Deliverable D2.2. Accompanying document for the µDevOps learning engine package

May 2022

1 Scope

This is the accompanying document of Deliverable D2.2 of the µDevOps project entitled, "The µDevOps learning engine package". The type of the deliverable is marked as *Other*, and includes software artifacts along with this accompanying document. The delivered artifacts are also made available at the following GitHub repository:
`https://github.com/uDEVOPS2020/ContextLearning`
and at the linked Zenodo repository:
`https://doi.org/10.5281/zenodo.6628321`, indexed by OpenAIRE

2 Introduction

This document describes the *learning engine* artifact. The artifact implements the *context learning* feature of the testing process outlined in the project, which is paramount to support Software Quality Assurance (SQA) activities in Microservice DevOps. Specifically, this document reports about the context learning functionalities developed in the context of the project, along with implementation details. Instructions are provided for using the learning functionalities and reproducing the use cases we have developed for illustrative purpose. The artifact will be extended during the project, as the Consortium will advance with the WP3, WP4 and WP5.

3 The learning engine

Figure 1 gives an overview of the context in which the learning engine can be used. The engine takes data gathered from monitoring and a specification of the decision (i.e., the SQA objective) to pursue. Based on this, the proper learning

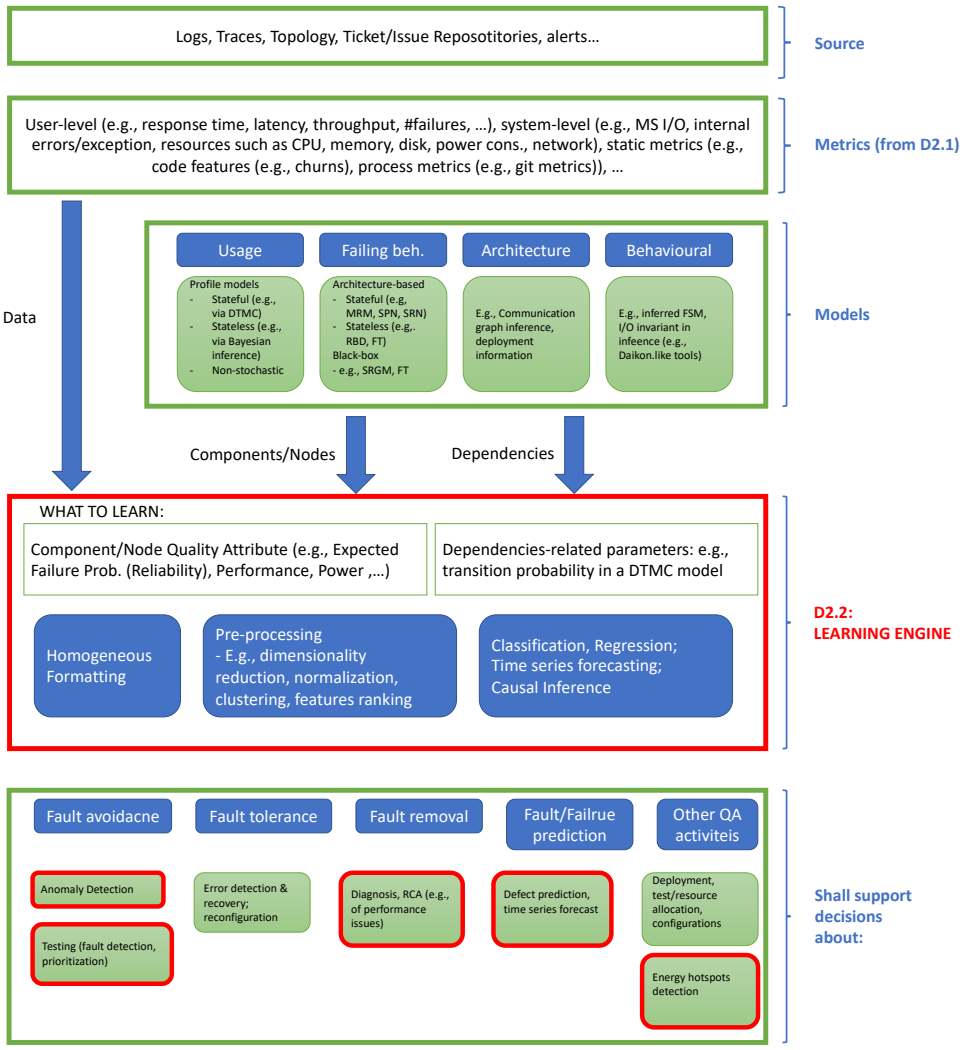

Figure 1: Context of use for the µDevOps learning engine.

algorithm is used, with associated pre-processing steps when needed, and gives, as output, the prediction supporting that decision. The boxes highlighted red represent the SQA activities it supports.

