Developing Data-Intensive Cloud
Applications with Iterative Quality
Enhancements

Quality anomaly detection and trace checking tools - Initial Version

Deliverable 4.3

Deliverable:	D4.3
Title:	Quality anomaly detection and trace checking tools
Editor(s):	Gabriel Iuhasz (IEAT)
Contributor(s):	Gabriel Iuhasz (IEAT), Ioan Dragan (IEAT), Tatiana Ustinova (IMP), Marcello Bersani (PMI)
Reviewers:	Efstratios Tzoannos (ATC), Simona Bernardi (ZAR)
Type (R/P/DEC):	DEM
Version:	1.0
Date:	29-July-2016
Status:	Final version.
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2016, DICE consortium – All rights reserved

The DICE project (February 2015-January 2018) has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No. 644869

Executive summary

This deliverable documents the anomaly and trace checking tools from the DICE solution. It details the development and architecture of the Anomaly Detection Tool (ADT) from Task 4.2 and that of the Trace Checking (TraCT) from T4.3. The initial versions of the Regression based Anomaly Detection method is also detailed in this deliverable. In the initial versions of these tools the main goal was to create a comprehensive and extensible yet lightweight base on which further advancements related to anomaly detection can be built. We have done this by defining the overall architecture and workflow for ADT as well as TraCT. Furthermore, we also detail a Regression based AD solution that is able to compare and highlight anomalies in different versions of the same application.

The document is structured as follows: the Introduction section highlights the objectives and features of the anomaly detection, trace checking tools as well as that of the Regression based AD method. It also describes the contributions of these tools to DICE objectives and DICE innovation objectives. This is followed by the presentation of the position of the tools inside overall architecture and its interfaces to other DICE tools. The first section also highlights the achievements of the period under report. The second section, Architecture and design of the tool, details the constituent components of each of the tools. The third section connects the DICE Monitoring platform to DICE use cases and requirements identified and presented in deliverable D1.2. Deployment and validation of the tools is tackled in section 4. The last section draws final conclusions and sets the future development plans for DICE ADT and Trace Checking.

Glossary

AD	Anomaly Detection
ADT	Anomaly Detection Tool
DIA	Data Intensive Applications
DICE	Data-Intensive Cloud Applications with iterative quality enhancements
DMon	DICE Monitoring
ELK	Elasticsearch, Logstash and Kibana
IDE	Integrated Development Environment
LM	Log Merger
PMML	Predictive Model Markup Language
TCE	Trace Checking Engine
TraCT	Trace checking
UML	Unified Modelling Language

Contents

Executive summary . . . 3

Glossary . . . 4

Table of Contents . . . 5

List of Figures . . . 6

List of Tables . . . 6

List of Listings . . . 6

1 Introduction . . . 7
- 1.1 Objectives . . . 8
- 1.2 Relation to DICE objectives . . . 8
- 1.3 Relation to DICE Tools . . . 8
- 1.4 Achievements of the period under report . . . 10
- 1.5 Structure of the document . . . 10

2 Architecture and Implementation . . . 11
- 2.1 Anomaly detection tool . . . 11
  - 2.1.1 Big Data framework metrics data . . . 11
  - 2.1.2 Types of anomalies . . . 11
  - 2.1.3 Anomaly detection methods . . . 12
  - 2.1.4 Anomaly detection Implementation . . . 13
- 2.2 Trace checking tool . . . 14
  - 2.2.1 Storm logging . . . 16
  - 2.2.2 DICE-TraCT architecture . . . 18
  - 2.2.3 Trace Checking Engine . . . 18
  - 2.2.4 Log Merger . . . 19
  - 2.2.5 DICE-TraCTor . . . 19
- 2.3 Regression based Anomaly Detection . . . 22
  - 2.3.1 Model Training module . . . 23
  - 2.3.2 Model Analysis module . . . 28
  - 2.3.3 Report generation . . . 31
  - 2.3.4 Discussion . . . 31

3 Use cases . . . 33
- 3.1 Anomaly Detection . . . 33
- 3.2 Trace Checking tool . . . 33
- 3.3 Regression based Anomaly Detection . . . 34

4 Integration and Validation . . . 36
- 4.1 Anomaly Detection . . . 36
- 4.2 Trace Checking tool . . . 37

5 Conclusions . . . 44
- 5.1 Summary . . . 44
- 5.2 Further work . . . 44

References . . . 45

List of Figures

1 Summary view of the project methodology. 7
2 DICE Overall architecture. 10
3 General overview of Anomaly Detection Stack. 14
4 ADT Sequence diagram. 15
5 Deployment in (a) for a Storm topology (b) 17
6 Partion of the ExclamationTopology log. 17
7 Architectural overview shows DICE-TraCT component within the Anomaly Detection tool. 18
8 DICE-TraCT architecure. 18
9 Merging logs w1.log and w2.log into three new log files for each topology node. 19
10 DICE-TraCT sequence diagram highlighting the intraction of all components. 20
11 Regression based Anomaly Detection. Architecture, inputs and interaction with other DICE tools. 25
12 Model training module: architecture and process flow.. 26
13 A figure with Architecture and Flow Diagram 28
14 Visualization of the term by term comparison of model coefficients 30
15 Anomaly Detection flow. 33
16 Examples of configuration files 34
17 Anomaly detection integration with DMON. 36
18 Snapshot of the execution of spark-submit for node *spoutA*. 43

List of Tables

1 Relation to DICE objectives 9
2 Anomaly Detection Tool requirements 15
3 Trace Checking tool requirements 22
4 List of inputs to the model training module. 24
5 Two-level full factorial design for two factors. 26
6 Outcomes and interpretation of the two application models’ term-by-term comparison. . 29
7 Requirements for Regression based AD 32
8 Input parameters for the packaged tool 35

Listings

1 Example of JSON script requesting trace checking analysis for *spoutA* and *boltA* 21
2 JSON job descriptor 37
3 Field ”node” of the JSON script used for validation experiment. 38
4 Json script sent by D-Mon to DICE-TraCT. Each node name is endow with the list of logs where it appears. 39
5 JSON script sent by D-Mon to DICE-TraCT. Each node name is endow with the list of logs where it is appears. 40
6 Event log ready for trace checking built by LM for node *boltA*. 41
7 Descriptor of regular expression used to parse the files. 41
8 Template of the temporal logic formula used in the validation example. 41
9 Temporal logic formula used in the validation example built from the template for node *spoutA*. 42
10 Temporal logic formula used in the validation example built from the template for node *spoutB*. 42
11 Command line to submit a spark job and execute trace checking. 42

1 Introduction

This section will describe the motivation and context for this deliverable. A summary view of the project methodology is shown in the Figure 1:

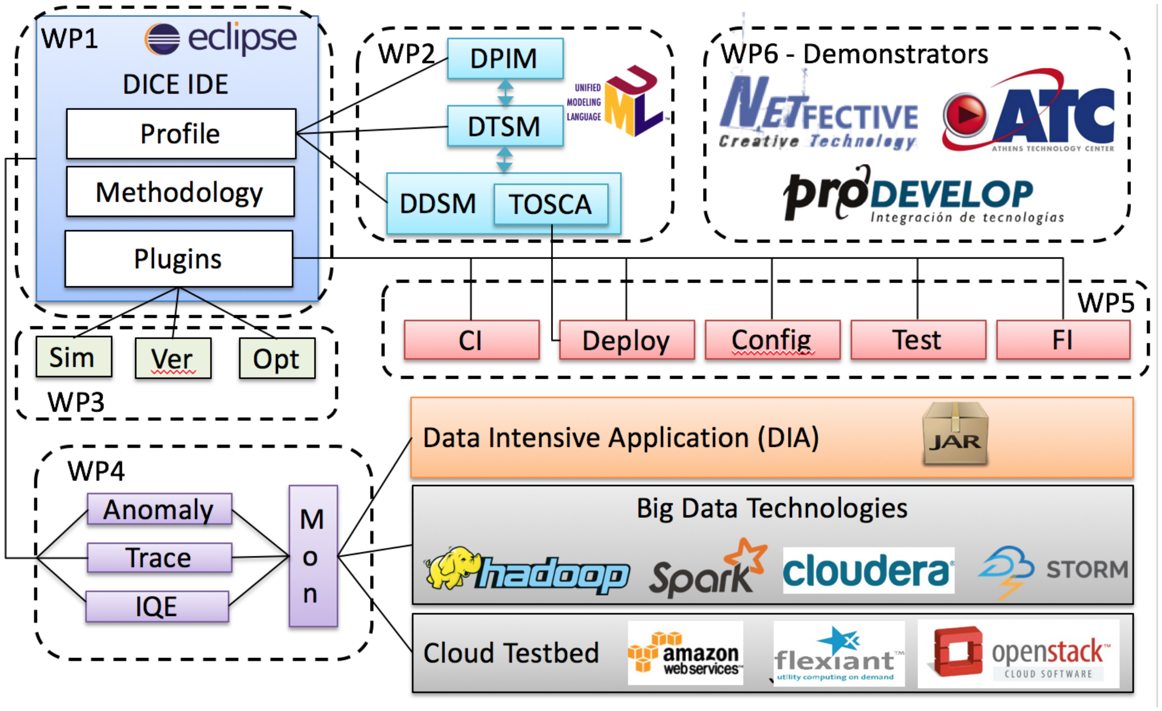


Figure 1: Summary view of the project methodology.

This deliverable presents the initial release of the DICE Anomaly Detection Tool (ADT) and Trace checking tool (DICE-TraCT) whose main goals are to enable the definition and detection of anomalous measurements from Big Data frameworks such as Apache Hadoop, Spark or Storm. Both tools are developed in WP4, more specifically the ADT is developed in T4.2 ”Quality incident detection and analysis” while the DICE-TraCT tool in T4.3 ”Feedbacks for iterative quality enhancement”. We can see that these tools are represented in Figure 1 and are responsible for signalling anomalous behaviour based on measured metrics (ADT and Regression based AD) and on framework logs (DICE-TraCT).

The main objectives of these tools are to detect anomalies, in particular contextual anomalies. DICE-TraCT on the other hand will be used for detecting sequential anomalies. Also, the creation of a lambda architecture when combining ADT with DMon. Lastly, the definition of a new Regression based anomaly detection method.

Main features of the anomaly detection are:

- Integration with several open source machine learning frameworks
- Trace checking capabilities for Apache Storm
- Regression based anomaly detection
- Integration with DMon [13]
- Ability to train and validate supervised predictive models

The remaining of this section presents the positioning of ADT and Trace checking tool relative to DICE innovation objectives, DICE objectives and relation to other tools from DICE tool-chain.

1.1 Objectives

The focus of the DICE project is to define a quality-driven framework for developing data-intensive applications that leverage Big Data technologies hosted in private or public clouds. DICE will offer a novel profile and tools for data-aware quality-driven development. The methodology will excel for its quality assessment, architecture enhancement, agile delivery and continuous testing and deployment, relying on principles from the emerging DevOps paradigm. The DICE anomaly detection and trace checking tools contribute to all core innovations of DICE, as follows:

I1: *Tackling skill shortage and steep learning curves in quality-driven development of data- intensive software through open source tools, models, methods and methodologies.*

ADT and Regression based AD will enable the detection and alerting of anomalous behaviour during data intensive application development. DICE-TraCT on the other hand will deal with sequential anomalies identified from log data. This will help identify quality related anomalies and signal these, in essence making the debugging and identification of performance bottlenecks much easier.

I2: *Shortening the time to market for data-intensive applications that meet quality requirements, thus reducing costs for ISVs while at the same time increasing value for end-users.*

Several tools and actors profit from the information (anomalies) signalled by ADT and DICE-TraCT, thus using the detected anomalies in their initial setup.

I3: *Decreasing costs to develop and operate data-intensive cloud applications, by defining algorithms and quality reasoning techniques to select optimal architectures, especially in the early development stages, and taking into account SLAs.*

By detecting quality and performance related anomalies operational costs can be reduced by the optimized version of the application. At the same time other tools may use the detected anomalies to provide feedback to the end user/developer and the output of these optimization tools can provide significant financial and performance advantages.

I4: *Reducing the number and severity of quality-related incidents and failures by leveraging DevOps-inspired methods and traditional reliability and safety assessment to iteratively learn application runtime behaviour.*

Runtime application behaviour is collected by DMon which is then used as a data source for ADT permitting the timely detection of quality-related incidents.

1.2 Relation to DICE objectives

The following table 1 highlights the contributions of ADT and Trace checking tool to DICE objectives.

1.3 Relation to DICE Tools

Figure 2 illustrates the interfaces between the ADT (marked with red) and the rest of the DICE solution. The main goal of ADT is to detect inconsistencies at runtime and on historical data for jobs and services in data intensive applications. It is meant to provide a powerful but still light weight solution for both developers, architects and software engineers.

As mentioned in the deliverable D4.1 [13], there exists a tight integration between DMon and ADT as these two tools will for the basis of a lambda type architecture. DMon is the serving layer while instances of ADT can take the role of both speed and batch layers.

Other tools that make us of ADT are: Fault Injection, Quality Testing and IDE. The fault injection tool is able to produce system level anomalies which can be used by ADT for the creation of training/-validation datasets. Quality testing tool will use the detected anomalies while the IDE will permit the

Table 1: Relation to DICE objectives

DICE Objective Description	Relation to Anomaly Detection tools
DICE profile and methodology, Define a data-aware profile and a data-aware methodology for model-driven development of data-intensive cloud applications. The profile will include data abstractions (e.g., data flow path synchronization), quality annotations (e.g., data flow rates) to express requirements on reliability, efficiency and safety (e.g., execution time or availability constraints).	None Future versions may be used to enable the ad-hoc creation of feature vectors from the DICE Profile
Quality analysis tools, Define a quality analysis tool-chain to support quality related decision-making through simulation and formal verification.	The quality testing and Delivery tool [6] will be able to use detected anomalies.
Quality enhancement tools, An approach leveraging on DevOps tools to iteratively refine architecture design and deployment by assimilating novel data from the runtime, feed this information to the design time and continuously redeploy an updated application configuration to the target environment.	Enhancement tools may use the detected anomalies to further streamline it's input data.
Deployment and testing tools, Define a deployment and testing toolset to accelerate delivery of the application.	The initial versions of the tools described in this deliverable use simple bash scripts for installation. Future versions will most likely be deployed using the WP5 Deployment Service
IDE, Release an Integrated Development Environment (IDE) to simplify adoption of the DICE methodology.	ADT will be controllable from the IDE, meaning that query definition.
Open-source software, Release the DICE tool-chain as open source software.	ADT as well as Regression based AD and TraCT rely heavily on open source technologies.
Demonstrators, Validate DICE productivity gains across industrial domains through 3 use cases on data-intensive applications for media, e-government, and maritime operations.	All demonstrators will use the ADT, in particular the ATC usecase will use the TraCT tool for their Storm based application
Dissemination, communication, collaboration and standardisation, Promote visibility and adoption of project results through dissemination, communication, collaboration with other EU projects and standardisation activities.	All tools have been or will be presented in both scientific publications as well as Big Data innovation events.
Long-term tool maintenance beyond life of project.,The project coordinator (IMP) will lead maintenance of tools, project website and user community beyond DICE project lifespan.	Monitoring platform source code and homepage are stored using, Github as a public open-source software. Community is welcome to contribute,to the platform, during and after DICE end.

