Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

Quality anomaly detection and trace checking tools - Final Version

Deliverable 4.4

Deliverable:	D4.4
Title:	Quality anomaly detection and trace checking tools
Editor(s):	Gabriel Iuhasz (IEAT)
Contributor(s):	Gabriel Iuhasz (IEAT), Ioan Dragan (IEAT), Giuliano Casale (IMP), Tatiana Ustinova (IMP), Marcello Bersani (PMI), Ismael Torres (PRO)
Reviewers:	Andrew Phee (FlexiOPS), Vasilis Papanikolaou (ATC)
Type (R/P/DEC):	DEM
Version:	1.0
Date:	31-July-2017
Status:	Final version.
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright 2017, DICE consortium All rights reserved

The DICE project (February 2015-January 2018) has received funding from the European Unions 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. With the final versions of these tools we have created a comprehensive and extensible yet lightweight solution which can be used for quality and performance related contextual and sequential anomaly detection. We have done this by implementing the architecture and workflow for ADT as well as TraCT defined during the course of the project. 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 the overall architecture and 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
DICE-TraCT	DICE Trace-checking Tool
DMon	DICE Monitoring
D-VerT	DICE Verification Tool
ELK	Elasticsearch, Logstash and Kibana
IDE	Integrated Development Environment
LM	Log Merger
PMML	Predictive Model Markup Language
TCE	Trace Checking Engine
UML	Unified Modelling Language

Contents

List of Figures

List of Tables

Listings

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:

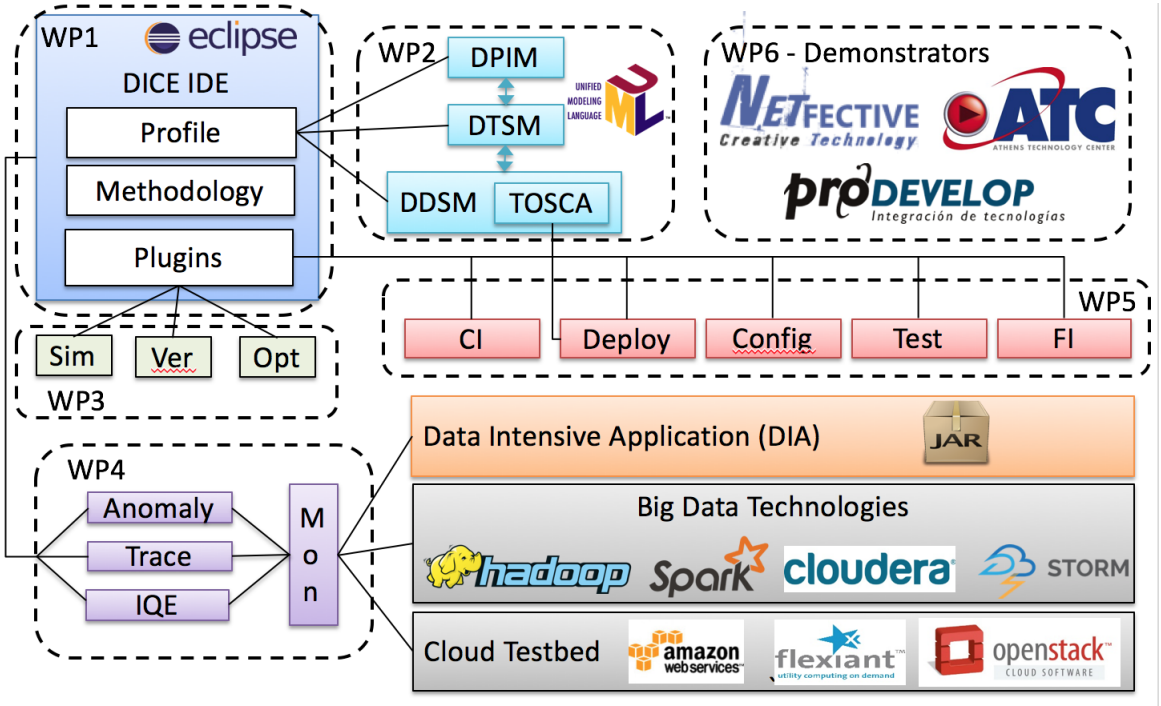

Figure 1: Summary view of the project methodology.

This deliverable presents the final 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, Storm as well as NoSQL databases such as MongoDB and Cassandra . 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 signaling anomalous behavior 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.