The engine customizes the implementation of a conventional machine learning workflow for the purpose of supporting the SQA decisions. It exploits open source libraries that implement machine learning and causal inference algorithms. The following Section describes the learning functionalities through a set of use cases implemented in the project.

The artifact is not meant to interact directly with the end user (it will be

part of the proof of concept foreseen in WP5 and, as such, will act as a library called by other components of the proof of concept); however, the code provided with Deliverable 2.2 has a minimal form of interaction with the user, allowing him/her to load data, ask for the desired processing, and retrieve the result.

The context learning functionalities are described in the following, with respect to the supported SQA activities of Figure 1; these are especially relevant for engineering Microservice Archicteture (MSA) applications in a DevOps environment. They are about: (Regression) Testing, Performance-related causal structure discovery in a Microservice Architecture, Energy consumption anomaly detection and root cause analysis, Just-in-time software defect prediction. In the following, we first describe the functionality, then a use case we implemented for each of them.

3.1 Functionality 1: Test case prioritization via Learning-to-rank techniques

- **Description**. The objective of this functionality is to support test prioritization in an MSA, namely: given a list of tests to run, the goal is to run first the ones more likely to expose failures. Prioritization is done by applying machine learning (learning-to-rank) algorithms to features of request/response and/or of the invoked Microservice that correlate more with quality metrics, such as performance (e.g., response time), reliability (e.g., status code) or coverage. The feedback allows for prioritizing test cases. The tests can be already given, or can be generated automatically by the testing tool we are developing in the context of this project, called `uTest`[1], starting from the API specification of the microservices under test.
- **Source of information**: *Execution traces* of previously executed testing sessions, or resulting from monitoring of the application during operation.
- **Metrics**: The metrics used as features of the test cases are:
  - testID
  - HTTP status code
  - Response Time
  - URL
  - HTTP method
  - Input Class 1 | ... | Input Class N |

  Response Time and status code are both considered as objectives to prioritize the tests (namely, we want to run first those tests with failing status code and high response time). The *ranking score* is computed as follows:

$$ranking\_score = response\_code + \frac{1}{response\_time} \times 100 \qquad (1)$$

[1] `https://github.com/uDEVOPS2020/uTest`

The response codes higher than or equal to 400 correspond to failed tests; at the same time, a lower response time implies a higher priority of the test. As result, the higher the value of the ranking score, the higher the priority of the considered test.

- **Modeled facet**: *Behavioural*, *Failing behaviour*
- **Type of model**(s): Learning-to-rank algorithm
- **Learning algorithm** used: Learning-to-rank (classification/regression) algorithms included in the open source `RankLib` library[2]:
  - MART (Multiple Additive Regression Trees, a.k.a. Gradient boosted regression tree)
  - RankNet
  - RankBoost
  - AdaRank
  - Coordinate Ascent
  - LambdaMART
  - ListNet
  - Random Forests
- **Decision support category**: *Fault avoidance*
- **Decision support** about: what regression tests to run first in order to support early fault detection.

3.2 Functionality 2: Performance-related causal structure discovery in a Microservice Architecture

- **Description**. The objective of this functionality is to characterize the cause-effect relationships in a Microservice Architecture (MSA) in terms of performance, measured as response time. It allows for identifying performance-related dependencies that can complement the behavioural dependencies information, inferred for instance from the methods call, in order to figure out to what extent the performance variation of a microservice impacts performance of another microservice. The analysis allows for discovering hidden cause-effect relations between microservices, useful for design (e.g., load balancing, architectural decisions) and testing purposes (e.g., focus more on microservices found to be a performance bottleneck).
- **Source of information**: *Execution traces* of previously executed testing sessions, or resulting from monitoring of the application during operation.

[2]Dang, V. "The Lemur Project-Wiki-RankLib." Lemur Project,[Online]. Available: `http://sourceforge.net/p/lemur/wiki/RankLib.`

- **Metrics**: The metrics used the observed response times of each microservices
- **Modeled facet**: *Architectural*, *Failing behaviour*
- **Type of model**(s): Causal structure discovery algorithm for time series.
- **Learning algorithm** used: Causal structure discovery algorithms, included in the open source `Tetrad` library [3]
  - Lingam
  - PC
  - FGES
  - GFCI
  - RFCI
- **Decision support category**: *Fault avoidance, Fault removal*
- **Decision support** about: testing (e.g., focus more testing effort on microservice more causally-related to edge or critical microservice), load balancing, deployment/architectural decisions, root cause analysis of performance issues.