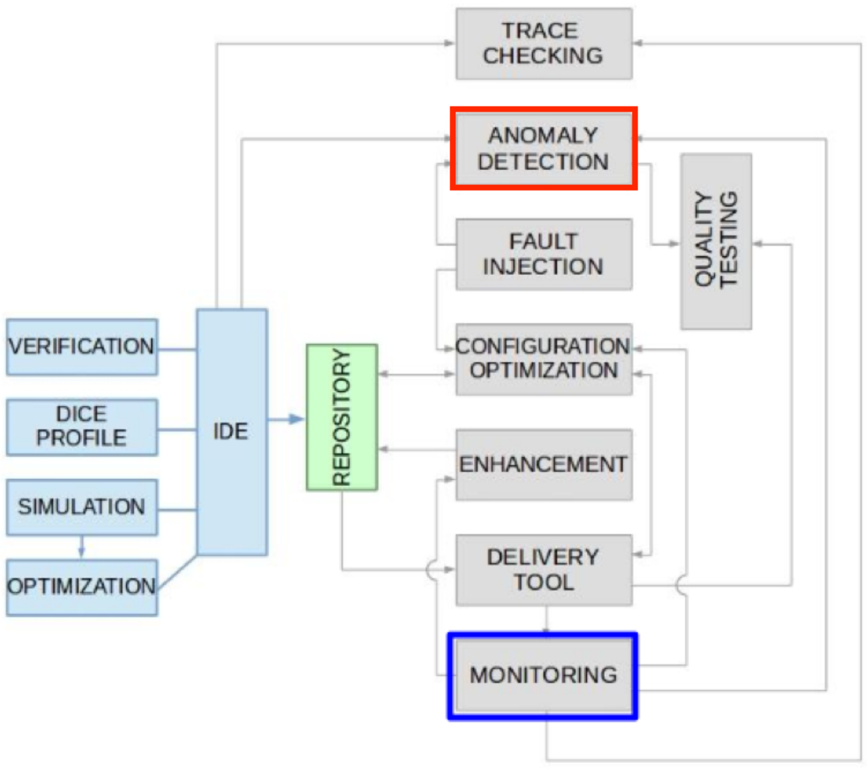


Figure 2: DICE Overall architecture.

interaction of actors with ADT, creating custom feature vectors, defining roles etc.

1.4 Achievements of the period under report

Overview of the main achievements in the reported period:

- Definition of ADT, Trace Checking and Regression based AD architectures
- Initial validation of the Trace Checking and Regression based AD
- Definition of the *dmon-gen* tool for the semi-automated creation of labelled data for anomaly detection
- Outline of future work and integration

1.5 Structure of the document

The structure of this deliverable is as follows:

- Section 2 the architecture and implementation details of the anomaly detection, Trace Checking and Regression based Anomaly Detection tools and methods
- Section 3 gives some details related to the use cases for the tools
- Section 4 presents details on initial integration and validation of these tools
- Section 5 gives conclusions and outlines future work

2 Architecture and Implementation

The following section will detail the overall architecture, implementation as well as the requirement coverage of each tool. IT Also covers the main rationale behind the necessity of each tool as well as the interaction between them and the overall DICE toolchain.

2.1 Anomaly detection tool

Anomaly Detection is an important component involved in performance analysis of data intensive applications. We define an anomaly as an observation that does not conform to an expected pattern [7, 8]. Most tools or solutions such as Sematex[1], Datadog[2] etc. are geared more towards a production environment in contrast to this the DICE Anomaly Detection Tool (ADT) is designed to be used during the development phases of big data applications.

2.1.1 Big Data framework metrics data

In DICE most data preprocessing activities will be done within DMon [13]. However, some additional preprocessing such as normalisation or filtering will have to be applied at method level.

In anomaly detection the nature of the data is a key issue. There can be different types of data such as: binary, categorical or continuous. In DICE we deal mostly with the continuous data type although categorical or even binary values could be present. Most metrics data relate to computational resource consumption, execution time etc.. There can be instances of categorical data that denotes the status/state of a certain job or even binary data in the form of boolean values. This makes the creation of data sets on which to run anomaly detection an extremely crucial aspect of ADT, because some anomaly detection methods don't work on categorical or binary attributes.

It is important to note that most, if not all, anomaly detection techniques and tools, deal with point data, meaning that no relationship is assumed between data instances [8]. In some instances this assumption is not valid as there can be spatial, temporal or even sequential relationships between data instances. This in fact is the assumption we are basing ADT on with regard to the DICE context.

All data in which the anomaly detection techniques will use are queried from DMon. This means that some basic statistical operations (such as aggregations, median etc.) can already be integrated into the DMon query. In some instances this can reduce the dataset size in which to run anomaly detection.

2.1.2 Types of anomalies

An extremely important aspect of detecting anomalies in any problem domain is the definition of the anomaly types that can be handled by the proposed method or tool. In the next paragraphs we will give a short definition of the classification of anomalies in relation to the DICE context.

First we have point anomalies which are the simplest types of anomalies, represented by data instances that can be considered anomalous with respect to the rest of the data [7]. Because this type of anomaly is simple to define and check a big part of research effort will be directed towards finding them. We intend to further investigate this type of anomalies and consider them for inclusion in DICE ADT. However, as there are a lot of existing solutions already on the market this will not be the main focus of ADT instead we will use the Watcher[3] solution from the ELK stack to detect point anomalies.

A more interesting type of anomalies in relation with DICE are the so called contextual anomalies. These are considered anomalous only in a certain context and not otherwise. The context is a result of the structure from the data set. Thus, it has to be specified as part of the problem formulation [19, 7]. When defining the context we consider; contextual attributes which are represented by the neighbours of each instance and behavioural attributes which describe the value itself. In short anomalous behaviour is determined using the values for the behavioural attributes from within the specified context [7]. In DICE most data is time-series data which is the most common type of data in which contextual anomalies can

[1]https://sematext.com/spm/
[2]https://www.datadoghq.com/
[3]https://www.elastic.co/guide/en/watcher/current/index.html

occur. Also, the meaningfulness of contextual anomalies is heavily dependant of the target application domain. Because of this in the context of our tool we must have a set of anomalies for each of the Big Data services covered in DICE project. In this deliverable the main focus is on creating a basic framework that enables ad-hoc definition of context rather than an exhaustive list of predefined ones. Future work will also feature some instances of these predefined contexts and anomalies.

The last types of anomalies are called collective anomalies. These anomalies can occur when a collection of related data instances are anomalous with respect to the entire data set. In other words, individual data instances are not anomalous by themselves. Typically collective anomalies are related to sequence data and can only occur if data instances are related. In ADT these types of anomalies will be handled by the Trace Checking tool (See Section 2.2).

2.1.3 Anomaly detection methods

There are a wide range of anomaly detection methods currently in use [7]. These can be split up into two distinct categories based on how they are trained. First there are the methods used in supervised methods. In essence these can be considered as classification problems in which the goal is to train a categorical classifier that is able to output a hypothesis about the anomality of any given data instances. These classifiers can be trained to distinguish between normal and anomalous data instances in a given feature space. These methods do not make assumptions about the generative distribution of the event data, they are purely data driven. Because of this the quality of the data is extremely important.

For supervised learning methods labelled anomalies from application data instances are a prerequisite. False positives frequency is high in some instances, this can be mitigated by comprehensive validation/testing. Computational complexity of validation and testing can be substantial and represents a significant challenge which has been taken into consideration during in the ADT tool. Method used for supervised anomaly detection include but are note limited to: Neural Networks, Neural Trees, ART1, Radial Basis Function, SVM, Association Rules and Deep Learning based techniques.

In unsupervised anomaly detection methods the base assumption is that normal data instances are grouped in a cluster in the data while anomalies don't belong to any cluster. This assumption is used in most clustering based methods [16, 17] such as: DEBSCAN, ROCK, SNN FindOut, WaveCluster. The second assumption [7, 20] on which K-Means, SOM, Expectation Maximization (EM) algorithms are based is that normal data instances belong to large and dense clusters while anomalies in small and spars ones. It is easy to see that the effectiveness of each of unsupervised, or clustering based, method is largely based in the effectiveness of individual algorithms in capturing the structure of normal data instances.

It is important to note that these types of methods are not designed with anomaly detection in mind. The detection of anomalies is more often than not a by product of clustering based techniques. Also, the computational complexity in the case of clustering based techniques can be a serious issue and careful selection of the distance measure used is a key factor.

Anomaly detection libraries

In recent years there have been a great deal of general machine learning frameworks developed. These can deal with a wide range of problems. One of the problem domains that can be tackled using them is that of anomaly detection. It is important to mention that we will use not only bespoked anomaly detection libraries/methods but also more general supervised (i.e. classification based) and unsupervised (i.e. clustering based) techniques in ADT. In Figure 3 we have a short overview of the core libraries in the current version of ADT. For the sake of completeness we will briefly describe the machine learning libraries used, and the rationale behind using them in ADT.

Scikit-learn [18] is a Python based open source machine learning library. Some of its core algorithms are written in Cython to achieve performance. It has Cython wrappers around LIBSVM and LIBLINEAR as well as using the Numpy and Scipy modules. It is a general purpose library with minimal dependencies. It features various classification and clustering methods including DEBSCAN, k-means and SVM.

Weka [11] is a well known data mining and machine learning library and workbench. It supports several data mining tasks such as; preprocessing, clustering, classification, regression and feature selection. It is arguably the most utilized open source machine learning/data mining tool. We leverage the existing methods from Weka and use them in ADT to further our goal of detecting contextual anomalies in data intensive applications which utilize Big Data services.

TensorFlow [2] is an open source library designed for numerical computation using data flow graphs extensively used in deep learning applications. It's architecture enables the use of one or more CPUs or GPGPUs. It was originally developed by the Google Brain[4] Tream. In the DICE context we will use some of the features and neural network implementations from Tensorflow in order to detect contextual anomalies. It is able to export predictive models using the TensorServing[5] module means that pre-trained models can be easily exploited in a production type environment.

ELKI [20] is a Java based open source data mining software. It aims at implementation and development of various unsupervised methods for cluster analysis and outlier detection. Performance of the tool is guaranteed due to the performant indexing structures that are implemented as core of the software (e.g R*-tree). The base philosophy that drives development of ELKI is use of highly parameterizable algorithms in order to allow ease of customization and benchmarking. ELKI's architecture is modular with most algorithms based on various distance functions and neighbourhood definitions. All its functionalities can be used either through the minimalistic GUI that is provided or via command line. Extensions to the currently implemented algorithms are done by implementing various APIs. Specific APIs are provided in order to further enhance the power of ELKI with bespoked algorithms. In terms of output, it provides a variety of writers that handle a big palette of standard formats as well as a complex visualization tool, providing support for SVG rendering for high quality graphics.

All of the above mentioned tools will be integrated as submodules inside ADT. Currently we have working versions of most of these tools inside ADT with the exception of ELKI methods. In the next version of ADT validation and a more complete integration will be presented.

2.1.4 Anomaly detection Implementation

The ADT is made up of a series of interconnected components that are controlled from a simple command line interface. This interface is meant to be used only for the initial version of the tool. Future versions will feature a more user friendly interface.

In total there are 8 components that make up ADT. The general architecture can be seen in Figure 3 These are meant to encompass each of the main functionalities and requirements identified in the requirements deliverables [9].

First we have the *dmon-connector* component which is used to connect to DMon. It is able to query the monitoring platform and also send it new data. This data can be detected anomalies or learned models. For each of these types of data *dmon-connector* creates a different index inside DMon. For anomalies it creates an index of the form **anomaly-UTC** where UTC stands for Unix time. Similarly to how the monitoring platform deals with metrics and their indices. Meaning that the index is rotated every 24 hours.

After the monitoring platform is queried the resulting dataset can be in JSON, CSV or RDF/XML. However., in some situations some additional formatting is required. This is done by the *data formatter* component. It is able to normalize the data, filter different features from the dataset or even window the data. The type of formatting the dataset may or may not need is highly dependant on the anomaly detection method used.

The *feature selection* component is used to reduce the dimensionality of the dataset. Not all features of a dataset may be needed to train a predictive model for anomaly detection. So in some situations it is important to have a mechanism that allows the selection of only the features that have a significant impact on the performance of the anomaly detection methods. Currently only two types of feature selection is supported. The first is *Principal Component Analysis*[6] (from Weka) and *Wrapper Methods*.

[4]https://research.google.com/teams/brain/

[5]https://tensorflow.github.io/serving/

[6]http://weka.sourceforge.net/doc.dev/weka/attributeSelection/PrincipalComponents.html

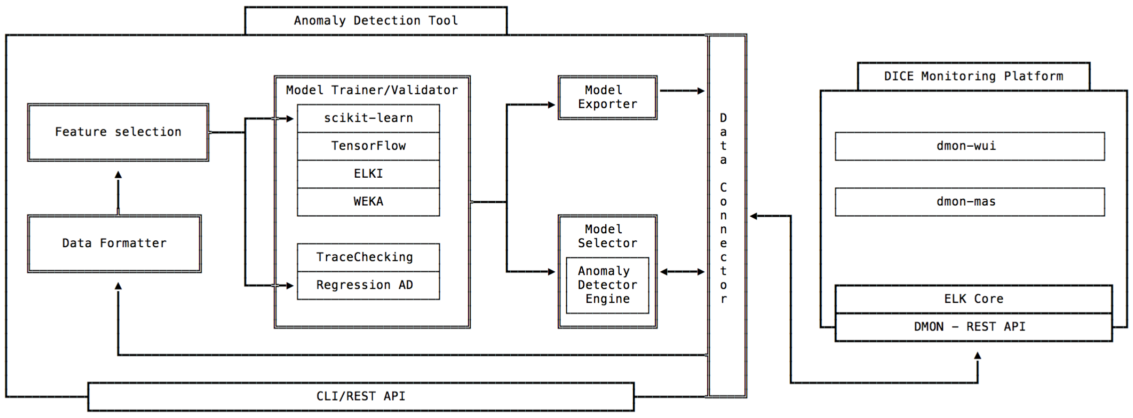


Figure 3: General overview of Anomaly Detection Stack.

The next two components (see Figure 3) are used for training and then validating predictive models for anomaly detection. For training a user must first select the type of method desired. The dataset is then split up into training and validation subsets and later used for cross validation. The ratio of validation to training size can be set during this phase. Parameters related to each method can also be set in this component.

Validation is handled by a specialized component which minimizes the risk of overfiting the model as well as ensuring that out of sample performance is adequate. It does this by using cross validation and comparing the performance of the current model with past ones.

Once validation is complete the *model exporter* component transforms the current model into a serialized loadable form. We will use the PMML [10] format wherever possible in order to ensure compatibility with as many machine learning frameworks as possible. This will also make the use of ADT in a production like environment much easier.

The resulting model can be fed into DMon. In fact the core services from DMon (specifically Elasticsearch) have to role of a serving layer from a lambda architecture. Both detected anomalies and trained models are stored in the DMon and can be queried directly from the monitoring platform. In essence this means that other tools from the DICE toolchain need to know only the DMon endpoint in order to see what anomalies have been detected.

Furthermore, the training and validation scenarios (see Figure 17) is in fact the batch layer while unsupervised methods and/or loaded predictive models are the speed layer. Both these scenarios can be accomplished by ADT. This integration will be further detailed in later sections.