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 [17]
- 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 behavior 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) signaled 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 behavior*

Runtime application behavior 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 deliverable D4.1 [17], 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 use 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 Ad-hoc feature vector are definable in ADT however no direct link to the DICE Profile exists. These feature vectors have to be manually added by the user.
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 [5] 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 its input data.
Deployment and testing tools, Define a deployment and testing toolset to accelerate delivery of the application.	The final versions of the tools use setuptools for installation. This method allows easy installation on most operating systems.
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 and are using an Apache 2.0 license scheme.
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 and POSIDONIA Operations will use the anomaly detection.
Dissemination, communication, collaboration and standardisation, Promote visibility and adoption of project results through dissemination, communication, collaboration with other EU projects and standardization 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.

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

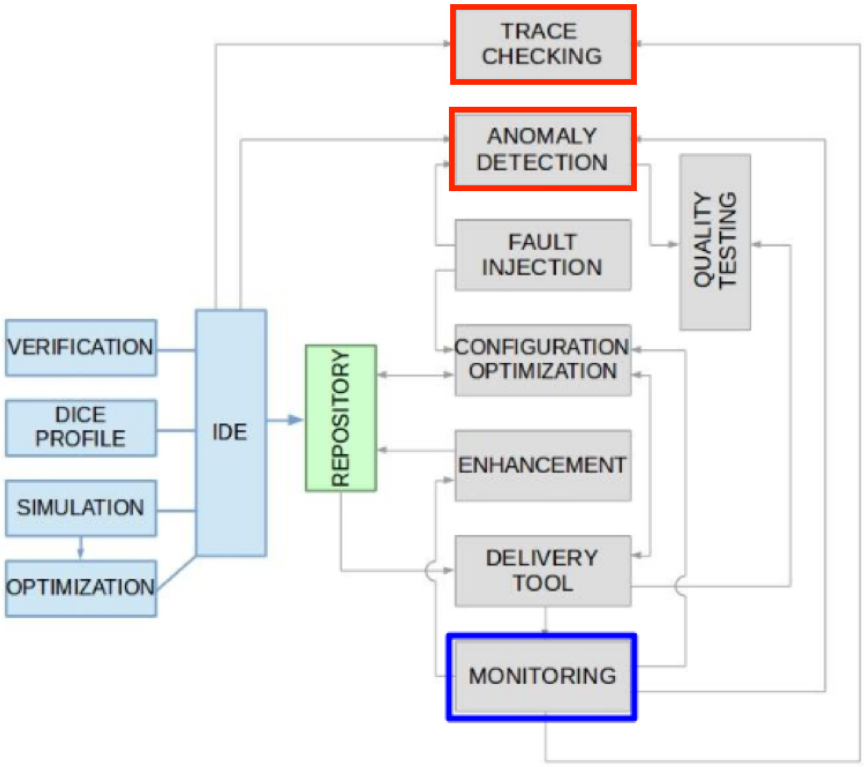

Figure 2: DICE Overall architecture.

1.4 Achievements of the period under report

Overview of the main achievements in the reported period:

- Final versions of ADT, Trace Checking and regression based AD tools
- Validation of tools using toy applications and use cases
- Integration of tools into the DICE ecosystem
- Supervised methods for anomaly detection
- Grid search for automatic method parameter setting
- Checked performance of DMon (DICE Monitoring Platform) and ADT on container based solutions
- Outline of potential improvements

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 since last deliverable
- 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 [6, 10]. 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) which 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 [17]. 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 [10]. 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 the data which the anomaly detection techniques 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.

Since the last deliverable D4.3 [9] we have added native support for several Big data frameworks on which Data Intensive Applications (DIAs) are based. ADT now supports; Yarn, Spark, Storm, Cassandra, MongoDB. In addition to these we have also added native support for the Complex event processor (CEP) component from the Posidonia Operations usecase. Later additions are possible as the moccations necesary in order to add support for other technologies is limited to the Data Formatter component.

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.

During integration of the anomaly detection libraries we have encountered a few unforeseen difficulties. The performance of JVM based libraries such as Weka [16] and ELKI [23] is quite poor. This doesn't mean that the trained models are of a lower quality but rather their reliance on JVM creates some difficulties when running them from ADT. Because of this, we decided to focus our work on the libraries

[1] https://sematext.com/spm/

[2] https://www.datadoghq.com/

which are written or officially support bindings in Python. The JVM methods are still integrated and usable from ADT but they are not used on any use cases or experiments.

Since the last deliverable we have integrated TensorFlow [1] based deep learning methods into ADT. We have found that although there is quite a substantial documentation available for Tensorflow using it is quite difficult. The Keras[3] library is a high level neural network API which is capable of running on top of TensorFlow, CNTK[4] or Theano[5]. It is extremely user friendly and is very good at abstracting the underlying backend.