3.3 Functionality 3: Energy consumption anomaly detection and root cause analysis

- **Description**. The objective of this functionality is to support QA activities pertaining the detection and diagnosis of anomalous energy consumption: given an operating MSA, the goal is to identify those microservices whose energy consumption is higher than expected in a given execution scenario (*energy hotspots detection*) and then identify the microservice contributing more to it.

  This is done by applying statistical techniques on time series containing metrics of interest correlated with energy consumption.
- **Source of information**: *Execution traces* of previously executed testing sessions, or resulting from monitoring of the application during operation.
- **Metrics**: We use, as proxy of energy consumption metrics, CPU and memory consumption at container level. Note that while the availability of measurements of physical energy consumption (via hardware devices) does not affect the implementation of the functionality, it will indeed improve the accuracy of the final result. This will be investigated during the project.
- **Modeled facet**: *Architectural*, *Failing behaviour*

[3] https://github.com/cmu-phil/tetrad

- **Type of model**(s): forecast-based time series models for anomaly detection, causal models for root cause analysis.
- **Learning algorithm** used:
  - Forecast-based methods for anomaly detection (e.g., STL, GESD, IQR, Twitter), followed by multivariate transfer entropy (MuTE) to spot the most impacting containers. The algorithms are included in open source libraries [4]
- **Decision support category**: *Fault removal*, *QA Activities*
- **Decision support** about: *Energy hotspots detection*, *Diagnosis, RCA*, *deployment.*

3.4 Functionality 4: Just-In-Time Defect Prediction

- **Description**. The objective of this functionality is to support the early identification of commits more likely to introduce defects, namely: given an application developed in a continuous integration/DevOps setting (hence with frequent commits), the goal is to alert on those commits more likely to introduce a defect in the deployed code and to identify the metrics more *stable* and relevant for the prediction. This is done by applying *just-in-time* (JIT) prediction enriched with the *feature stability score* computation.

  Source of information: *Source code, commits*
- **Metrics**: Git commit metrics, such as *Number of modified subsystems, Number of modified directories, Number of modified files, Entropy, Lines of code added, Lines of code deleted, Lines of code in a file before the change, Whether or not the change is a defect fix*
- **Modeled facet**: *Failing behaviour*
- **Type of model**(s): Classification
- **Learning algorithm** used: Random Forest, Logistic Regression
- **Decision support category**: *Fault prediction*
- **Decision support** about: Test/QA effort allocation, fault tolerance. Selection of features that act as best predictors (in terms of accuracy and stability)

[4] The `anomalize` R package, available at: `https://cran.r-project.org/web/packages/anomalize/`, and the `IDTxl` toolkit for transfer entropy, available at `https://github.com/pwollstadt/IDTxl`

4 Content of the package and implementation details

The package contains the code implementing the above-mentioned four functionalities along with the datasets and code to illustrate their usage.

4.1 Use Case for functionality 1

The "use case 1" folder of the delivered package contains the dataset and code to reproduce the test prioritization example we have used to test the feasibility of the strategy:

Dataset. *testFeatures.csv*. This is the dataset used as illustrative example for this use case. It is derived by running a load on a well-known open-source benchmark for microservice architecture (MSA), named Train Ticket [5]. The application simulates a train ticket booking system, composed of 41 microservices communicating to each other via REST over HTTP. Train ticket is polyglot (e.g., Java, golang, Node.js, etc).

Workload generation. The dataset is automatically generated with a teseting tool we are developing in the context of the project's WP2, called `uTest`[6]. The tool generates tests starting from microservices' OpenAPI specifications. Configured in pairwise mode, the tool generated 4690 test cases by a combinatorial testing strategy.

The so-obtained dataset has the following columns:

`testID | HTTP status code | Response Time | URL | HTTP method | Input Class 1 | ... | Input Class N |`

Each row represents an executed test.

Training and test sets generator. The training and test sets are generated through the `csv_parsing.py` script. The datasets are encoded in the format required by `RankLib` to perform the training and the prioritization.

Prioritization. Response Time and status code are both considered to perform the prioritization. The ranking score is computed as follows: $ranking\_score = \frac{response\_code+1}{(response\_time)} \times 100$. The response codes higher than or equal to 400 correspond to failed tests; at the same time, a lower response time implies a higher priority of the test. As result, the higher the value of the ranking score, the higher the priority of the considered test.

Results. The output of the Learning-to-Rank algorithms are ordered lists of test IDs (column "1"). Testers should run the tests according to this list in order

[5] `https://github.com/FudanSELab/train-ticket`

[6] `https://github.com/uDEVOPS2020/uTest`

to expose failures (both as failing status code and as high response time) earlier.

