train via **CUDA**, the training will be done on the onboard CPU.

Co-funded by the Horizon 2020
Framework Programme of the European Union

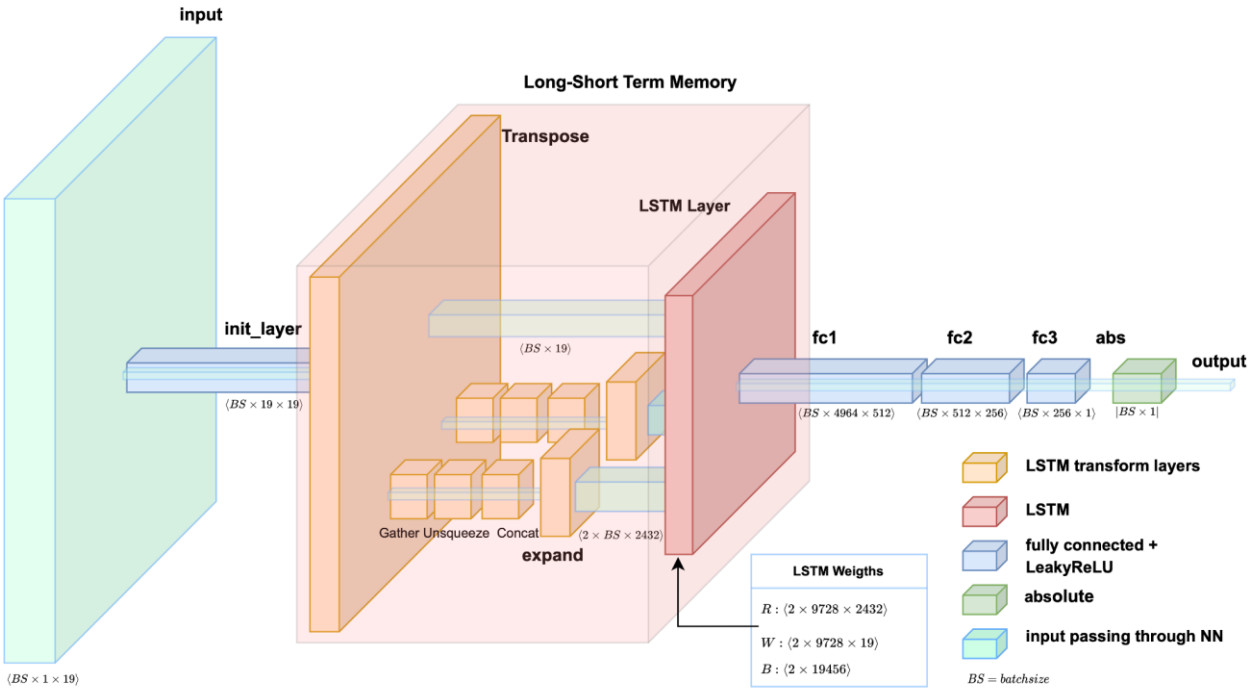


Figure 3: Long-Short Term Memory architecture.

4.3 TRAINING LOOP

The progress of the training loop is visualized by the module **TQDM** [12], which displays a training progress bar and how many seconds each iteration takes to complete.

The entire data set is then loaded into a **PyTorch Dataloader** that provides functionalities such as defining the *batch size,* if a batch should be *randomly shuffled*, multithreading the *number of workers,* and the option of using a *sampler.*

Dataloader Parameters:

- *Batch size:* How many features are used for the training at once. This also heavily influences the prediction of the future resource utilization.
- *Randomly shuffled:* This is a boolean option set to **False** in the training case, since we require the time-series to be in chronicle order.
- *Number of Workers:* The number of workers defines how many subprocesses will be used for data loading.
- *Sampler:* A sampler defines the strategy that draws samples from datasets. A **PyTorch SubsetRandomSampler** is used as the strategy to ensure randomly picked batches each iteration, so the model does only take the structure of a batch into account, but not the entire training cycle of a dataset.