The last component is the *anomaly detection engine*. It is responsible for detecting anomalies. It is important to note the it is able to detect anomalies however it is unable to communicate them to the serving layer (i.e. DMon). It uses the *dmon-connector* component to accomplish this. The *anomaly detection engine* is also able to handle unsupervised learning methods. We can see this in Figure 3 in that the Anomaly detection engine is in some ways a subcomponent of the *model selector* which select both pre-trained predictive models and unsupervised methods.

We can see in Figure 4 the sequence diagram for ADT and DMon. It is clearly observable that both anomalies and predictive models are served and stored inside DMon.

Requirements

In table 2 we can see the current status of the requirements related to ADT. Requirements marked with an x are still to be started while the other ones are either started (grey) or fully operational (black).

2.2 Trace checking tool

DICE Trace checking tool (DICE-TraCT) performs trace checking in the D-mon platform.

Trace checking is an approach for the analysis of system executions that are recorded as sequences of timestamped events. Collected logs are analyzed to establish whether the system logs satisfy a property,

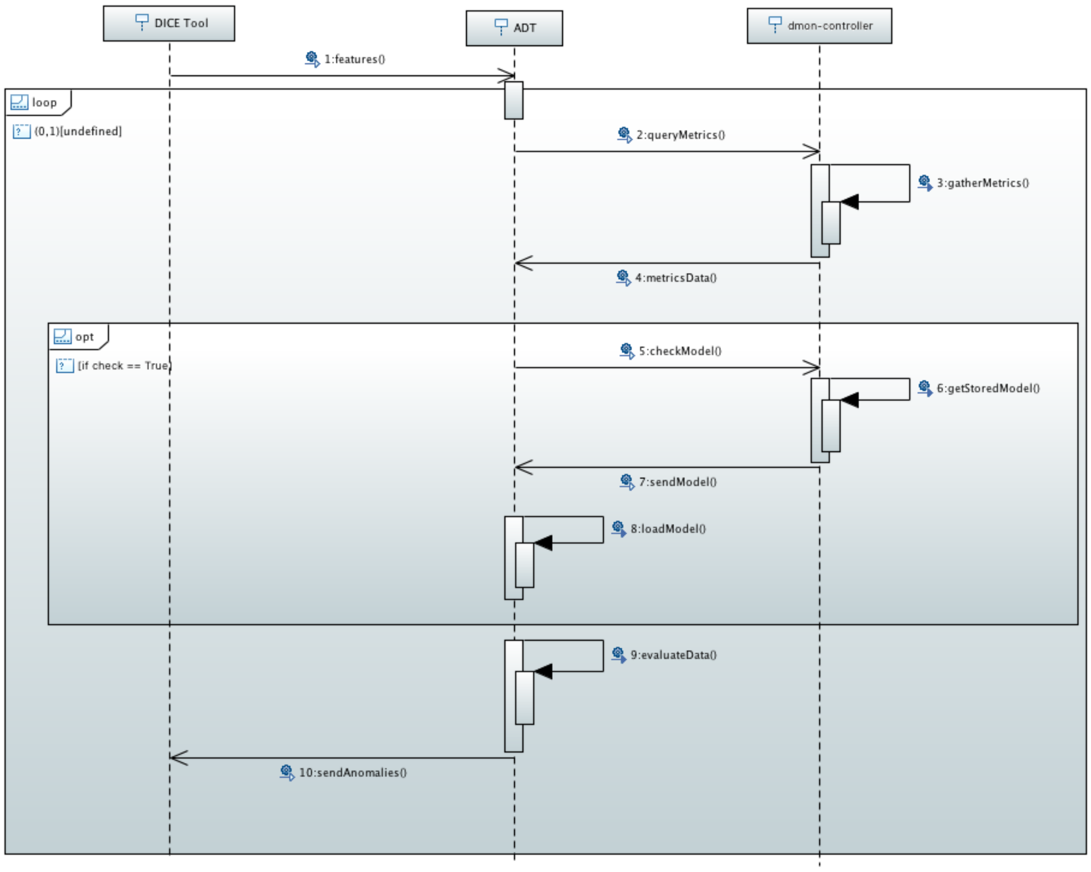


Figure 4: ADT Sequence diagram.

Table 2: Anomaly Detection Tool requirements

ID	**Title**	Priority	Status	Comments
R4.24	Anomaly detection in app quality	MUST	✓	
R4.24.1	Unsupervised Anomaly Detection	MUST	✓	ADT is capable of running clustering based methods
R4.24.2	Supervised Anomaly Detection	MUST	✓	ADT is able to query and generate datasets for training and validation.
R4.24.3	Contextual Anomalies	Should	✓	Is possible to define feature vectors that define context.
R4.24.4	Collective anomalies	Should	✗	
R4.24.5	Predictive Model saving for AD	MUST	✓	Is capable of generating PMML or serialized predictive models
R4.24.6	Semi-automated data labelling	Could	✓	Can be don via dmon-gen component.
R4.24.7	Adaptation of thresholding	Could	✗	
R4.26.2	Report generation of analysis results	Should	✓	Local generation of report is possible.
R4.36	AD between two versions of DIA	MUST	✗	
R4.37	ADT should get input parameters from IDE	MUST	✗	

usually specified in a logical language; in the positive case, the sampled system behavior conforms with the constraints modeled by the property.

When the language allows temporal operators, trace checking is a way to check the correctness of the ordering of the events occurring in the system and of the time delays between pairs of events. For instance, if property requires that *all the emit events of a certain bolt occur not more than ten millisecond after the latest receive event*, then checking the property over a trace results in a boolean outcome which is positive if the distance between two consecutive and ordered pair of emit and receive events is less than ten milliseconds.

Trace checking is especially useful when the aggregated data that are available from the monitoring system are not enough to conclude the correctness of the system executions with respect to some specific criteria. In some cases, in fact, these criteria are application dependent as they are related to some non-functional property of the application itself and they do not depend on the physical infrastructure where the application is executed. Trace checking is a possible technique to achieve this goal and can be used on purpose to extract information from the executions of a running application.

Logical languages involved in the trace checking analysis are usually extensions of metric temporal logics which offer special operators called aggregating modalities. These operators hold if the trace satisfies particular quantitative features like, for instance, a specific counting property of events in the trace. DICE-TraCT uses Soloist [3] which offers the following class of aggregating modalities:

- number of occurrences of an event *e* in a time window of length *d*,
- maximum/average number of occurrences of an event *e*, aggregated over right-aligned adjacent non-overlapping subintervals (of size *h*) in a time window of length *d*,
- average time elapsed between a pair of specific adjacent and alternating events *e* and *e'* occurring in a time window of length *d*.

According to the DICE vision, trace checking is performed after verification to allow for continuous model refinement. The result obtained through the log analysis confirms or refutes the outcome of the verification task, which is run at design time. The value of the parameters in the design-time model is compared with the value at runtime; if the two are "compatible" then the results of verification are valid, otherwise the model must be refined. For a complete description of the model and the parameters for verification see "DICE Verification Tool – Initial version" [15].

Since verification of DICE models, reported in "DICE Verification Tool – Initial version" [15], deals only with Storm applications, DICE-TraCT currently supports Storm logs analysis. Next releases (M24-M36) will consider other big-data technologies and relevant properties to monitor.

2.2.1 Storm logging

The initial version of DICE-TraCT is implemented for Storm logs analysis as its functionality is currently tailored to the manipulation of collected logs in a Storm deployed topology. This paragraph briefly introduces some features of the logging mechanism implemented in Storm.

Storm topologies are defined by graphs of computational nodes. "Spouts" are the data sources of a topology and always produce messages - or tuples - that are elaborated by the bolts. Bolts receive tuples from one or more nodes and send their outcome to other bolts, unless they are final. In this last case, a bolt does not emit tuples.

The deployment of a topology in a cluster of physical machines is realized automatically by Storm. A topology is organized by means of workers that are Java virtual machines running on some physical machine of the cluster. A worker executes a subset of a topology and one or more workers implement the whole topology. For each physical machine there is at least one active worker and within each worker at least one executor runs. An executor is a thread that is spawned by a worker and runs one or more tasks which actually perform the data processing. Defining a topology requires the user to specify the number of workers for the topology, the number of executors and tasks for each spout and bolt.

Figure 5 shows an example of a running topology. The running configuration, depicted in (a), is a

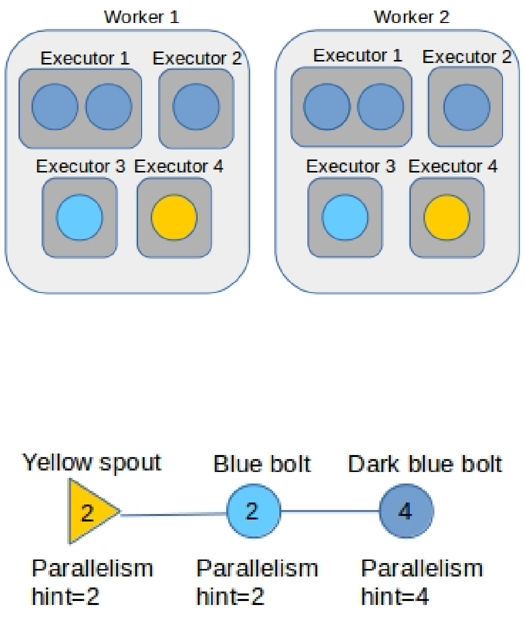


Figure 5: Deployment in (a) for a Storm topology (b)

possible deployment for the topology depicted in subfigure (b) when 2 workers are chosen (light gray boxes). Since the total number of parallelism is 8 then each worker runs 4 executors (dark gray boxes). Circles are the tasks that Storm instantiate to execute the node functionality. The final bolt is executed with 6 tasks.

Storm logs contain messages related to events occurred in the topology. The relevant ones, supporting the verification the task of WP3, are the emit and receive events for the bolts and emit for the spouts. An example of such logs is provided in Fig. 6. The topology producing that log is called *Exclamation-Topology* and can be found in the Storm distribution.

```
2016-02-18T10:52:04.052+0000 [Thread-15-word] b.s.d.task [INFO] Emitting: word default [mike]
2016-02-18T10:52:04.050+0000 [Thread-2-exclaim1] b.s.d.task [INFO] Emitting: exclaim1 default [bertels!!!]
2016-02-18T10:52:04.147+0000 [Thread-2-exclaim1] b.s.d.executor [INFO]
                             Processing received message source: word:16, stream: default, id: {}, [nathan]
2016-02-18T10:52:04.147+0000 [Thread-2-exclaim1] b.s.d.task [INFO] Emitting: exclaim1 default [nathan!!!]
2016-02-18T10:52:04.149+0000 [Thread-16-word] b.s.d.task [INFO] Emitting: word default [jackson]
2016-02-18T10:52:04.149+0000 [Thread-16-word] b.s.d.task [INFO] Emitting: word default [mike]
2016-02-18T10:52:04.149+0000 [Thread-15-word] b.s.d.task [INFO] Emitting: word default [jackson]
2016-02-18T10:52:04.149+0000 [Thread-2-exclaim1] b.s.d.executor [INFO]
                             Processing received message source: word:15, stream: default, id: {}, [mike]
2016-02-18T10:52:04.149+0000 [Thread-2-exclaim1] b.s.d.task [INFO] Emitting: exclaim1 default [mike!!!]
```

Figure 6: Partion of the ExclamationTopology log.

The portion of log in Fig. 6 contains events of spout *word* and bolt *exclaim1*. Each line reports:

- the timestamp measured in milliseconds (e.g., `2016-02-18T10:52:04.052+0000`)
- the thread and node name (e.g., `[Thread-1-word]`)
- the class of the node which performs the action (e.g., `b.s.d.task`)
- the event triggered by a node.
  - spout: the log shows the event, the spout name, the stream name and the emitted tuple (e.g., `Emitting: word default [mike]`)

– bolt: the log shows the actual event along with the source node and thread id (e.g., Processing received message source: word:16), the stream name (e.g., default) and, finally, the id tuple and the tuple itself (e.g., `d: {}, [nathan]`. If id is empty when the topology is run without reliable message processing).

2.2.2 DICE-TraCT architecture

DICE Trace checking tool (DICE-TraCT) is the module which performs trace checking in DICE. It is designed as a component of the anomaly detection tool so that it exploits a direct access to the D-mon APIs and to the DICE-IDE through the API exported by the anomaly detection service. DICE-TraCT collects user requests from the DICE-IDE and, based on the information retrieved through the queries sent to the D-mon platform, executes one or more instances of trace checking. The DICE-IDE allows the user to select a property to verify for a selected DIA application, currently shown in the IDE, and run the trace checking. The input format for DICE-TraCT is a JSON file which contains the name of the topology to verify, the set of nodes that the user wants to analyze and the property to verify. The current version of the tool does not support user-defined properties but only those related to parameter of the verification model.

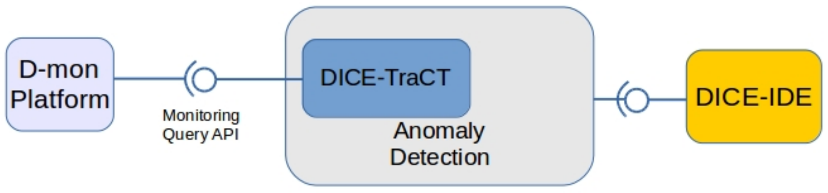


Figure 7: Architectural overview shows DICE-TraCT component within the Anomaly Detection tool.

The following section provides a detailed description of all the components implementing DICE-TraCT which is composed of three components. The architecture is depicted in Fig. 8.

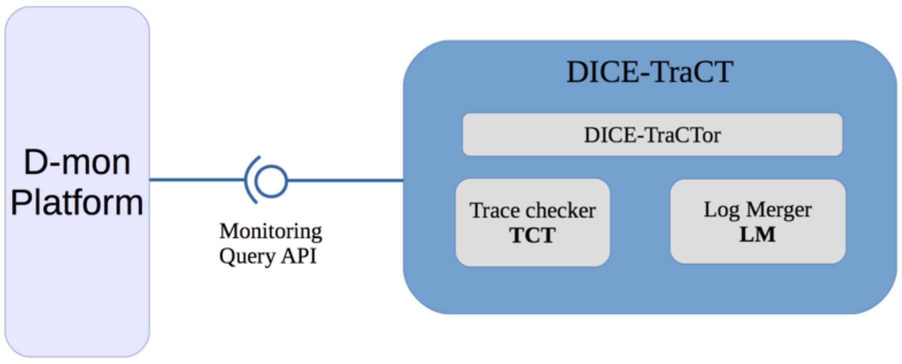


Figure 8: DICE-TraCT architecure.

2.2.3 Trace Checking Engine

Trace Checking Engine (TCE) is the engine that actually performs the trace analysis. It is implemented in Spark and takes advantage of the distributed implementation to realize a parallel algorithm for evaluating temporal formulae over the logs. The input is a time stamped log of events and a Soloist [3] formula. The output is a boolean outcome which is the result of the evaluation of the formula over the specified log. The positive outcome is obtained if the log satisfies the property.

2.2.4 Log Merger

As explained in Sec. 2.2.1, a worker log might contain more than one sequence of events, each one associated with an executor spawned in that worker. A topology node, either spout or bolt, might then be deployed over different workers and the information related to a single node, either spout or bolt, may be spread over many log files.