Code. Python code files/scripts to: i) generate training and test sets, ii) train and execute the Lerning-to-Rank techniques, iii) build the ordered list of testIDs.

Prerequisites:
Python (version >3), JVM/JRE (version > 1.8).
Libraries: `RankLib` library[7]

Commands:
Run the "`sh ranking.sh`" script to perform the ranking of the selected test dataset.

4.2 Use Case for functionality 2

The "use case 2" folder of the delivered package contains the code and data to run the cause-effect characterization example.
Dataset. *MSA_medians_RT.csv*. This is the dataset used as illustrative example for this use case. It is derived by running a workload, as described below, on a well-known open-source benchmark for microservice architecture (MSA), named Train Ticket [8]. The application simulates a train ticket booking system, composed of 41 microservices communicating to each other via REST over HTTP. Train ticket is polyglot (e.g., Java, golang, Node.js, etc).

The dataset has the following columns:

| `Reponse Time median MS1` | `...` | `Response Time median MSn` |

Each row is a sample, gathered every 5 seconds.

Workload generation. The dataset is generated by stressing the system with a workload. The folder contains the files/scripts to: i) gather monitoring data, ii) parse the raw monitoring data and creating the dataset. The "locust.py" file customize the tool Locust [9], which we exploited to generate the load. Locust is a distributed, open-source load testing tool that simulates concurrent users in an application for each benchmark. We customize the workload to reflect the behavior of real users. In total, we run on average 8 requests per second for 10 minutes, with 10 active users.

Code. To derive the causal structure, run the Python notebook (`MSA_causal_model.ipynb`) to apply one of the algorithms for causal structure discovery and create a causal model. Then, the SEM model can be obtaiend by

[7] Dang, V. "The Lemur Project-Wiki-RankLib." Lemur Project,[Online]. Available at `http://sourceforge.net/p/lemur/wiki/RankLib`

[8] `https://github.com/FudanSELab/train-ticket`

[9] `https://locust.io/`

running the `Sem.java` code.

Prerequisites:
Python (version >3), JVM/JRE (version > 1.8), Locust (used with version 2.8.6).
Libraries: `py_causal`[10]. This is directly cloned in the notebook code, there is no need to clone and load it.
`tetrad-gui-7.1.0-launch.jar` (add this library to the build when running `Sem.java`) – the library is from `Tetrad`[11].

Commands:
You can run the `MSA_causal_model.ipynb` notebook.
You can then run the java code, once added the `tetrad` library to the build path: `java Sem.java <path-to-file/filename>.csv`

The example dataset file is `MSA_medians_RT.csv` located in the "Use Case 2/dataset" folder, which is the same suggested in the `MSA_causal_model` notebook.

Results. The results will be both in the form of Causal Graph (a Directed Acyiclic Graph – DAG) and in a textual form, both representing the relations between microservices in terms of response times (which response time is likely to cause any other). The types of relations (with the associated probability) can be:

A $--> $ B. A is a cause of B. It may be a direct or indirect cause that may include other measured variables. Also, there may be an unmeasured confounder of A and B. It also implies that B is not a cause of A.
A $<-> $ B. There is an unmeasured confounder (call it L) of A and B. There may be measured variables along the causal pathway from L to A or from L to B. It is also implied that A is not a cause of B and B is not a cause of A.
A o$->$ B. Either A is a cause of B (i.e, A $--> B$) or there is an unmeasured confounder of A and B (i.e, A $<-> $ B) or both. It implies that B is not a cause of A.
A o$-$o B. Exactly one of the following holds: A is a cause of B; B is a cause of A; there is an unmeasured confounder of A and B.

The Structural Equation Model (SEM) expresses, for any cause-effect pair, how much of the effect is explained by a unit variation of the cause. This allows discovering the relations between microservices, in terms of response time, useful for design and testing.

[10] `https://github.com/bd2kccd/py-causal`
[11] `https://www.tetradcausal.app/`

4.3 Use Case for functionality 3

The "use case 3" folder of the delivered package contains the code and data to run the anomaly detection and Root Cause Analysis (RCA) example.

Dataset. *ex_10m_spike_30s_10-150_DT.csv*. This is the dataset used as illustrative example for this use case. It is derived by running a workload, as described below, on a well-known open-source benchmark for microservice architecture (MSA), named Train Ticket[12]. The application simulates a train ticket booking system, composed of 41 microservices communicating to each other via REST over HTTP. Train ticket is polyglot (e.g., Java, golang, Node.js, etc).