Each batch is then split into a set of features and a set of labels (also called ground truth) and is sent to a **CUDA** graphics card if available.

Next, the **AdamW** optimizer sets the ML model's parameter gradients to zero. “It is beneficial to zero out gradients when creating a neural network model. This is because, by default,

gradients are accumulated in buffers (i.e., not overwritten) whenever `.backward()` is called." *PyTorch - Zeroing out gradients in PyTorch [13].*

After zeroing the gradients, the *feature set* gets fed to the ML model to receive a prediction.

This prediction is then compared with help of the *loss function* and the *label set.* Depending on the calculated loss, the gradients of the ML model are calculated with the `.backward()` function. Then, the **AdamW** optimizer uses the calculated gradients and updates the weights of the ML model with its `.step()` function. For each prediction, the *root mean squared error* and *mean absolute error* gets calculated and is logged with the *loss value* to **wandb**.

This is done for each batch, and once all batches of the dataset are used for the training, the next training epoch starts, and the entire process is repeated with the updated ML model parameters for a predefined amount of epochs.

After the training loop is finalized, the corresponding hyperparameters and weights of the ML model and its optimizer are stored on a hard drive, so it can be used at a later time to either train the model further or to use it as a prediction model for resource utilization.

4.4 EVALUATION OF THE TRAINING PERFORMANCE

After the training process, the ML model is set to evaluation mode so the predictions of the training process can be visualized without further training the model. Since the dataset from the DataCloud monitoring tool used for training was scaled to fit the ML model, the transformation of the feature and label set is converted to a **Numpy array** [14] and inverted to retrieve the original values again. These values are then converted back to **Pandas Dataframes** for the calculation of the performance via metrics and to visualize the comparison of prediction and actual data.

Then, the overall *root mean square error* and *mean absolute error* are calculated and logged with **wandb**. For the visualization of the results, the **Python** library **Matplotlib** is used. The results are visualized for each label in the labelset, (i.e., CPU utilization, average memory usage, runtime). *Note: The evaluation and visualization of the training dataset are primarily conducted to compare the performance of different training runs and to observe the behavior of the prediction if a heavy resource utilization is expected. The actual evaluation is performed with a test dataset that is independent of the training dataset.*

Next, the training dataset, which is gathered from the DataCloud monitoring tool based on Prometheus, is prepared in the same manner as the training set above for the actual evaluation of the ML model performance. A prediction is made based on the feature set and is inverted and compared to the actual data points using the aforementioned metrics and also graphically visualized with **Matplotlib**. The results and graphs of each label are logged with **wandb** for later comparison and to see the actual performance of the ML model on a previously not seen dataset.

5 INTEGRATION

5.1 SCHEDULING AND ADAPTATION INTEGRATION WORKFLOW

In this section, we provide a detailed description of the interactions between the ADS scheduling and adaptation algorithms with the possible deployment frameworks.

- **Step-1**: To begin with, the Big Data application owner submits the pipeline for deployment over the Computing Continuum using DEF-PIPE. In addition, the owner provides information on the pipeline structure, tasks, and executable data.
- **Step-2**: After receiving the structure of the pipeline, the scheduling algorithm performs dependency analysis between the tasks in the pipeline to identify the correct sequence of tasks and those that can be executed in parallel.
- **Step-3**: Afterwards, the scheduling algorithm analyzes the requirements of each specific task, such as processing speed, memory, and storage sizes.
- **Step-4**: Based on the identified task requirements, the scheduling algorithm searches for appropriate resources in R-MARKET with associated performance metadata.
- **Step-5**: The scheduling algorithm analyzes the resources based on the requirements of the pipeline tasks, creates a schedule for every pipeline task, and sends it to the deployment and run-time system (such as Kubernetes [15]).
- **Step-6**: The deployment system deploys the pipeline tasks using the DEP-PIPE tool on the resources following the schedule provided by the scheduling algorithm.
- **Step-7**: A monitoring system observes the execution of the Big Data pipeline and the behavior of the computing continuum resources.
- **Step-8**: The monitoring information is sent to the adaptation algorithm for analysis of over-utilization (such as high CPU load or low amount of operating memory available).
- **Step-9**: If over-utilization is predicted, the adaptation algorithm uses a rule-based mitigation strategy. For example, in the cases where there is no available operating memory, a migration of the Big Data pipeline will be performed.
- **Step-10**: The ADS adaptation algorithm continuously monitors the execution of the Big Data pipeline and applies the rule-based mitigation actions until the pipeline finishes with its execution.