However, TCE can analyze one log file per execution. To check a property for a node running in different workers, the property has to be tested over many logs through independent trace checking instances. To reduce the number of the executions while leveraging on the distributed implementation of the trace checking engine, DICE-TraCT manipulates the collected logs to aggregate all the events related to a node (or a subset of nodes) into a new log trace. For instance, the running topology of Fig. 5 would produce two logs, one for "Worker 1" and one for "Worker 2". The Log Merger splits the events of the three nodes into three new log files, as shown in the next Fig. 9

LM receives in input a set of worker logs of a deployed topology under monitoring and a description of the topology listing all its computational nodes. The outcome it produces is a set of logs where each log records all, and only, the events related to a certain node in the topology.

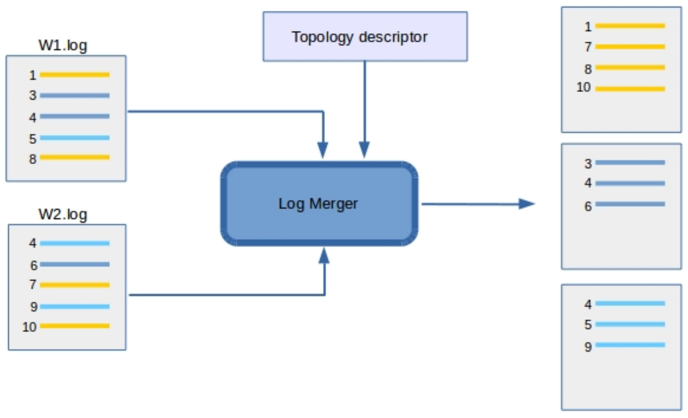


Figure 9: Merging logs w1.log and w2.log into three new log files for each topology node.

2.2.5 DICE-TraCTor

DICE-TraCTor (Tor) coordinates the activity of LM and TCE upon a request from the DICE-IDE. First it builds the inputs file to run LM, runs suitable transformations on the new extracted logs, if they are needed to run trace checking, and then defines the input file of the property for TCE. Finally, it runs TCE and, when TCE terminates, it notify the D-mon platform with the outcome of the analysis. Fig. 7 shows the sequence digram of the interactions of components in DICE-TraCT.

The message invocations among the components described in the diagram are described in the following:

- *monitor(t,p,f.w)*: run the trace checking for the topology described in *t*, for parameter *p* or with formula *f* over time window *w*.
- *merge(l,logs)*: merge log files based on the topology description in *t*.
- *transform(nodeLogs)*: prepare logs for trace checking. This function might change, add or delete event names based on what to check (defined by *p* or *f*)
- *buildProperty()*: define the property to be checked based on topology description *t* and time window *w*.
- *runTC(nodeLog,pr)*: run trace checking for nodeLog with property *pr*.

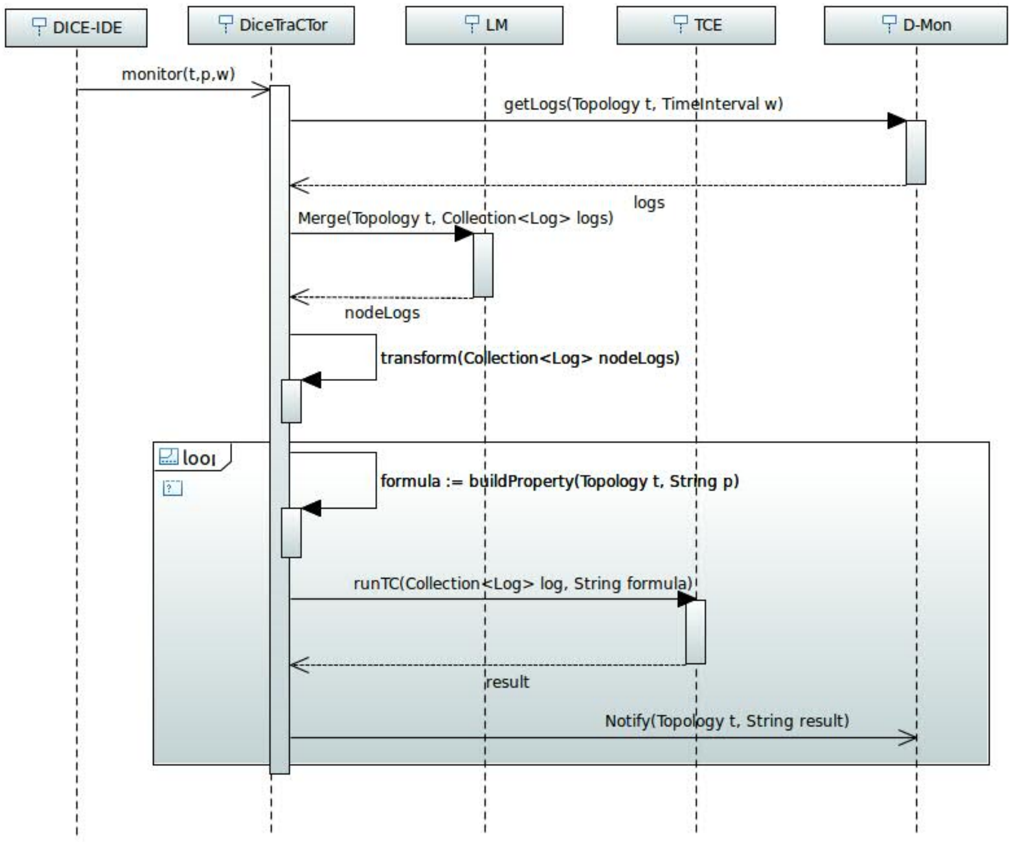

DICE-IDE
DiceTraCTor
LM
TCE
D-Mon
monitor(t,p,w)
getLogs(Topology t, TimeInterval w)
logs
Merge(Topology t, Collection<Log> logs)
nodeLogs
transform(Collection<Log> nodeLogs)
loop
formula := buildProperty(Topology t, String p)
runTC(Collection<Log> log, String formula)
result
Notify(Topology t, String result)

Figure 10: DICE-TraCT sequence diagram highlighting the intraction of all components.

- *notify()*: notify DMon with the result.

Tor receives in input a descriptor which defines the parameters to run trace checking for a given topology. The descriptor is a JSON file that is built through the DICE-IDE by the user who monitors the topology. An example of a descriptor is shown in listing 1. The information stored into JSON fields are the following:

- The field called *topologyname* specifies the topology name which is used by DICE-TraCT to query the monitoring platform and obtain all the necessary log files to perform trace checking.
- A list of descriptors that specify, for each node, a non-functional property to check. The properties can be related to parameters of the verification model like, for instance, the ratio *sigma* between the number of messages in input and the number of messages in output of a Storm node; or any user-defined property which can be translated by DICE-TraCT into a trace checking instance. Each item defined by curly brackets defines a list of data that are needed to collect the suitable set of logs from D-mon and to define the temporal logical formula specifying the property to check. DICE-TraCT is equipped with standard handlers which are able to manipulate predefined properties (like the one related to *sigma*) but allows for defining new custom handlers based on user needs.
- A list of formulae descriptors that specify user-defined logical formulae to be used for trace-checking.

Requirements

This paragraph reports on the achievements obtained for the trace checking tool. Table 3 provides the most relevant requirements and shows the degree of completion.

Listing 1: Example of JSON script requesting trace checking analysis for *spoutA* and *boltA*

```
{
        "topologyname": "ATopology",
         "nodes": [
                        {
                           "name": "spoutA",
                           "type": "spout",
                           "parameter": "idleTime",
                           "timewindow": 3600,
                           "inputrate": 100,
                           "method": "",
                           "relation": "",
                           "designvalue": 0.0
                        },
                        {
                           "name": "boltA",
                           "type": "bolt",
                           "parameter": "sigma",
                           "timewindow": 3600,
                           "inputrate": 100,
                           "method": "counting",
                               "relation": "=",
                                   "designvalue": 1.1
                              }
                           ]
         "formulae": [
                           {
                              ...
                           }
                            ]
    }
```

Table 3: Trace Checking tool requirements

IF	Title	Priority	Status	Comments
R4.28	Safety and privacy properties loading	MUST	✗	
R4.26	Report generation of analysis results	Should	✓	Trace checking results are stored into an output file.
R4.28	Safety and privacy properties loading	MUST	✓	User can define templates of the relevant properties and choose them when trace checking is invoked.
R4.28.1	Definition of time window of interest for safety/privacy properties	MUST	✓	Storm monitoring allows the user to select the the time window.
R4.29	Event occurrences detection for safety and privacy properties monitoring	MUST	✓	DICE-TraCT implements the logic to customize how to select events from Storm logs.
R4.30	Safety and privacy properties monitoring	MUST	✓	Storm monitoring is currently supported.
R4.30.1	Safety and privacy properties result reporting	MUST	✓	
R4.31	Feedback from safety and privacy properties monitoring to UML models	Could	✗	
R4.30	Safety and privacy properties monitoring	MUST	✓	Privacy properties are not supported yet. Safety properties are related to some parameters of the verification model
R4.32	Correlation between data stored in the DW and DICE UML models	MUST	✓	The case of Storm application has been studied to verify the need of instrumenting the source code.

2.3 Regression based Anomaly Detection

In the wake of growing complexity of data-intensive applications, market competition and pressure to deliver applications to the market as quickly as possible without decrease in their quality, application development lifecycle needs to be continuous, iterative, automated and cost-efficient at the same time.

SPEC group outlined the need in the performance-oriented DevOps for the enterprise applications [4]. The same stands true for the Data-Intensive Applications (DIAs). Anomaly Detection (AD) tool presented below is devised to automatically train statistical (linear regression) models for any type of software applications (web-based, cloud, enterprise, data-intensive) or Big Data technology at each new deployment and then look for the possible presence of anomalous behaviour by comparing the current model (deployment n) with the one trained at the deployment $n-1$. In other words, AD tool seeks to uncover any potential anomalies that may arise after every modification of the application.

Sequential model selection algorithm employed in the model training module was adopted based on the requirement for cost-efficiency. During application development (especially on the early stages) historical data is not available, while conducting model selection by conventional means – providing the algorithm with the set of observations of a certain size – might be costly (as some of the data points might turn out to be redundant), especially with the possible presence of interactions and/or non-linear terms which are not known in advance. Therefore, each new observation should be added only when absolutely necessary and also should add as much information about the system under test as possible.

The purpose of this tool/method: detect presence of performance anomalies with the elements of root cause analysis at the application design time iteratively and in a cost-effective manner by:

- Accept a set of inputs (from the developer and certain deployment information)
- Train statistical model describing application behaviour for the given performance or reliability metric(s) of interest (set by the developer)
- Compare this model with the model, trained at the previous deployment and identify the presence of anomaly(-ies), if any.
- Generate report for the developer indicating the presence/absence of performance anomalies and possible root causes (if anomalous behaviour is detected).
- Repeat the process for each deployment version.

On the high level the Regression based AD tool consists of two modules: model training and model analysis. Model training module accepts input from the user (via IDE or DMon) along with the required deployment data (from the IDE translated by the DICE deployment tool [6]) and builds the statistical model of the application behaviour for the performance or reliability metric defined by the user (developer) at the current deployment (***n***). This model is then submitted to the model analysis module where it is compared to the application performance model trained at the previous deployment (***n-1***). Afterwards, the report is generated to inform the developer whether the anomalous behaviour is present and suggest possible root cause(s). The high-level architecture of the Regression based AD tool is presented on the Figure 11:

Architecture and functionality for each of the modules are described in the following sections.

2.3.1 Model Training module

The purpose of this module is to train a statistical (linear regression) model capturing the behaviour of the application at the current deployment for the chosen performance or reliability metric of interest.

In order to do this the tool accepts a list of inputs, some of them are deployment-related information and some are tuning/configuration parameters for the tool. These inputs can be either provided interactively by the developer or a set of predefined ”optimal” settings can be used instead. The list of input parameters to the tool along with their description is given in the Table 4:

On the high-level, model training module consists of two principal parts: initial model and *sequential model selection*. Sequential model selection block is composed of the *logical processing* block, '*action centre*', script, running the deployed application and *sequential model selection* algorithm. Their functionality and operation are described in details below.

Low-level architecture of the model training module combined with the process flow diagram is presented on the Figure 14.

Initial Model

At this stage the tool selects two factors from the list **L**, creates the simplest two-level factorial design (see Table 3), runs application four times with the settings from the Table 5 to obtain observations $y_1 - y_4$ and fit the model (1).

$$y = \beta_0 + \beta_1 * x_1 + \beta_2 * x_2 \tag{1}$$

Table 4: List of inputs to the model training module.

Input	Description	
Performance or reliability metric to be modelled (response)	This metric is always defined by the developer. Examples: response time, data processing time, memory utilisation, CPU utilisation, I/O rate, throughput etc.,In the current implementation the tool allows to create only one model per entire model training cycle (i.e. there is currently no functionality to train models for several metrics from the same data simultaneously)	
Budget	Budget limitation can arise from the money or time constraints and is expressed as a maximum number of experiments the developer can afford to run to obtain observations used in training the model. Two operation modes are currently available: unconstrained and constrained. In the case of the budget-constrained option user should provide the value of the maximum number of experiments they can afford.	
Mode	**Automated**	**Interactive**
	When choosing this mode developer does not participate in setting any input parameters except for the response and budget.	Developer can influence model training process by tuning certain settings manually.
R^2	Minimal required goodness of model fit to data ([0;1]). In general, it is a trade-off with the budget. Smaller threshold R^2 allows to collect less observations (but results in less accurate predictive models). However, very large R^2(close to 1) in most cases would imply overfitting (good fit to the training data, but bad predictive capability). The optimal interval is considered to lie in the range of 0.7-0.85.	
	Hard-coded with 'optimal' value (0.85)	User can tweak it.
List,of factors (L)	List of deployment-specific parameters (hardware, configuration parameters etc.) and their settings (min, max, mid etc.) to be used for model selection (fitting). Examples: number ofexecutors, allocated RAM, number of VMs,...	
	Chosen automatically from the available deployment information.	Developer can compile their own list, based, for example, on the perceived importance of certain input parameters (factors).