2.1.2 Anomaly detection methods

There are a wide range of anomaly detection methods currently in use [6]. These can be split up into two distinct categories based on how they are trained. First there are 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 labeled 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. Methods 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 [20, 21] such as: DBSCAN, ROCK, SNN FindOut, WaveCluster. The second assumption [6, 23] 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 unsupervised, or clustering based, method is largely based on 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.

The following paragraphs will present the available anomaly detection methods from ADT.

Unsupervised

KMeans [3] is one of the simplest unsupervised learning algorithms used for clustering. It is able to classify any given data set through a certain number of clusters which have to be defined a priori. Each of the user defined clusters will be represented internally by a centroid as far away from each other as possible. Then each data point is associated with the closest centroid. The next step is to recalculate new centroids and this process is repeated until the centroids do not move anymore. The goal of the algorithm is to minimize an objective function denoted by equation 1:

$$J = \sum_{j=1}^{k} \sum_{i=1}^{n} \left\| x_i^{(j)} - c_j \right\|^2 \tag{1}$$

where $\left\| x_i^j - c_j \right\|^2$ is the distance measure between a datapoint $x^{(j)}$ and cluster center c_j, is an indicator of the distance of the n data points from their perspective cluster center.

[3] https://keras.io/
[4] https://github.com/Microsoft/cntk
[5] https://github.com/Theano/Theano

DBSCAN [8] is a density based data clustering algorithm that marks outliers based on the density of the region they are located in. There are several advantages to using this algorithm. First, we don't have to specify the number of clusters a priori as opposed to KMeans. Secondly, and most importantly DBSCAN has the concept of noise or outliers which are in fact anomalies from the point of view of the DICE Methodology.

As with many algorithms DBSCAN has some limitations, some of which directly impact it's usefulness in DICE. First, because it is density based, its performance on large feature spaces is quite poor and the amount of data points categorized as noise is unrealistic. Second, performance is largely dependent on the distance measure used. If the standard euclidian distance is used the problem of high dimension data is greatly increased.

EM [24] can be used to generate the best hypothesis for the distribution of parameters for multimodal data. In this case we define the best hyphotesis as being the most likely one. This algorithm has been successfully used in some for or another in the detection of anomalies for differernt types of use cases [27].

Isolation Forest [18] 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. Since recursive partitioning can be represented by a tree structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node. This path length, averaged over a forest of such random trees, is a measure of normality and our decision function. Random partitioning produces noticeably shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies.

Supervised

Random Forest [4] is a meta estimator that fits a number of decision tree classifiers on various subsamples of a data-set. In order to improve accuracy and reduce overfitting it uses averaging. It can be used for classification as well as regression tasks. Bootstrap aggregation is used as a generalized training technique. Given a training set $X = x_1, ..., x_n$ with the desired features as $Y = y_1, ..., y_n$, bagging repeatedly (B times) and selecting random samples from the training set to which trees are fitted. For $b = 1, ..., B$:

1. B training examples from X,Y (denoted as X_b, Y_b)
2. Training of a regression tree f_b on X_b, Y_B

After training, predictions for unseen sample $x^{'}$ is made by averaging predictions from all distinct regression trees on $x^{'}$ as seen in equation 2.

$$f = 1/B \sum_{b=1}^{B} f_b(x^{'}) \tag{2}$$

Decision Trees [2] are used as predictive models for observation (represented by the branches) to conclusions (represented by leaves). In our case the leaves of a learned decision tree will present class labels while branches will represent conjunctions of features that lead to the labels. Algorithms usually work top down by choosing a variable at each step that best splits the set of items. The *Gini* impurity is a measure of how often a randomly chosen element would be incorrectly labeled. It is the sum of the probability p_i of an item with label i being chosen times the probability $1 - p_i$ of a mistake in categorizing that item. To compute *Gini* of a set of items with T classes, suppose $i \in 1, 2, ..., T$ and let p_i be the fraction of items labeled with class i in the set, see equation 3.

$$I_G(p) = \sum_{i \neq k} p_i p_k \tag{3}$$

AdaBoost [7] is a meta learning algorithm specially designed to tackle the curse of dimensionality problem related to the features present in a data-set. During training, AdaBoost selects only the features which improve the predictive power of the model, thus leading to a reduction in the dimensionality of

the data and a great potential to reduce execution time. A boost classifier as used in AdaBoost is defined as seen in equation 4

$$F_T(x) = \sum_{t=1}^{T} f_t(x) \tag{4}$$

where f_t is a weak learner that takes an object x as input and returns a value indicating the class of the object. Each weak learner produces a hypothesis denoted by $h(x_i)$, for each sample in the training set. At each iteration t a learner is selected and a coefficient α_t is assigned to it such that the training error E_t (equation 5) is minimized.

$$E_t = \sum_i E[F_{t-1}(x_i) + \alpha_t h(x_i)] \tag{5}$$

Neural Networks [25, 13, 26] are based on biological neural network and are designed to mimic the way a biological brain functions. It has a series of artificial neurons which are interconnected with weighted connections. The number of layers (they are referred to as hidden layers) is a variable as well as the number of input and outputs of the neural network. In the context of anomaly detection, neural networks and deep neural networks have been extensively used. In DICE we aim to enable the use of not only feed forward but also of recurrent and even deep learning based neural networks.