The integration of the prediction is done by loading a pre-trained ML model from the disk and executing it in evaluation mode. For making a prediction, current, monitored data is sent to the Pre-processing Unit inside the Monitoring and Analysis component, that prepares the data to be fed into the ML model. This pre-processed data is then forwarded to the Analysis component part in which the ML model resides.

The ML model returns a prediction based on the monitored data. This prediction is then sent to the Adaptation component, which then decides based on this prediction and the current state of the resources and network, how to proceed in regard to scheduling, provisioning and migration.

Co-funded by the Horizon 2020
Framework Programme of the European Union

5.2 SCHEDULING AND ADAPTATION FUNCTIONAL ARCHITECTURE

We present in this section the architecture of ADS in one integrated tool in detail.

The architecture of the integrated ADS scheduling and adaptation tool consists of five components, interconnected with the other DataCloud tools as displayed in Figure 4.

- **Dependency analysis** orders the candidate tasks scheduled on each resource in a preference list based on the aggregate pipeline communication time to each task. The results for the dependencies analysis are provided by DEF-PIPE using the defined API.
- **Execution event detection** uses data from an external monitoring system to identify anomalies during execution based on an ML approach defined in Section 4. The data is received for a Prometheus based monitoring system provided by DEP-PIPE, which continuously monitors the Big Data pipelines and the computing resources.
- **Scheduling** maps the pipeline tasks to the resources using a matching theory algorithm applied to the task and resource preference lists in response to infrastructure drifts. In order to provide better scheduling results, ADS receives information from the SIM-PIPE tool that gives suggestions on possible schedules that provide higher performance.
- **Adaptation** dynamically applies re-scheduling or migration of the tasks based on the analysed monitoring received from the execution event detection. Both the ADS scheduling and adaptation interact with the deployment and orchestration system of DEP-PIPE and receive information on available resources from R-MARKET to detect performance bottlenecks using a ML algorithm and take appropriate measures against them.
- **Public API** enables the integration of the ADS scheduling and adaptation tool with the other DataCloud tools. The API provides predefined interfaces, already described in Deliverable 1.2.