The dataset has the following columns:

|`Container1 CPU usage`| ... |`ContainerN CPU usage`|`Container1 Memory usage`| ... |`ContainerN Memory usage`|

Each row is a sample every 5 seconds.

Workload generation. The dataset is generated by stressing the system with a workload. The folder contains the files/scripts to: i) gather monitoring data, ii) parse the raw monitoring data and creating the dataset. The "`locust.py`" file customize the tool Locust[13], which we exploited to generate the load. Locust is a distributed, open-source load testing tool that simulates concurrent users in an application for each benchmark. We customize the workload to reflect the behavior of real users. In total, we run on average 8 requests per second for 10 minutes, with 10 users. The run configuration models a "spike" in which the load is increased to 150 users after 5 minutes for 30 seconds in order to see if any anomaly is detected.

Code. To run anomaly detection on all microservices data, run the `anomaly.R` script, as described below; then, if any anomlay is detected, you can run the `MuTE.py` script to diagnose the possible root cause by assessing any temporal cause-effect relation between microservices (CPU and memory).

Prerequisites:

Python (version >3), JVM/JRE (version > 1.8), Locust (used with version 2.8.6).

Libraries: `Anomalize`[14], `IDTxl`[15]. For `IDTxl`, follow the instructions at `https://github.com/pwollstadt/IDTxl/wiki/Installation-and-Requirements` to install it.

[12] `https://github.com/FudanSELab/train-ticket`
[13] `https://locust.io/`
[14] R code, `https://cran.r-project.org/web/packages/anomalize/`
[15] `https://github.com/pwollstadt/IDTxl`

Commands:

`./anomaly.R <dataset_filename>.csv` #for anomaly detection, output a jpg image

`./MuTE.py <dataset_filename>.csv` #for modelling relation between microservices and the anomaly

The example dataset file is *ex_10m_spike_30s_10-150_DT.csv* located in the "Use Case 3/dataset' folder.

Results. The anomlay detection script generates the graphical representation of possible anomalies detected in any container-level metric; the result of Multivariate Transfer Entropy (MuTE) applied to the same series. The MuTE output are both in textaul form and as a graph (i.e., the network) representing which time series transfer more entropy to which other and at which significance. Given a target microservice that experienced anomalies (in CPU or memory data) and for which you want to diagnose the root cause, the graph tells which other microservice is more likely the cause of the anomaly (i.e., which transfer more entropy, and the corresponding time lag between the time series that maximize the transfer).

4.4 Use Case for functionality 4

The "use case 4" folder of the delivered package contains the code and data to run the feature stability analysis in Just-In-Time Defect Prediction (JIT-DP) example.

Dataset. This folder contains the dataset used as illustrative example for this use case. The dataset consists of commit data for 6 open source applications. It is cloned from a JIT-DP research work repository.[16] As in [1], the generation procedure cosists of data extraction through python scripts, and labelling by the SZZ algorithm.

Each subject dataset has the following columns:

`_id | date | ns | nd | nf | entropy | la | ld | lt | fix | ndev | age | nuc | exp | rexp | sexp`

Each row refers to a commit.

Code. Python code files/scripts to: i) perform defect prediction on a dataset in the same format as the Dataset's folder files; this includes code for training

[16] `https://github.com/ZZR0/ISSTA21-JIT-DP`

the Random Forest and Logistic Regression models. Feature stability is computed according to the algorithm described in [2]. ii) Random Forest algorithm represent the worst case, in which each tree considers different features at each execution; Logistic Regression represents the best case since the selected features (the ones with a positive coefficient) are very often the same.

To reproduce: run the `artifact_RF.py` and `artifact_LR.py` scripts.

Prerequisites: Python (version >3)

Commands:

Run the `python artifact_RF.py` script to run the feature stability on the Random Forest model.
Run the `python artifact_LR.py` script to run the feature stability on the Logistic Regression model.

Results. The output reports the feature importance of each feature for the considered model (RF: Random Forest, LR: Logistic Regression), and the feature stability value computed as in [2]. This can be useful to assess which metrics are the best predictors not only from the accuracy point of view, but from the stability perspective.

References

[1] Zhengran Zeng, Yuqun Zhang, Haotian Zhang, and Lingming Zhang. 2021. Deep just-in-time defect prediction: how far are we? In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021). Association for Computing Machinery, New York, NY, USA, 427–438. https://doi.org/10.1145/3460319.3464819

[2] Sarah Nogueira, Konstantinos Sechidis, and Gavin Brown. 2017. On the stability of feature selection algorithms. J. Mach. Learn. Res. 18, 1 (January 2017), 6345–6398.