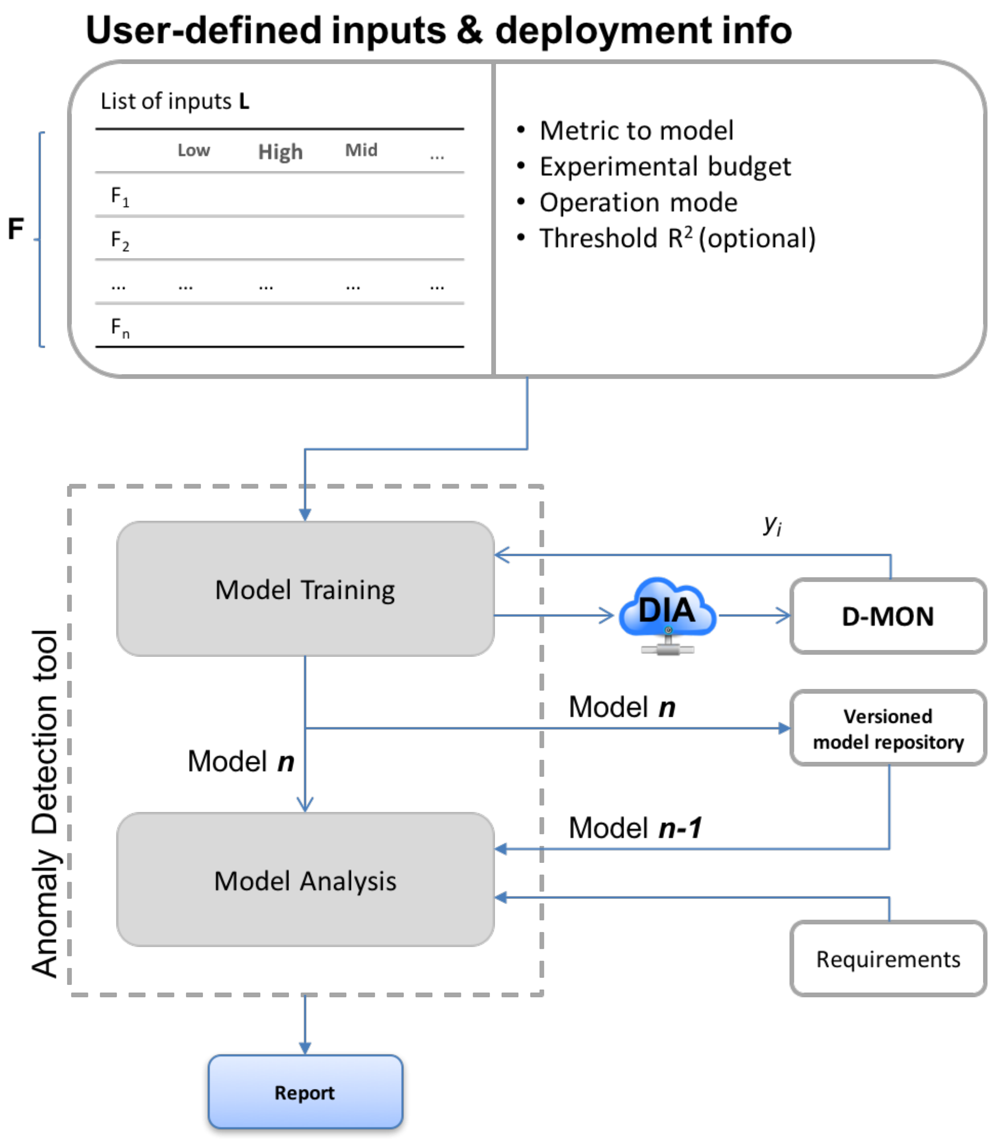


Figure 11: Regression based Anomaly Detection. Architecture, inputs and interaction with other DICE tools.

Where $\beta_0 - \beta_2$ are coefficients of the fitted model.

Additionally to the coefficients and model terms the tool saves the following 'metadata' for internal use: estimated R^2 for the selected model on each iteration, p-values for model coefficients and Boolean flags *interactions* and *squared_terms*.

This initial model along with the metadata is then passed to the sequential model selection block for further model fitting.

Sequential model selection

Sequential model selection stage involves four modules (shown on the Figure 14):

- Logical processing module, where the decision what to do next based on the available data and metadata is made by the tool. Action centre', where the candidate model is formed.
- Script running application under test and retrieving experimental data (observations) from the DICE Monitoring Platform to be used in model selection (fitting).
- Model selection algorithm, which uses observations to select relevant model terms from the candidate model formed in the 'action centre'.

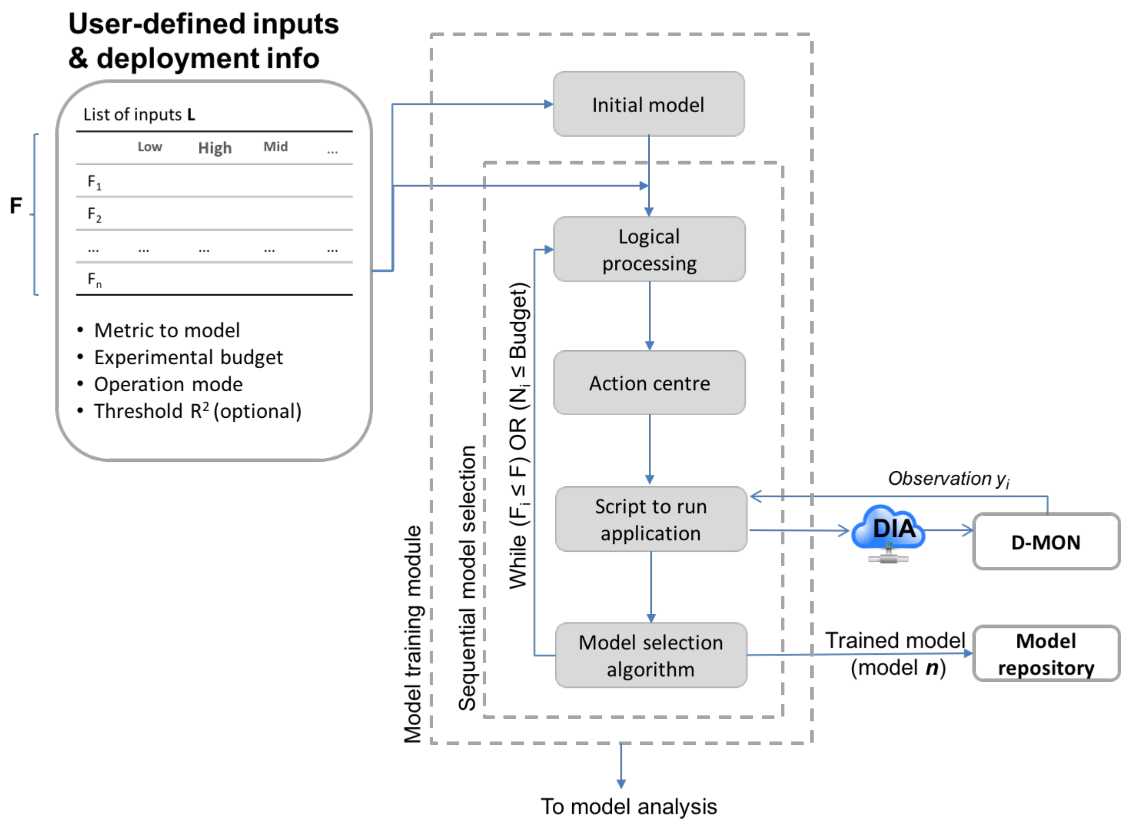


Figure 12: Model training module: architecture and process flow..

Table 5: Two-level full factorial design for two factors.

X_1^*	X_2	Observation
-1^{**}	-1	y_1
-1	1	y_2
1	-1	y_3
1	1	y_4

Logical processing module accepts the following inputs: R_2^i (of the latest fitted model) and R^2 threshold (from the Table 4), number of factors from the list **L** already used (F_i) and total (F), number of experiments conducted (N_i) and total (Budget), p-values of estimated model coefficients and Boolean flags **interactions** and **squared_terms** for the given iteration. Depending on the combination of these inputs it will trigger execution of one of the four blocks in the 'action centre' or terminate the operation:

a) Option 1 - if the model contains only main effect for the newly added factor (it can contain interactions for the previously added factors) and its measure of fit is less than the threshold, it means that there are interdependencies (interactions) between this newly added factor and previously chosen factors that influence response. In this case the tool will pass the information to the 'action centre' to generate linear interactions (up to the highest order) between the newly added factor and the factors already in the model.

b) Option 2 - if the model contains linear interactions but the fit is still not good enough, this signifies the presence of non-linear dependency between at least one of the factors and response. The tool will send the signal to the action centre to generate all possible non-linear terms (if it is possible for the given factor) up to the predefined polynomial order (e.g. 2 or 3) and update design matrix (matrix of factors and interactions and their settings) with one more row of settings obtained via D-optimal design.

c) Option 3 - the situation when current model contains both interactions and/or non-linear terms, but the fit still hasn't reached the threshold means that the number of available data points (observations) is less than the number of significant model parameters. In this case the tool will update design matrix with one more row of settings obtained via D-optimal design.

d) Option 4 – when the measure of fit is finally satisfactory for the selected model, the tool adds one more factor (only the main effect) and updates design matrix with one more row of settings obtained via D-optimal design.

e) When either the tool has sifted through all factors from the list L or ran out of the experimental budget, the model training process terminates and the model is exported and saved as a data structure (cell array).

Boolean flags *interactions* and *squared_terms* mentioned above are used by the logical processing module to identify the composition of the model fitted on each iteration (i.e. the terms it contains) and use this information in the decision-making process.

In the '**action centre**' candidate model for the terms selection is formed based on the information sent from the logical processing block using D-optimal design [14]. D-optimal design aims to find the factors' settings which would allow to obtain an observation (after running the application with these settings) that would be the most useful for model selection.

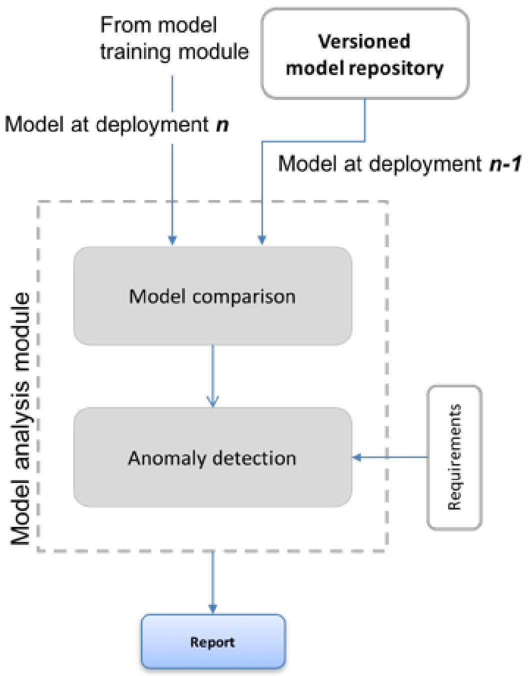
Model selection algorithm builds on the idea of creating the model in the situation where the number of available observations is less than the size of the candidate model (from which algorithm aims to select only relevant terms). There are a number of methods available, but Dantzig Selector (DS) [5] was chosen due to it employing the so-called non-asymptotic bounds. Other model selection methods operating on small samples (e.g. Lasso [21], ridge regression [12]) assume that sample statistics (mean, variance) can be accepted for population statistics in obtaining estimators (model coefficients). This assumption negatively affects accuracy of the asymptotic bounds used by the optimization algorithms in [21], [12] to estimate model coefficients and, as a consequence, prediction accuracy of the resulting model. Dantzig Selector, on the other hand, employs approximation theory to quantify approximation error arising from using the sample mean and variance instead of the population mean and variance for the estimator. This approximation error, in essence, describes the discrepancy between the approximated model (the model that will be fit) and 'true' model (the unknown model that accurately describes the process in the system), which is then added to the bounds in the optimisation algorithm for finding estimators.

However, all mentioned model selection methods rely on the tuning parameter λ (lambda), which controls the magnitude of estimated coefficients (whether these coefficients are above or below the noise level and, hence, should be selected into the model or discarded). Therefore, DS could not be used on its own (as optimal is unknown). This issue was addressed by fitting the model with Lasso before DS, as its Matlab implementation provides an entire coefficient path with the range of lambda values (i.e. not one model, but a range of models, each with its own estimated λ). Then DS can be implemented for each value in this narrow range of lambdas. Like other model selection methods, DS does not provide accurate estimates of model coefficients, they are 'shrunk' (reduced) by the value of λ (soft-thresholding). Therefore, an approach widely applied in Machine Learning is used: after the model is selected (for each λ provided by Lasso), chosen model terms are fit with OLS (Ordinary Least Squares) and the model with the highest R^2 (with account for overfitting and possible multicollinearity) is then chosen.

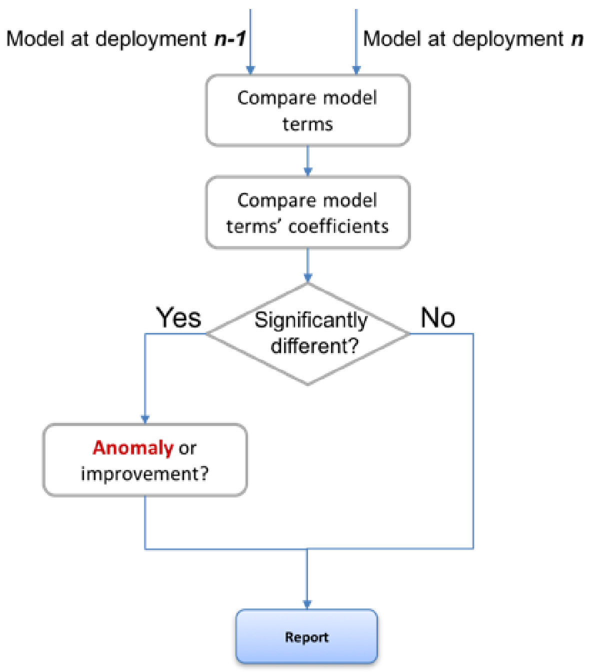
The tool continues to sequentially select the 'best' model until one of the two following conditions is met:

- All factors from the list **L** are investigated and the satisfactory fit is achieved;
- The tool still hasn't got through all factors, but exceeded the experimental budget (e.g. because the number of factors on the list **L** was larger than available experimental budget).

After the model selection process is finished, the tool checks coefficients' p-values to remove statistically insignificant terms (some of the terms in the resulting model still can be 'noise') and then saves



(a) Model Analysis module. Architecture



(b) Model Analysis module. Process flow diagram

Figure 13: A figure with Architecture and Flow Diagram

the model as a data structure (cell array). The data structure contains application deployment version number and the list of model terms with an array of respective coefficients.

This model is then used as an input (the model of the application at the deployment **n**) for the **model analysis** module.

2.3.2 Model Analysis module

After the linear regression model of the application at the deployment n for the performance or reliability metric of interest is created and saved as a data structure (cell array), the next step is to identify whether there is an anomaly present in the modified application.

Model Analysis (MA) module accepts as inputs two linear regression models and compares them. The first model is of the application at the previous deployment $(n-1)$ stored externally as a data structure (cell array), and the second is of the current deployment (n) created and saved on the model training step. The principle of the anomaly detection procedure implemented in the DICE AD tool utilises the well-known property of the linear regression models. Namely, that each model term (and its corresponding coefficient) can be interpreted as a predicted change in response caused by the corresponding input when all other inputs are fixed.

This means that comparison of the models can be carried out term by term. After the comparison is finished, MA module generates report for the developer.

The high-level diagram of the MA module and the process flow are shown on the Figures 13a and 13b respectively.

Process flow diagram shown on the Figure 13b demonstrates that model analysis is undertaken in three principal stages:

- Detection of model terms missing from the new model (n) and/or new ones which were not present in the model of the previous deployment $(n-1)$;
- Detection if there is a change in the response (performance or reliability metric) caused by any of the model terms by comparing coefficients for the terms between two models;
- In the case change is detected, analyse if this change is performance degradation (manifestation of

Table 6: Outcomes and interpretation of the two application models' term-by-term comparison.