Co-funded by the Horizon 2020
Framework Programme of the European Union

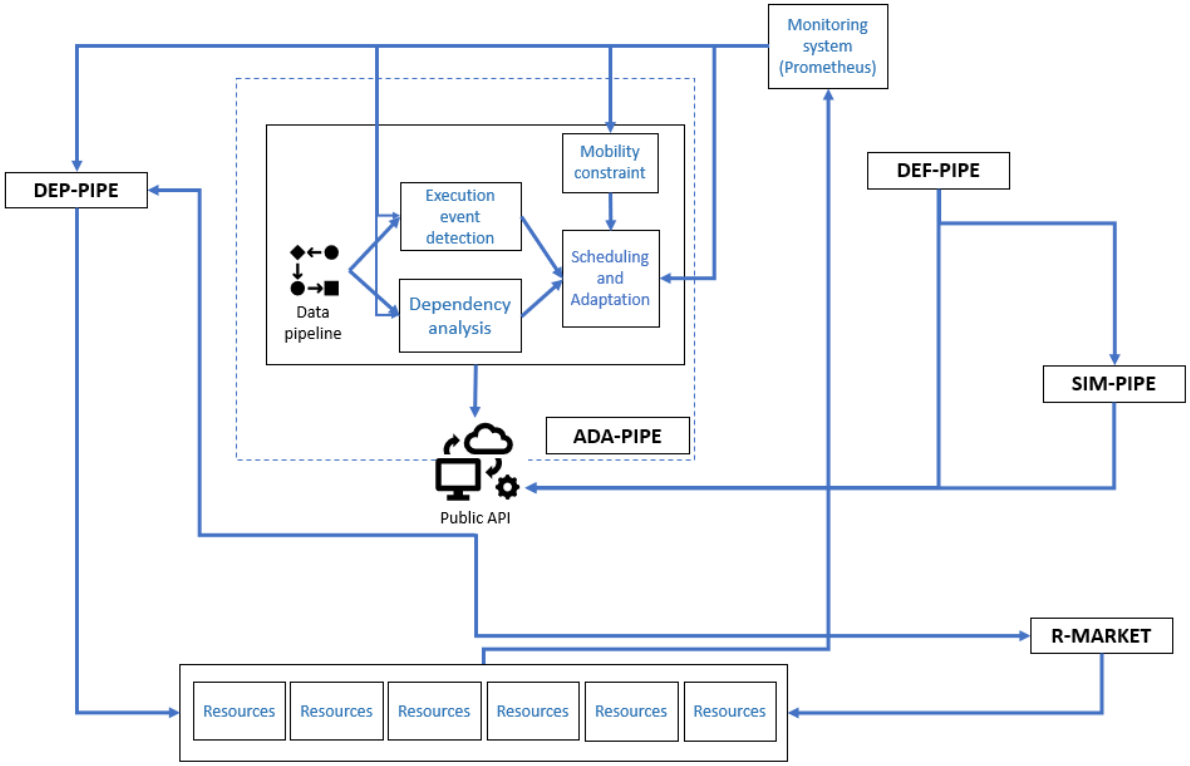


Figure 4: ADS Scheduling and adaptation tool architecture and integration with DataCloud.

Co-funded by the Horizon 2020 Framework Programme of the European Union

6 PRELIMINARY EXPERIMENTAL EVALUATION

The evaluation of the performance of the resource utilization prediction algorithm for the adaptation tool is preliminary done with **Python Jupyter** notebooks. The ML model to evaluate is loaded from disk with all necessary parameters. Next the test dataset is loaded and transformed to be able to be fed to the ML model. Next the features of the test dataset are forwarded to the ML model, which returns the predicted labels **y**. The performance of the prediction is then compared to the actual values with popular regression metrics described in Section 4.2 Preparing for training.

Following are the parameters used for the testbed:

Parameter Name	**Parameter Value**
Batch size	25
Timesteps into the Future	10
Number of epochs	200
Learning rate	0.001
LSTM input size	19
LSTM hidden size	2451 (19 * 128)
LSTM number of layers	1
LSTM number of classes	1
Loss criterion	Mean Square Error Loss (MSELoss)
MSELoss size_average	10
Optimizer	AdamW

Table 1: Testbed parameters

Furthermore, for proper evaluation we have defined the following performance metrics:

- Root Mean Squared Error (RMSE)
  - The RMSE of an estimator $\underline{\theta}$ is defined as the square root of the *mean square error* using an estimated parameter θ.
  - $RMSE(\underline{\theta}) = \sqrt{MSE(\underline{\theta})} = \sqrt{E((\underline{\theta} - \theta)^2)}$
- Mean Absolute Error (MAE)
  - The MAE is a measure of errors between observed y_i and predicted x_i values in a dataset containing of n observed-predicted value pair.
  - $MAE = \frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$

Co-funded by the Horizon 2020 Framework Programme of the European Union

In what follows we present the preliminary results of the resource utilisation prediction and event detection for the adaptation too. To begin with, Figure 5 shows a runtime prediction based on a test dataset in which the black line represents the actual runtime of tasks and the green line represents the predicted runtime based on previous tasks. We successfully were able to detect multiple runtime spikes using the ML model. The metrics used for evaluating how well the ML model are the *root mean square error (RMSE)* and *mean absolute error (MAE).* The results of the metrics can be seen in figure 5 in the top-right corner. The closer the values are to zero, the closer is the prediction to the actual values.

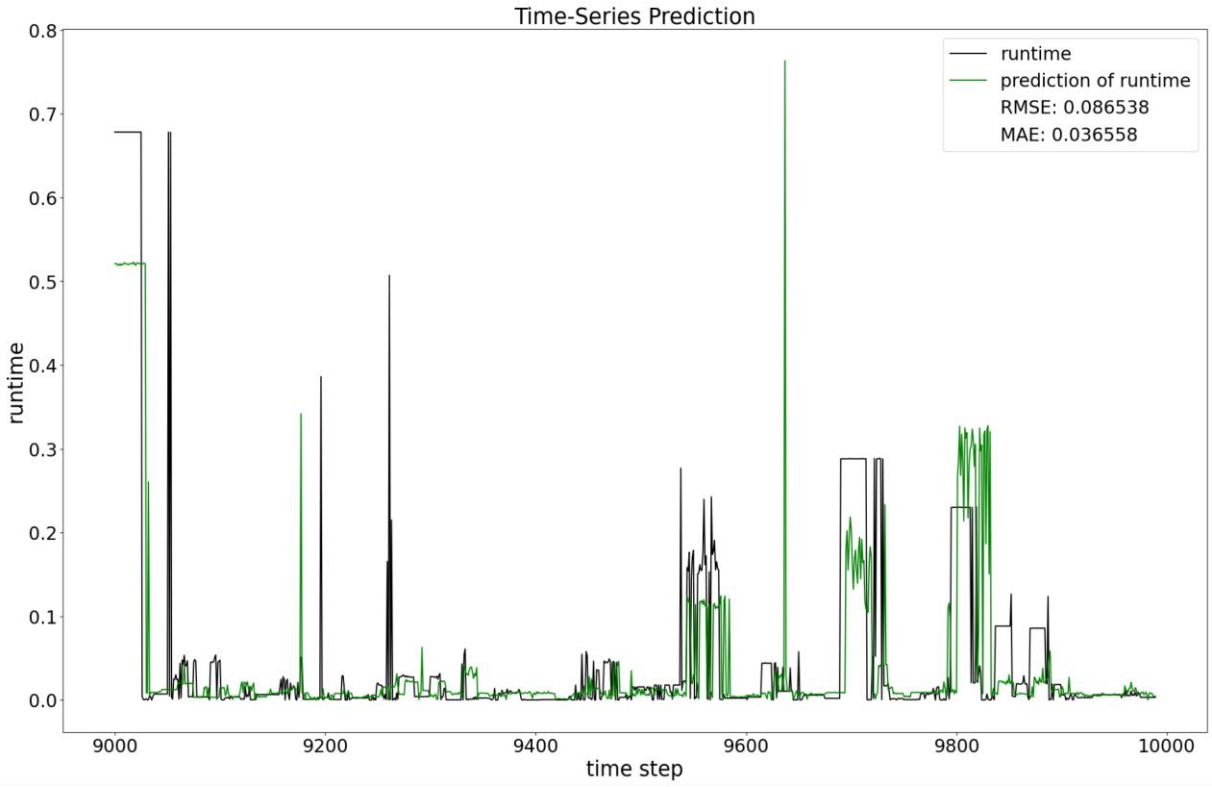


Figure 5: ML model runtime prediction

As can be observed, the ML model was able to predict multiple important runtime spikes at time steps 9000, 9500-9600, 9700 and 9800. In addition, the lower runtime trends were predicted with good accuracies. There are two occurrences in which the ML model did not predict or wrongly predicted a utilization spike. The first one can be seen at approx. 9250, where an actual runtime spike occurred but wasn't detected. The reason for this might be that a similar occurrence wasn't found in the training data. The second, wrongly predicted spike happened for similar reasons. A pattern was found in the training data that led to a runtime spike that was also predicted to be present in the test dataset.