	1	2
Model n-1	$\gamma = \beta_{0n-1} + \beta_{1n-1} * x_1 + \beta_{2n-1} * x_4$	$\gamma = \beta_{0n-1} + \beta_{1n-1} * x_1 + \beta_{2n-1} * x_4$
Model n	$\gamma = \beta_{0n} + \beta_{1n} * x_1 + \beta_{3n} * x_6^2$	$\gamma = \beta_{0n} + \beta_{1n} * x_1 + \beta_2 * x_4$
What tool reports	Term(s) [insert name(s) of the input(s)] disappeared from the application model after modifications (at deployment n)	Term(s) [insert name(s) of the input(s)] appeared in the application model after modifications (at deployment n)
What it means	One or more terms stopped influencing response (performance or reliability metric of interest) after modifications were made to the application.	One or more terms started influencing response (performance or reliability metric of interest) after modifications were made to the application.

	3	4
Model n-1	$\gamma = \beta_{0n-1} + \beta_{1n-1} * x_1 + \beta_{2n-1} * x_4$	$\gamma = \beta_{on-1} + \beta_{1n-1} * x_1 + \beta_{2n-1} * x_4$
Model n	$\gamma = \beta_{0n} + \beta_{1n} * x_1 + \beta_{3n} * x_6^2$	$\gamma = \beta_{on} + \beta_{1n} * x_1 + \beta_2 * x_4$
What tool reports	Term(s) [insert name(s) of the input(s)] disappeared from the application model after modifications (at deployment n) and term(s) [insert name(s) of the input(s)] appeared in it.	Models,are identical in terms
What itmeans	One or more terms stopped influencing response (performance or reliability metric of interest) after modifications were made to the application, while one or more terms started to influence it.	If there is an anomaly, then it's manifested in some other way

anomalous behaviour) or not (improvement in performance caused by the modifications made to the application on the n-th stage of development)

More detailed anomaly detection procedure and interpretation of the results are presented below using two generic linear regression models as an example.

Step 1

Investigating models' composition. Various outcomes of the models' composition comparison are presented in the Table 6:

Outcomes 1-3 from the Table 6 do not automatically imply that missing/new terms are manifestations of anomalous behaviour. In this case the tool checks whether the addition/disappearance of the term(s) from the model leads to the improvement/degradation of the response or does not change it significantly.

At this stage the algorithm also identifies which of the terms coincide for $n-1$ and n version models and passes this information to the next step of the procedure.

Step 2

Analyse the behaviour of the application at the latest deployment for potential changes. In all of the outcomes outlined in the Table 6 the next step is to compare the coefficients of the coinciding model terms on the term-by-term basis. This is done by generating a sample (one for each model) of response values for one model term (thinking of other terms as fixed), while varying input on the normalised interval [-1;1]. Then a t-test is conducted on these two samples in order to establish if there is a significant difference between their means. T-test tests the null hypothesis that two data samples come from the normal distributions with equal means and equal (but unknown) variances (essentially meaning that these

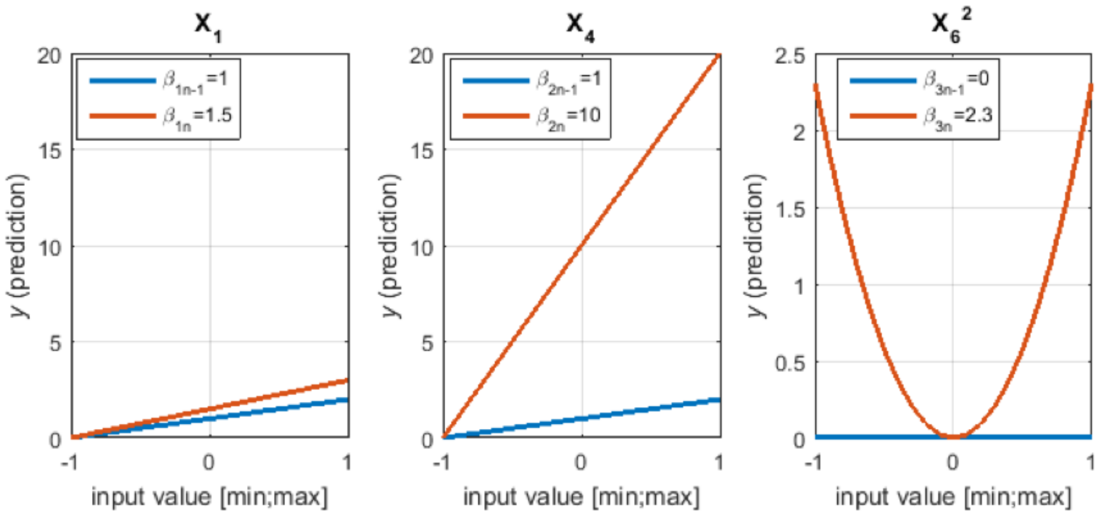


Figure 14: Visualization of the term by term comparison of model coefficients

two data samples were drawn from the same population and the difference in their means lies within the sample variance). If this null hypothesis is rejected, it means that these two independent samples come from different distributions. In relation to the software applications rejection of the null hypothesis can be interpreted in a way that these two data samples were generated by different processes.

T-test is not directional, i.e. it is impossible to tell if significant difference detected means improvement or degradation of the performance. Therefore, if it's flagged, the tool next compares two sample means to identify whether there is a performance improvement or degradation.

The procedure for detecting if non-coinciding significantly influence response is almost identical to the comparison of coefficients of coinciding terms. The samples are generated for each not coinciding term and their means are calculated. Where these terms are absent in the compared model, their coefficients and thus sample means are assumed to be zero. Then t-tests are conducted to identify if there is a significant difference between each of the means and zero. If the new/absent model terms cause significant difference in the predicted response, then the tool goes to the step 3 to establish if this difference means improvement or degradation.

Below is the example for the term-by-term comparison of the model coefficients for the generic linear regression model used in the Table 6 (the values for coefficients are chosen arbitrarily for illustrative purpose).

$$\begin{aligned} y_{n-1} &= 10 + 1 * x_1 + 1 * x_4 + 0 * x_6^2 \\ y_b &= 11 + 1.5 * x_1 + 10 * x4 + 2.3 * x_6^2 \end{aligned} \tag{2}$$

By assuming that 'other model terms are fixed', we can set them to zero and thus look at the projections of the response surface onto the axes corresponding to the specific terms. The visualisation of these projections for the terms from the equations (2) is presented on the Figure 14.

The leftmost plot on the Figure 14 illustrates the term X_1 for both models, with significant difference not detected. The curves lie close to each other and the model coefficients, which in this setup (only one variable and with 0 as the centre of the interval) are, essentially, sample means are also close. The plot in the middle illustrates the case where significant difference between deployments $n-1$ and n was detected for the term X_4. The rightmost plot shows the comparison between the term X_6^2 which is absent from the model $n-1$ and thus modelled as having coefficient = 0.

Step 3

Establish if detected change in response is performance degradation (anomaly) or not (improvement). Because t-test is not directional and some performance or reliability metrics improve by decreasing (e.g. response time, data processing time), while others improve by increasing (e.g. throughput), it is impossible for the MA module to identify the influence of change (positive/negative) only by comparing two sample means. Therefore, it needs additional information to make a decision.

This information can be obtained from the relevant requirement (e.g. supplied directly by the developer on the tool's prompt or extracted by the tool from the requirements repository). For example, if there is a requirement exists: '*Performance (or reliability) metric A must not exceed value X*', the MA module can parse it looking for the keywords such as 'not exceed', 'not larger than/not greater than', /larger than/greater than' etc. and use this information to identify if the direction of change in performance (or reliability) metric is positive or negative (improvement or degradation).

2.3.3 Report generation

After the model analysis is finished, the AD tool generates report with the analysis results and suggestions for possible root causes in the case performance degradation is detected. This functionality is currently not implemented, because it is proposed that the tool would communicate its output to the developer via DICE Monitoring Platform and the work on integration hasn't started yet.

For the visualisation of the entire model or specific terms (similar to the plots from the Figure 14) in the DICE Monitoring Platform the range of input parameters can easily be re-scaled from the normalised interval [-1;1] back to their original values.

2.3.4 Discussion

From the description of the model analysis process it can be seen that in addition to flagging the presence/absence of an anomaly the AD tool allows to identify if there is a significant improvement in performance as a result of modifications undertaken and also provides basic (preliminary) root cause analysis.

The information obtained in the Model Analysis procedure can be potentially exploited in the root cause analysis due to the assumption based on the nature of the linear regression models. It is well-known that linear regression models are the so-called 'black box' models. Meaning that any change in the system they describe can only be reflected in the model terms composition and their coefficients. However, due to the fact that each coefficient and model term (i.e. input parameters such as, e.g. configuration, hardware and so on) in isolation reflects its impact on the response, it can be suggested that if its negative influence is detected, then it might be the source/manifestation of the problem (or architecture/code directly connected with this 'problematic' input parameter). This additional functionality comes as a 'by-product' of the developed approach to the anomaly detection.

Next steps for the tool implementation and development would be to integrate with the other tools in the DICE framework (DMon, Deployment tool, IDE), provide support for PMML to store the model in the universal format and validate the tool with experiments.

As a conclusion, Table 7 indicates how the work reported in this deliverable addresses the requirements identified by the requirement analysis.

Table 7: Requirements for Regression based AD

ID	**Title**	Priority	Status	Comments
R4.24.5	Predictive Model saving for Anomaly Detection	MUST	✓	Currently trained model is exported and stored as a data structure. Next step would be to support PMML format
R4.26.2	Report generation of analysis results	Should	✗	Because the tool is not integrated with DMon at this stage
R4.36	Detect anomalies between two versions of DIA	MUST	✓	
R4.37	ADT should get input parameters from IDE	MUST	✓	Because the tool is not integrated with DICE IDE or deployment tool at this stage, all input is provided via command line and configuration files

3 Use cases

This section details what use cases are handled by each tool. It shows the main workflow for ADT as well as that of TraCT. For the Regression based AD the input parameters for the method are detailed as well as example configuration files.

3.1 Anomaly Detection

Anomaly detection tool will check for anomalies during the runtime of a deployed application on a wide range of Big Data frameworks. These frameworks are unchanged from those supported by the DICE Monitoring platform (DMon) [13]. In the case of unsupervised anomaly detection methods the querying of DMon will result in the generation of the data sets on which these methods will operate. In essence the only thing that the end user needs to do is give ADT the query string and the desired time frame.

For supervised anomaly detection methods this is a bit more complicated as it is not enough to give the query string and time frame. The data sets must be labelled in order to create a viable training and validating data set. Once this is done the resulting predictive models can be easily applied during runtime.

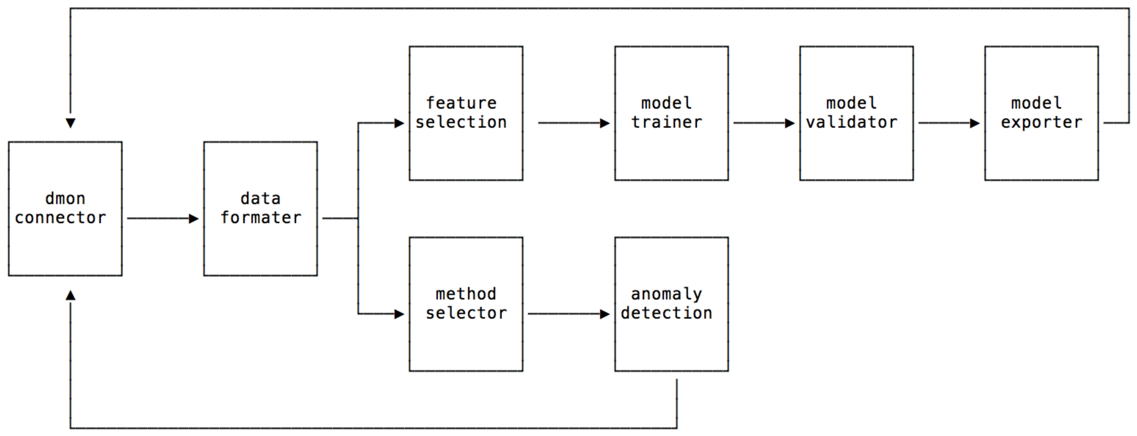


Figure 15: Anomaly Detection flow.

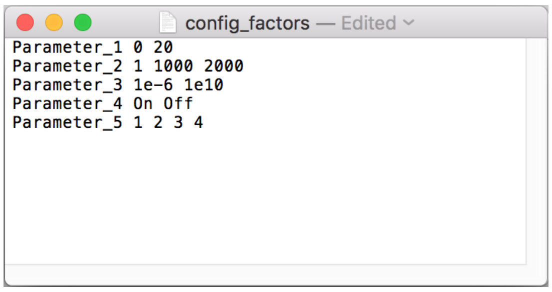
Figure 15 show the basic flow of data between all components of ADT. It is easy to see that there are two branching workflows. The first one is meant for training and validating the aforementioned predictive models while the second one is meant for unsupervised methods and the loading of the validated models.

It is important to note that the *method selector* and *anomaly detection engine* are the two tools required for detecting and signalling anomalies. The *method selector* is used to select between unsupervised and supervised methods (including their runtime parameters). This component is also responsible for loading pre-trained predictive models. The *anomaly detection engine* is in charge of instantiating the selected methods and signalling to the *dmonconnector* any and all detected anomalies.

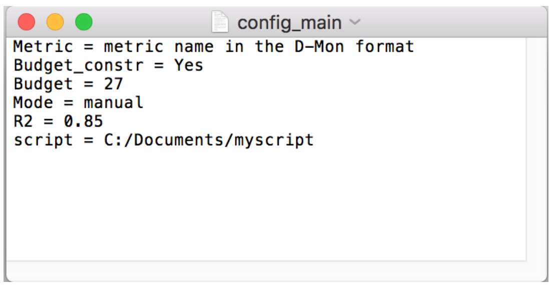
We can think of the first branch as the batch layer of a lambda architecture. Once it trains and validates a model it sends it to be stored and indexed into DMon. From there the second branch can download it and instantiate it. This in essence represents the speed layer. There can be more than one instance of ADT at the same time so scaling should not pose a significant problem. However, this has not been tested for M18. Future work will tackle this issue.

3.2 Trace Checking tool

Trace checking is employed to verify the runtime behavior of a deployed application. This approach requires that log traces meet specific properties (over time in our case) to certify the adherence of (portions of) the runtime executions to the behavioral model that is assumed at design time. If the runtime behavior does not conform to the design, then the design must be refined and later verified to obtain a


config_factors — Edited

```
Parameter_1 0 20
Parameter_2 1 1000 2000
Parameter_3 1e-6 1e10
Parameter_4 On Off
Parameter_5 1 2 3 4
```


config_main

```
Metric = metric name in the D-Mon format
Budget_constr = Yes
Budget = 27
Mode = manual
R2 = 0.85
script = C:/Documents/myscript
```

Figure 16: Examples of configuration files

new certification of correctness. Specifically, trace checking in DICE allows the DICE users to verify two classes of properties that are derived from the DPIM, DTSM and DDSM models.

The first class of property of interest for DICE consists of temporal properties that allow for checking the validity of the results obtained with the verification analysis defined in WP3. Currently, verification is carried out at DTSM level on UML models enriched with information that are useful to perform the analysis of Storm applications (more details in DICE Verification Tool - Initial version [15]). Storm topologies, that are captured by DTSM models, are analyzed through a logical model which captures their behavior over time. Storm topologies consist of nodes that represent computational resources implementing the application. They can be either data sources or data processors (bolt) that manipulate input messages.

To verify DTSM models the designer must provide some parameter values that abstract the (temporal) behavior of spouts and bolts (the complete description can be found in the document DICE Verification Tool - Initial version [15]).Trace checking is employed to extract from real executions those parameter values that are not available from the monitoring component of the framework as they might be inherently specific of the modeling adopted for verification. An example of such a parameter is the ratio between the number of messages that are received by a bolt and the number of messages that it emits in output.

The second class of property concerns privacy aspects of the applications. Privacy constraints are designed through the the DPIM, DTSM and DDSM models by means of suitable annotations and possibly new ad-hoc constraints specifying, for instance, authentication and authorization restrictions, resources policies, encryption on communication etc. Checking the integrity of the deployed and running application can be achieved through trace checking with the analysis of application logs and suitable properties derived from the models.

3.3 Regression based Anomaly Detection

Regression-based Anomaly Detection tool is implemented in Matlab and compiled as an executable file that can be run as a standalone application from the command line. *MyAppInstaller_web.exe* is an executable file of the MATLAB Generated Standalone Application and detailed installation instruction can be found in [1]. After the application is installed it can be executed by running *regressionad_main* file.

All user-defined input parameters are supplied via the *config_main.txt* and *config_factors.txt* files that can be found in the folder with installed application. These parameters are listed in the Table 6 and an example for each of the configuration files is shown on the Figure 8.

It is important to note that automated mode is currently not supported, because in this mode factors should be imported from the DICE deployment tool [6], which is not integrated yet.

The code and documentation for the tool are released on the Github repository and can be found on the official DICE Github [7].

[7]https://github.com/dice-project/Anomaly-Detection-Regression-Based-Tool

Table 8: Input parameters for the packaged tool

Parameter	Description	Input format
Metric	Performance or reliability metric to model and investigate for anomaly	The name of the metric should be provided exactly how it's supplied by the DMon, because the tool queries DMon to obtain the measurements for this metric.
Budget_constr	Is there a limit on the number of experiments to run	Yes/ No
Budget	If Budget_constr is 'Yes', provide the maximum number of experiments possible to run. If 'No', do not enter any value	Number (e.g. 27) or blank space
Mode		auto – automated mode, user does not control R^2 (see below) and leaves the config_factors.txt blank manual – user can modify R^2 and, config_factors.txt
R^2	Script running the Data-Intensive Application to obtain observations for the performance or reliability metric,(external executable file provided by the user)	Number from 0 to 1 (optimal interval 0.7¸0.85). For the auto mode it is set at the default value (0.85)
script	Script running the Data-Intensive Application to obtain observations for the performance or reliability metric,(external executable file provided by the user)	Full path to the file location, including file name.
config_factors.txt	Separate configuration file containing the list of the parameters of the DIA and the range in which they can vary (min/max/mid etc.)	Each line in the file contains the information for one input parameter in the following form (see detailed example in the Figure 6): Parameter_name level_1 level_2 . . .

4 Integration and Validation

This section covers integration as well as validation issues for each tool. The first subsection will deal with both ADT as well as Regression based AD and how it interacts with the overall DMon Architecture. The second section details the Trace Checking tool and how it combines logs and checks for sequential anomalies.

4.1 Anomaly Detection

ADT will have a closer integration with DMon than with other tools from the DICE solution. This is mainly due to two facts. Firstly, ADT needs data on which to run anomaly detection methods. Thus it is extremely important to have data available in a format which is usable. Second, ADT together with the monitoring forms a lambda architecture. Each instance of ADT can have the role of batch or speed layer while DMon has the role of a serving layer. For more details see Figure 17.

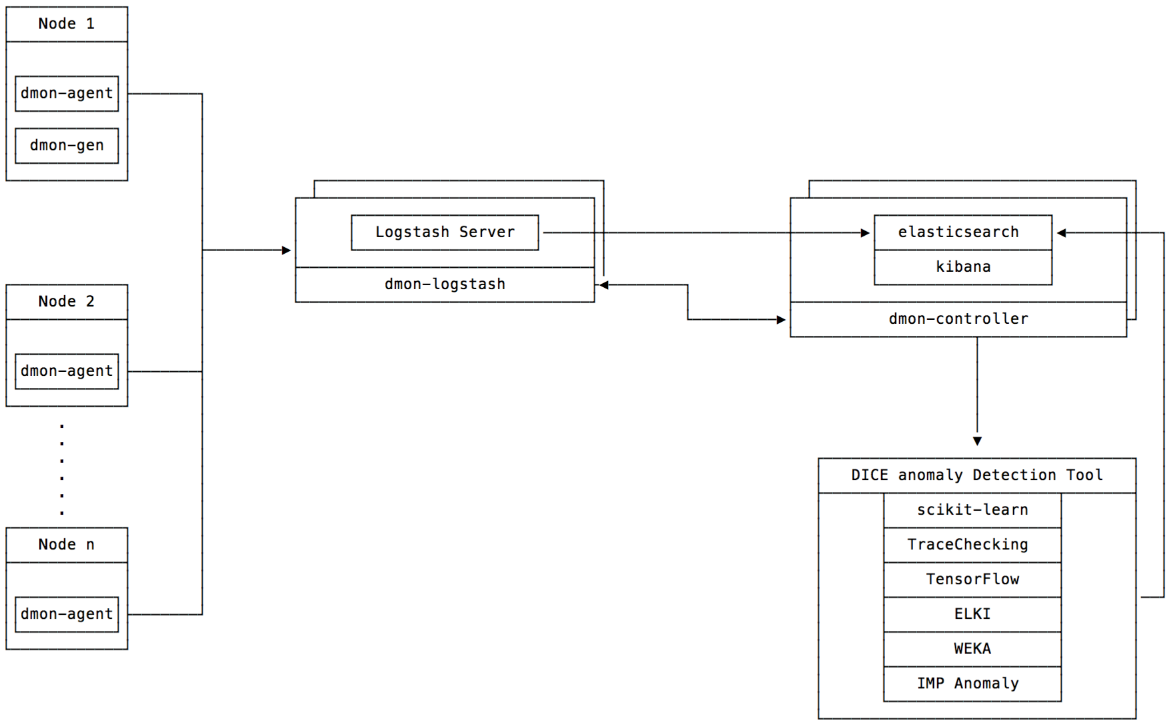


Figure 17: Anomaly detection integration with DMON.

As mentioned before the detected anomalies will be sent and indexed into DMon. All DICE actors and tools will be able to query this special index to see/listen for detected anomalies. In this way it is possible to create specialized ADT instances for each anomaly detection method in part. The result will be reflected in the same index from DMon. This architecture also allows us serve the results of both the monitoring and anomaly detection on the same endpoint (DMon).

As mentioned in section some anomaly detection methods, more precisely the ones using supervised learning techniques, need labelled data in order to function properly. This is a fairly complicated thing to accomplish. One solution is to label all normal data instances and all unlabelled instances are considered anomalies. In most systems the normal data instances far outnumber the anomalous ones so labelling them is extremely impractical.

We have decided to create a semi automated way of creating labelled data. This is accomplished by inducing different types of anomalies during job definition and execution. A specialized tool called *dmon-gen* has been implemented which allows the changing of both runtime parameters as well as platform specific parameters of a Yarn, Spark job.

The tool *dmon-gen* has to be located on one of the VMs from the Yarn/Spark cluster. As input the user has the ability to define a set of experiments using a JSON descriptor. An example descriptor can

Listing 2: JSON job descriptor

```
{
        "exp1":
        [
                {
                        "yarn":["pi","10","100"],
                        "cardinality": 1,
                        "conf":{"hdfs":{"DATANODE":
                        {"dfs_datanode_du_reserved": "8455053312"}},
                                        "yarn":{"NODEMANAGER":
                                        {"mapreduce_am_max-attempts": "2"}}}
                }
        ]
}
```

be seen in Listing 2.

First the user can define the name of the experimental run. Next, a user can define a list of both application specific as well as platform specific settings. For example in Listing 2 we see that the there is one yarn experiment called *pi* which has two parameters (10 signifies the number of maps while 100 the sample size). The *cardinality* setting is used to define how often the experiment is to be run.

Lastly, we have the *conf* settings which denote the settings based on roles for each big data service. It is important to note that we use Cloudera CDH 5.7.x for *dmon-gen* so the naming conventions are the same for our tool as with the current version of CDH [8].

It is easy to see that by using *dmon-gen* we are able to induce some types of anomalies in an automatic manner and are able to correlate these with the metrics collected during a specified time-frame. In essence labelling the data.

ADT is dependent on DMon for what type of Big Data services it can be used upon. As long as DMon is able to collect metrics we can create training/validation datasets in order to define the types of anomalies (point or contextual anomalies) we wish to detect.

ADT[9] as well as the *dmon-gen*[10] tool can be found at the official DICE Github Repository. These repositories also contain the up to date documentation of the tools.

4.2 Trace Checking tool

The validation of the first version of trace checking tool that is described in this document is run locally on a single machine and without the integration of trace checking tool in the monitoring platform as the integration will be developed for the next release at month M24.

The goal of the validation is to show the workflow realizing the trace checking procedure in DICE. The main steps are the following:

1. DICE-TraCT receives a user request containing a trace checking specification problem;

2. Upon the request, DICE-TraCT query the D-mon platform which replies with the log files necessary to perform trace checking;

3. DICE-TraCT merges the log files as described in the previous Sec. 2.2.1 and build the formula to be used for trace checking the logs;

4. Finally, DICE-TraCT runs a Spark job and collects the result of the trace checking analysis.

The most relevant peculiarity of the trace checking engine (TCE) employed in DICE is the distributed procedure that TCE implements to evaluate temporal formulae on log files. TCE is implemented in

[8]https://www.cloudera.com/products/cloudera-manager.html

[9]https://github.com/dice-project/DICE-Anomaly-Detection-Tool

[10]https://github.com/igabriel85/dmon-experiments

Spark, a general engine for large-scale data processing which is also supported in DICE as technology to build big-data applications. Stemming from a distributed implementation, TCE takes advantage from a distributed file system to run the trace checking procedure on a (pseudo-)distributed architecure (local executions might be multi-threaded).

To validate DICE-TraCT, a local installation of Apache Hadoop [22] and Apache Spark [23] is first set up. The former allows the trace checking tool to work on top of HDFS, the distributed file system that Spark can access to store and retrieve data.

Next sections describe in detail the four steps mentioned before. The reference topology for the validation consists of 5 nodes: two spouts, called *spoutA* and *spoutB*, and three bolts, called *boltA*, *boltB* and *boltC*. The structure of the topology is not relevant to show the trace checking validation as the trace checking procedure is run to analyze the behavior of single nodes only and the considered property is not related to the topology graph.

Step 1

The trace checking procedure is activated by the user through the DICE-IDE. DICE-TraCT receives in input a descriptor which defines the parameters to run trace checking for the selected topology shown in the IDE. The descriptor is a JSON file that is built automatically through the IDE by the user who activates the analysis of the topology. The descriptor used to validate the tool is shown in Listing 3.

Listing 3: Field "node" of the JSON script used for validation experiment.

```
"nodes": [
                {
                   "name": "spoutA",
                   "type": "spout",
                   "parameter": "idleTime",
                   "timewindow": 3600,
                   "method": "",
                   "relation": "",
                   "min": 10,
                   "max": 1000
                },
                {
                   "name": "spoutB",
                   "type": "spout",
                   "parameter": "idleTime",
                   "timewindow": 3600,
                   "method": "",
                    "relation": "",
                   "min": 0,
                   "max": 2000
                }
```

The information stored in the descriptor are the following:

- The field called "topologyname" specifies the topology name which is used by DICE-TraCT to query the monitoring platform. This field allows DICE-TraCT to obtain all the necessary log files to perform trace checking.
- A list of descriptors that specify, for each node, a non-functional property that the user wants to check for a node on the logs collected from the running topology. The properties can be those specified in the verification model like, for instance, the ratio *sigma* between the number of messages in input and the number of messages in output of a Storm node; or any user-defined property which can be translated by DICE-TraCT into a trace checking instance. Each item defined by curly brackets defines a list of data that are needed to collect the suitable set of logs from D-mon and

to define the temporal logical formula specifying the property to check. DICE-TraCT is equipped with standard handlers which are able to manipulate predefined properties (like *idleTime* used later) but allows for defining new custom handlers based on user needs.

- A list of formulae descriptors that specify user-defined logical formulae to be used for trace-checking.

The query defined by the script in Fig. 3 contains two node descriptors (between curly brackets) associated with *spoutA* and *spoutB* nodes. Both specifies to verify that the value of the *idleTime* of the node is between the values defined with min and max values. Each descriptor also defines the duration (in seconds) of the log to use for running trace checking. The length will be specified in the query to the DMon platform to collect, from the Storm deployment, the most recent events occurred in last the period of that length (currently, we assume that the length is measured from the last timestamp recorded in the logs of the Storm application).

Step 2

Being not integrated with the monitoring platform, DICE-TraCT simulates the query to DMon through an I/O operation on the local file system. We assume therefore that DMon provides:

- A JSON descriptor which specifies, for each node of the topology, the log file name where Storm logs its events. As already explained before, a node can reside on different machines within many workers; therefore, the events associated with a node might be stored on different logs. Listing 4 shows the descriptor.

- The logs of the deployed topology. In the current example, they are called w1.log, w2.log and w3.log.

Listing 4: Json script sent by D-Mon to DICE-TraCT. Each node name is endow with the list of logs where it appears.

```
{
    "topologyname": "ATopology",
    "logs": [
                    {
                        "nodename": "spoutA",
                        "logs": "w2.log"
                    },
                    {
                        "nodename": "spoutB",
                        "logs": "w1.log"
                    },
                    {
                        "nodename": "boltA",
                        "logs": "w1.log,w3.log"
                    },
                    {
                        "nodename": "boltB",
                        "logs": "w3.log"
                    },
                    {
                        "nodename": "boltC",
                        "logs": "w2.log,w3.log"
                    }
                ]
}
```

Step 3

LM merges the log files to create a unique log file for each node of the topology. Each log contains all the events generated by the thread implementing the node running on different workers. As an example, we show a portion of w3.log which contains events of *boltA*, *boltB* and *boltC* and later a portion of the merged log of *boltA*. Some information in the log are removed for convenience.