Co-funded by the Horizon 2020
Framework Programme of the European Union

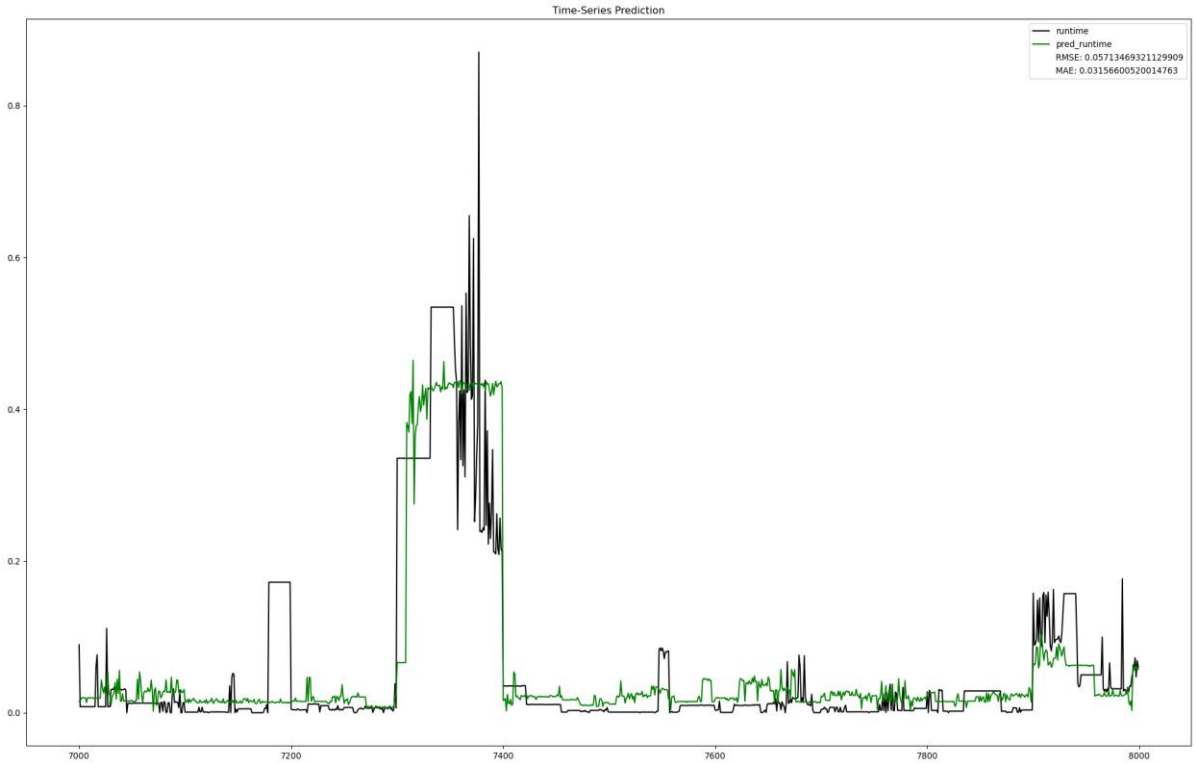


Figure 6: ML model runtime prediction with constant time step

As can be seen in Figure 6 a long-running runtime spike was correctly predicted at time step range 7300-7400. Also, a smaller spike was predicted correctly at 7900-8000. This implies that the ML model is able to predict future runtime spikes based on historical data if there is a reappearing pattern. For example, if it is highly probable that a sequence of tasks are executed after another, part of that sequence can be used to predict how much the successor tasks will impact the runtime of the whole system. Given the figure 6, we can predict important, long living runtime spikes but the runtime spike at 7150-7200 was not predicted by our ML model. This missing prediction, and that the average prediction was higher than the actual average prediction for low runtime tasks need to be improved in further development.