Listing 5: JSON script sent by D-Mon to DICE-TraCT. Each node name is endow with the list of logs where it is appears.

```
2016-10-16T12:10:00.000+0000 [Thread-11-boltB] ... [INFO] emit {tuple}
2016-10-16T12:10:00.197+0000 [Thread-11-boltB] ... [INFO] receive {tuple}
2016-10-16T12:10:00.308+0000 [Thread-11-boltB] ... [INFO] emit {tuple}
2016-10-16T12:10:00.462+0000 [Thread-11-boltB] ... [INFO] receive {tuple}
2016-10-16T12:10:00.844+0000 [Thread-11-boltB] ... [INFO] emit {tuple}
2016-10-16T12:10:00.858+0000 [Thread-9-boltA] ... [INFO] emit {tuple}
2016-10-16T12:10:01.029+0000 [Thread-11-boltB] ... [INFO] receive {tuple}
2016-10-16T12:10:01.202+0000 [Thread-11-boltB] ... [INFO] emit {tuple}
2016-10-16T12:10:01.740+0000 [Thread-9-boltA] ... [INFO] receive {tuple}
2016-10-16T12:10:01.830+0000 [Thread-9-boltA] ... [INFO] emit {tuple}
2016-10-16T12:10:01.890+0000 [Thread-12-boltC] ... [INFO] emit {tuple}
2016-10-16T12:10:01.899+0000 [Thread-9-boltA] ... [INFO] receive {tuple}
2016-10-16T12:10:02.266+0000 [Thread-9-boltA] ... [INFO] emit {tuple}
2016-10-16T12:10:02.339+0000 [Thread-9-boltA] ... [INFO] receive {tuple}
2016-10-16T12:10:02.387+0000 [Thread-12-boltC] ... [INFO] receive {tuple}
2016-10-16T12:10:02.679+0000 [Thread-12-boltC] ... [INFO] emit {tuple}
2016-10-16T12:10:02.710+0000 [Thread-12-boltC] ... [INFO] receive {tuple}
2016-10-16T12:10:02.881+0000 [Thread-11-boltB] ... [INFO] receive {tuple}
```

Based on the topology descriptor shown in Listing 4, node *boltA* is run on two different workers that logged events into the files w1.log and w3.log on different machines. The executors running the instances of *boltA* are two and they are called *Thread-9*, in w1.log, and *Thread-1*, in w3.log. The following Listing 6 shows the result of LM and, precisely, an extract of file boltA.log which contains all the events related to node *boltA*, that are labeled with the executor names *Thread-1* and *Thread-9* and that are ordered with respect to their timestamps.

To implement a flexible parser of log event files, DICE-TraCT gets information on the syntactical structure of the log lines from a configuration file that is set before running trace checking. The syntax must conform the output format of the logger component that is used in the Storm application. The configuration file used for the current validation is the following:

The field "regexp" defines the regular expression that must match all the input lines in the log files that LM has to manipulate. The regular expression is defined according to the regular expression library of Python 2.7. All the other fields have the following meaning:

- "numberOfgroups" defines the number of the parenthesized elements between round braces appearing in the regular expression;
- "valuePosition" is a list of positions defining the part of the log line that has to be stored as an event in the merged file (in this example, number 9 refers to the executor name defined with the group "(Thread-)" and number 11 refers to the node event defined with the group "(emit—receive)").
- "keyPositions" is a list of positions that are used to define the timestamp of events in the merged file. The final timestamp is defined as the concatenation of all the elements matching the positions in the specified list.
- Finally, "nodePosition" is the position of the node name.

Listing 6: Event log ready for trace checking built by LM for node *boltA*.

```
    20161016121000858 ; thread_9_emit
    20161016121001423 ; thread_1_emit
    20161016121001740 ; thread_9_receive
    20161016121001830 ; thread_9_emit
    20161016121001899 ; thread_9_receive
    20161016121002266 ; thread_9_emit
    20161016121002339 ; thread_9_receive
    20161016121002513 ; thread_3_emit
    20161016121002776 ; thread_1_receive
    20161016121003097 ; thread_9_emit
    20161016121003295 ; thread_1_emit
    20161016121004054 ; thread_3_receive
    20161016121005333 ; thread_1_receive
    20161016121005436 ; thread_9_receive
    20161016121005617 ; thread_9_emit
    20161016121005800 ; thread_1_emit
    20161016121005888 ; thread_9_receive
    20161016121006086 ; thread_9_emit
    20161016121006185 ; thread_1_receive
    20161016121006316 ; thread_1_emit
    20161016121006751 ; thread_9_receive
     ...
```

Listing 7: Descriptor of regular expression used to parse the files.

```
{
    "numberOfgroups": 12,
    "valuePositions": [9,11],
    "keyPositions": [1,2,3,4,5,6,7],
    "nodePosition": 10,
    "regexp": "(\\d{4})-(\\d{2})-(\\d{2})T(\\d{2}):
               (\\d{2}):(\\d{2})\\.(\\d{3})\\+
               (\\d{4}) \\[(Thread-\\d+)-(.*)\\] .* \\[INFO\\]
               (emit|receive) (.*)"
}
                    ...
```

After the merge phase and before launching an instance of trace checking on Spark, DICE-TraCT builds the property that TCE has to use to check the logs. DICE-TraCT reads a template file containing the formula that has to be used to check, for the current experiment, the property called "idleTime" on the logs of *spoutA* and *spoutB*, as required by the JSON trace checking descriptor of Listing 1. All the template formulae that are used for (trace) checking standard predefined properties (like, for instance, idleTime) must be available before the execution of DICE-TraCT in the folder ./template. The template formula for *idleTime* used for the current validation is shown in Listing 8:

Listing 8: Template of the temporal logic formula used in the validation example.

```
    -G- ( emit -> (! emit) -U-[$a,$b] emit )
```

DICE-TraCT writes in a .sol file each property related to idleTime which now contains the events that appear in the merged log file and, instead of the markers `$a` and `$b`, the values for min and max that are specified in the JSON trace checking descriptor. The output of this phase consists of two files containg a Soloist formula Listing 9 is the formula related to *spoutA* and Listing 10 the one for *spoutB*.

Listing 9: Temporal logic formula used in the validation example built from the template for node *spoutA*.

```
-G- (thread_0_emit -> (! thread_0_emit) -U-[10,1000] (thread_0_emit)) &
-G- (thread_8_emit -> (! thread_8_emit) -U-[10,1000] (thread_8_emit))
```

Listing 10: Temporal logic formula used in the validation example built from the template for node *spoutB*.

```
-G- (thread_2_emit -> (! thread_2_emit) -U-[0,2000] (thread_2_emit)) &
-G- (thread_5_emit -> (! thread_5_emit) -U-[0,2000] (thread_5_emit))
```

Step 4

DICE-TraCT finally runs Spark by submitting the trace checking jobs to the local executor. Listing 11 shows the command line run by DICE-TraCT.

Listing 11: Command line to submit a spark job and execute trace checking.

```
 ./bin/spark-submit
--class it.polimi.krstic.MTLMapReduce.SparkHistoryCheck
--master spark://localhost:7077
--executor-memory 4g
--executor-cores 2
--num-executors 1
 ../mtlmapreduce/target/MTLMapReduce-0.0.1-[...].jar
<dicetcrat_folder>/spoutA.his
<dicetract_folder>/idleTimespoutA.sol
 output_file
--reader spark -l
```

A portion of the output on the console only related to the trace checking instance for *spoutA* is provided in Figure 18. The outcome shows that the property in file idleTimespoutA.sol does not hold for the execution currently analyzed.

```
Using Spark’s default log4j profile: org/apache/spark/log4j-defaults.properties
16/07/17 13:13:37 INFO SparkContext: Running Spark version 1.6.2
16/07/17 13:13:37 WARN Utils: Your hostname, lap-bersani resolves to a loopback address: 127.0.1.1;
                  using 192.168.0.3 instead (on interface wlp2s0)
16/07/17 13:13:37 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
16/07/17 13:13:37 INFO SecurityManager: Changing view acls to: bersani
16/07/17 13:13:37 INFO SecurityManager: Changing modify acls to: bersani
16/07/17 13:13:37 INFO SecurityManager: SecurityManager: authentication disabled; ui acls disabled;
                  users with view permissions: Set(bersani); users with modify permissions: Set(bersani)
16/07/17 13:13:37 INFO Utils: Successfully started service ’sparkDriver’ on port 34304.
16/07/17 13:13:37 INFO Slf4jLogger: Slf4jLogger started
16/07/17 13:13:38 INFO Remoting: Starting remoting
16/07/17 13:13:38 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://sparkDriverActorSystem@192.168.0.3:44061]
16/07/17 13:13:38 INFO Utils: Successfully started service ’sparkDriverActorSystem’ on port 44061.
16/07/17 13:13:38 INFO SparkEnv: Registering MapOutputTracker
16/07/17 13:13:38 INFO SparkEnv: Registering BlockManagerMaster

...

16/07/17 13:13:40 INFO HadoopRDD: Input split: file:/home/bersani/Tools/DICE-WP4/dicestrator/merge/spoutA.his:8508+8509
16/07/17 13:13:40 INFO HadoopRDD: Input split: file:/home/bersani/Tools/DICE-WP4/dicestrator/merge/spoutA.his:0+8508

...

16/07/17 13:13:40 INFO DAGScheduler: Job 1 finished: take at SparkHistoryCheck.java:405, took 0.636121 s
==========================
Formula 1 is false
==========================
16/07/17 13:13:40 INFO SparkUI: Stopped Spark web UI at http://192.168.0.3:4040
16/07/17 13:13:40 INFO MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
16/07/17 13:13:40 INFO MemoryStore: MemoryStore cleared
16/07/17 13:13:40 INFO BlockManager: BlockManager stopped
16/07/17 13:13:40 INFO BlockManagerMaster: BlockManagerMaster stopped
16/07/17 13:13:40 INFO OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
16/07/17 13:13:40 INFO RemoteActorRefProvider$RemotingTerminator: Shutting down remote daemon.
16/07/17 13:13:40 INFO RemoteActorRefProvider$RemotingTerminator:
                  Remote daemon shut down; proceeding with flushing remote transports.
16/07/17 13:13:40 INFO SparkContext: Successfully stopped SparkContext
16/07/17 13:13:40 INFO ShutdownHookManager: Shutdown hook called
16/07/17 13:13:40 INFO ShutdownHookManager: Deleting directory /tmp/spark-c7956eea-31b8-459e-b46e-6c28a69bf31f
16/07/17 13:13:40 INFO RemoteActorRefProvider$RemotingTerminator: Remoting shut down.
16/07/17 13:13:40 INFO ShutdownHookManager: Deleting directory
                  /tmp/spark-c7956eea-31b8-459e-b46e-6c28a69bf31f/httpd-a5954a3c-ecf6-4707-bf53-781d28671cda
```

Figure 18: Snapshot of the execution of spark-submit for node *spoutA*.

5 Conclusions

5.1 Summary

This deliverable presented the initial versions of ADTas well as DICE-TraCT. The goal of the first prototypes (M18) of these tools to enable the definition and reporting of anomalies present in monitored performance and quality related data from Big Data technologies. This is directly related to milestone MS3 ”DICE Tool Initial release”. Currently it has been tested on two frameworks. First the DICE-TraCT tool has been used on Storm worker logs while the Regression based AD has been tested on system metrics as well as run time metrics for Apache Spark.

Furthermore we have defined a connector between the anomaly detection tool and the DICE monitoring platform. This connector can be used both to retrieve datasets and to send detected anomalies to the Monitoring platform. It is important to note that at this time (M18) this integration is not fully functional, data sets can be created however anomalies are not transmitted. For M24, milestone MS4 ”Integrated Framework First Release” these functionalities will be finalized.

5.2 Further work

Both the Trace Checking and Regression based AD are at the moment standalone tools. They do not use the querying and anomaly signalling components from ADT. Future versions will have a tighter integration between them. In particular the Regression based AD will be integrated as one of the anomaly detection methods in ADT. The Trace Checking tool requires log traces in order to function so integration will most likely be in the form of anomaly signalling.

The current version of ADT was meant as a prototype which enables the definition of training/validating data as well several anomaly types. In M18 we did not define examples of contextual anomalies nor the format in which these are indexed in DMon. This will be detailed in future deliverable.

References

[1] Install matlab generated standalone application. `http://www.mathworks.com/products/compiler/`. Accessed: 2016-07-29.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[3] Marcello Maria Bersani, Domenico Bianculli, Carlo Ghezzi, Srdan Krstic, and Pierluigi San Pietro. Smt-based checking of SOLOIST over sparse traces. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 276–290, 2014.

[4] Andreas Brunnert, Andre van Hoorn, Felix Willnecker, Alexandru Danciu, Wilhelm Hasselbring, Christoph Heger, Nikolas Herbst, Pooyan Jamshidi, Reiner Jung, Joakim von Kistowski, Anne Koziolek, Johannes Kroß, Simon Spinner, Christian Vögele, Jürgen Walter, and Alexander Wert. Performance-oriented devops: A research agenda. Technical Report SPEC-RG-2015-01, SPEC Research Group — DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), August 2015.

[5] Emmanuel Candes and Terence Tao. The dantzig selector: Statistical estimation when p is much larger than n. *Ann. Statist.*, 35(6):2313–2351, 12 2007.

[6] Giuliano Casale, Pooyan Jamshidi, Tatiana Ustinova, Gabriel Iuhasz, Matej Artač, Tadej Borovšak, and Matic Pajnič. Dice delivery tools – initial version.

[7] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58, July 2009.

[8] Matthias Gander, Michael Felderer, Basel Katt, Adrian Tolbaru, Ruth Breu, and Alessandro Moschitti. Anomaly detection in the cloud: Detecting security incidents via machine learning. In Alessandro Moschitti and Barbara Plank, editors, *Trustworthy Eternal Systems via Evolving Software, Data and Knowledge*, volume 379 of *Communications in Computer and Information Science*, pages 103–116. Springer Berlin Heidelberg, 2013.

[9] Pooyan Jamshidi Marc Gil Christophe Joubert Alberto Romeu José Merseguer Raquel Trillo Matteo Giovanni Rossi Elisabetta Di Nitto Damian Andrew Tamburri Danilo Ardagna José Vilar Simona Bernardi Matej Artač Madalina Erascu Daniel Pop Gabriel Iuhasz Youssef Ridene Josuah Aron Craig Sheridan Darren Whigham Giuliano Casale, Tatiana Ustinova. D1.2 dice requirement specification.

[10] Robert L. Grossman, Stuart Bailey, Ashok Ramu, Balinder Malhi, Philip Hallstrom, Ivan Pulleyn, and Xiao Qin. The management and mining of multiple predictive models using the predictive modeling markup language. *Information & Software Technology*, 41(9):589–595, 1999.

[11] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The weka data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

[12] Jerome H. Friedman Ildiko E. Frank. A statistical view of some chemometrics regression tools. *Technometrics*, 35(2):109–135, 1993.

[13] Gabriel Iuhasz and Daniel Pop. Monitoring and data warehousing tools – initial version. *DICE EU H2020 Project Deliverable*, 2016.

[14] AI Khuri, JA Cornell, and SS Sablani. Response surfaces: Designs and analyses, revised and expanded. *Drying Technology*, 15(5):1657–1658, 1997.

[15] Francesco Marconi Marcello M. Bersani, Madalina Erascu. D3.5 dice verification tools initial version.

[16] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of Machine Learning*. The MIT Press, 2012.

[17] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Comput. Netw.*, 51(12):3448–3470, August 2007.

[18] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[19] Hesam Sagha, Hamidreza Bayati, José Del R. Millán, and Ricardo Chavarriaga. On-line anomaly detection and resilience in classifier ensembles. *Pattern Recogn. Lett.*, 34(15):1916–1927, November 2013.

[20] Erich Schubert, Alexander Koos, Tobias Emrich, Andreas Züfle, Klaus Arthur Schmid, and Arthur Zimek. A framework for clustering uncertain data. *PVLDB*, 8(12):1976–1979, 2015.

[21] Robert Tibshirani. Regression shrinkage and selection via the lasso: a retrospective. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(3):273–282, 2011.

[22] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., 1st edition, 2009.

[23] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.