In Figure 7, a smaller time step range was chosen to be able to analyze smaller changes in the runtime in more detail. The overall trend was predicted and the first runtime spike could be predicted with some minor error in the time step offset. The next runtime spike could not be predicted properly, since the next predicted spike was off by a larger time frame than the first predicted runtime spike. This could be due to a unique runtime spike not seen before, or since there is an offset between the actual runtime spike and the predicted one, it is also possible that the runtime spike happened at a later time frame than it did in this scenario. Further improvements in the ML model and investigation and training on additional data might improve the prediction to be able to better predict such runtime spikes at the correct time frame.

The latter two runtime spikes could be predicted with some error in the amplitude in the prediction. While being correct in detecting a spike, these predicted outcomes are too low compared to the actual data and might be the result of not taking sufficient action.

Co-funded by the Horizon 2020
Framework Programme of the European Union

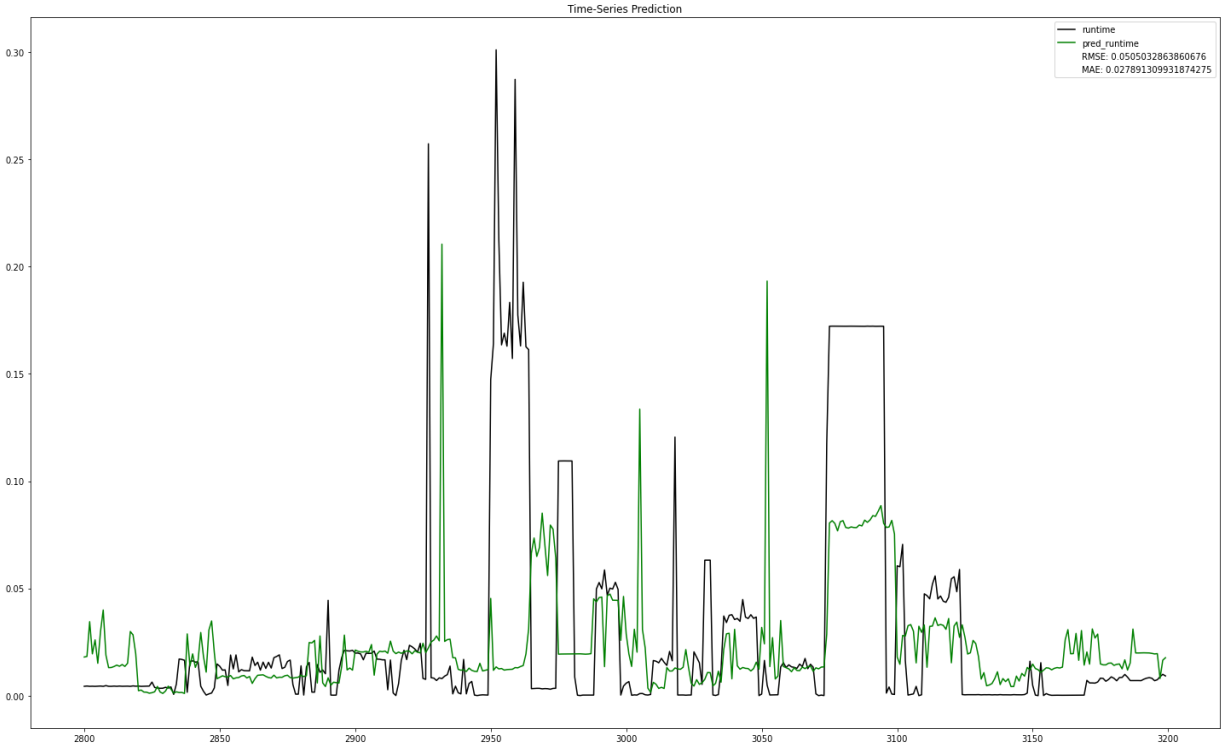


Figure 7: ML model runtime prediction with decreasing time step

Co-funded by the Horizon 2020
Framework Programme of the European Union

7 CONCLUSIONS

In this deliverable we presented an extension of the scheduling algorithm and its extension with adaptation approach for automating the lifecycle of Big Data pipelines execution in the Computing Continuum. The presented ADS approach separates the scheduling of the Big Data pipelines from the run-time adaptation stage, enables event detection and prediction of resources utilization for improved execution adaptation, thus empowering domain experts with little infrastructure and resources know-how to participate in their operation actively. ADS introduced a two-sided matching-based scheduling and machine learning adaptation methods integrated with R-MARKET resources and DEP-PIPE deployment tool. In addition, ADS is capable to adapt the execution of the pipeline on-the-fly by employment of state model to improve the execution the tasks based on their run-time requirements.

We, therefore, presented the detailed architectural design of the ADS tool and conducted a feasibility analysis for the resource's utilisation end event detection prediction. We presented concrete scenarios supported by authentic snapshots demonstrating how the ADS implementation automate the management of the lifecycle on a real Computing Continuum infrastructure.

Co-funded by the Horizon 2020
Framework Programme of the European Union

REFERENCES

[1] Barika, M., Garg, S., Zomaya, A. Y., Wang, L., Moorsel, A. V., & Ranjan, R. (2019). Orchestrating big data analysis workflows in the cloud: research challenges, survey, and future directions. ACM Computing Surveys (CSUR), 52(5), 1-41.

[2] Tuli, S., Gill, S. S., Garraghan, P., Buyya, R., Casale, G., & Jennings, N. (2021). Start: Straggler prediction and mitigation for cloud computing environments using encoder lstm networks. IEEE Transactions on Services Computing.

[3] M. V. Ngo, T. Luo, H. Chaouchi, & T. Q. Quek. Contextual-bandit anomaly detection for IoT data in distributed hierarchical edge computing. In 2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS) (pp. 1227-1230). IEEE, November 2020.

[4] K. Thonglek, et al. "Improving resource utilization in data centers using an LSTM-based prediction model." 2019 IEEE International Conference on Cluster Computing (CLUSTER). IEEE, 2019.

[5] Y. Tu, H. Chen, L. Yan, and X. Zhou. Task offloading based on LSTM prediction and deep reinforcement learning for efficient edge computing in IoT. Future Internet, 14(2), 30, 2022.

[6] J. Chen, S. Chen, Q. Wang, B. Cao, G. Feng, J. Hu. iRAF: A Deep Reinforcement Learning Approach for Collaborative Mobile Edge Computing IoT Networks. IEEE Internet Things J. 2019, 6, 7011–7024.

[7] J. Chen, S. Chen, S. Luo, Q. Wang, B. Cao, X. Li. An Intelligent Task Offloading Algorithm (iTOA) for UAV Edge Computing Network. Digit. Commun. Netw. 2020, 6, 433–443.

[8] L. T. Tan and R. Q. Hu, "Mobility-aware edge caching and computing in vehicle networks: A deep reinforcement learning." IEEE Transactions on Vehicular Technology 67.11 (2018): 10190-10203.

[9] Brownlee, Jason, Machine Learning Mastery, Why One-Hot Encode Data in Machine Learning? accessed 22 September 2022, <https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/>. Accessed 22 September 2022

[10] Kingma, D. P., and Jimmy Ba. "Adam: A method for stochastic optimization." arXiv preprint arXiv:1412.6980, 2014.

[11] F. Pedregosa, et al. "Scikit-learn: Machine learning in Python." the Journal of machine Learning research 12 (2011): 2825-2830.

[12] da Costa-Luis, Casper, et al. "tqdm: A fast, Extensible Progress Bar for Python and CLI." *Zenodo* (2021).

[13] PyTorch 2022, Zeroing out gradients in PyTorch accessed 13. September 2022, <https://pytorch.org/tutorials/recipes/recipes/zeroing_out_gradients.html#zeroing-out-gradients-in-pytorch>

[14] Van Der Walt, Stefan, S. Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation." *Computing in science & engineering* 13.2 (2011): 22-30.

[15] Luksa, Marko. Kubernetes in action. Simon and Schuster, 2017.

Co-funded by the Horizon 2020 Framework Programme of the European Union