2.1.3 Anomaly detection Implementation

The ADT is made up of a series of interconnected components that are controlled from a simple command line interface. An eclipse plugin was also developed for this tool which is integrated into the DICE IDE. A more user friendly interface is also possible.

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 deliverable [12].

First we have the data connector(called *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 used for both detected anomalies or as training dataset for creating predictive models. For each of these types of data *dmon-connector* creates a different index inside DMon. In deliverable 4.3 [9] we detailed how we created an anomaly index that was rotated every 24 hours. In the latest version of ADT this behavior has changed so that only one anomaly index is created and it is up to the user to decide when to purge it. This has been done in order to maximize the performance of the serving layer (i.e. DMon) as creating potentially hundreds of indexes of limited size is not efficient.

It is important to note that both DMon and ADT where tested on a container based deployment of Spark. This was done in collaboration with the CloudLightning research project. In CloudLightning they use an Apache Mesos cluster on top of which they run Marathon[6] container orchestrator. We managed to run DMon successfully on a container based deployment of Spark. No major issues where detected, DMon is fully capable of monitoring container based DIAs.

After the monitoring platform is queried the resulting dataset can be in JSON, CSV or RDF/XML. However, in some situations 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 dependent on the anomaly detection method used. One example of additional formatting is that of categorical feature encoding. Some anomaly detection methods require categorical data to be reencoded, in these situations ADT has the capability to encode these in a so called one hot encoding. This can be done automatically (ADT is able to detect categorical features) or for user defined features.

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

[6] https://mesosphere.github.io/marathon/

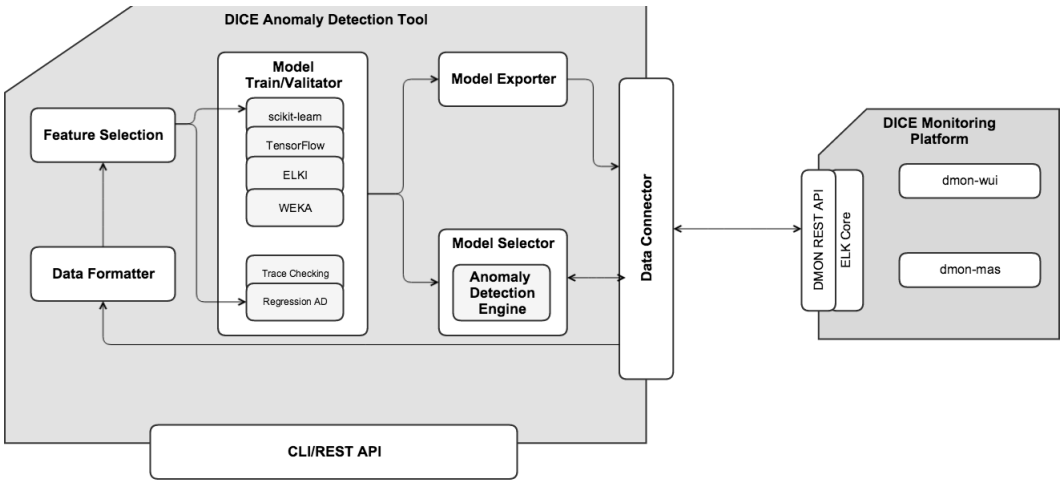

Figure 3: General overview of Anomaly Detection Stack.

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*[7] (from Weka) and *Wrapper Methods*. Because of the difficulties encountered during the integration of JVM based methods the PCA version implemented in scikit-learn is the recommended and the default one used.

The next two components (see Figure 3 model trainer and exporter) 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. It is also possible to ”side load” training data into ADT.

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. Crossvalidation can be ommited at user request.

Once validation is complete the *model exporter* component transforms the current model into a serialized loadable form. We will use the PMML [14] 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 uploaded into DMon. In fact the core services from DMon (specifically Elasticsearch) have the role of a serving layer from a lambda architecture. Both detected anomalies and trained models are stored in 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 15) 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 that 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.

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

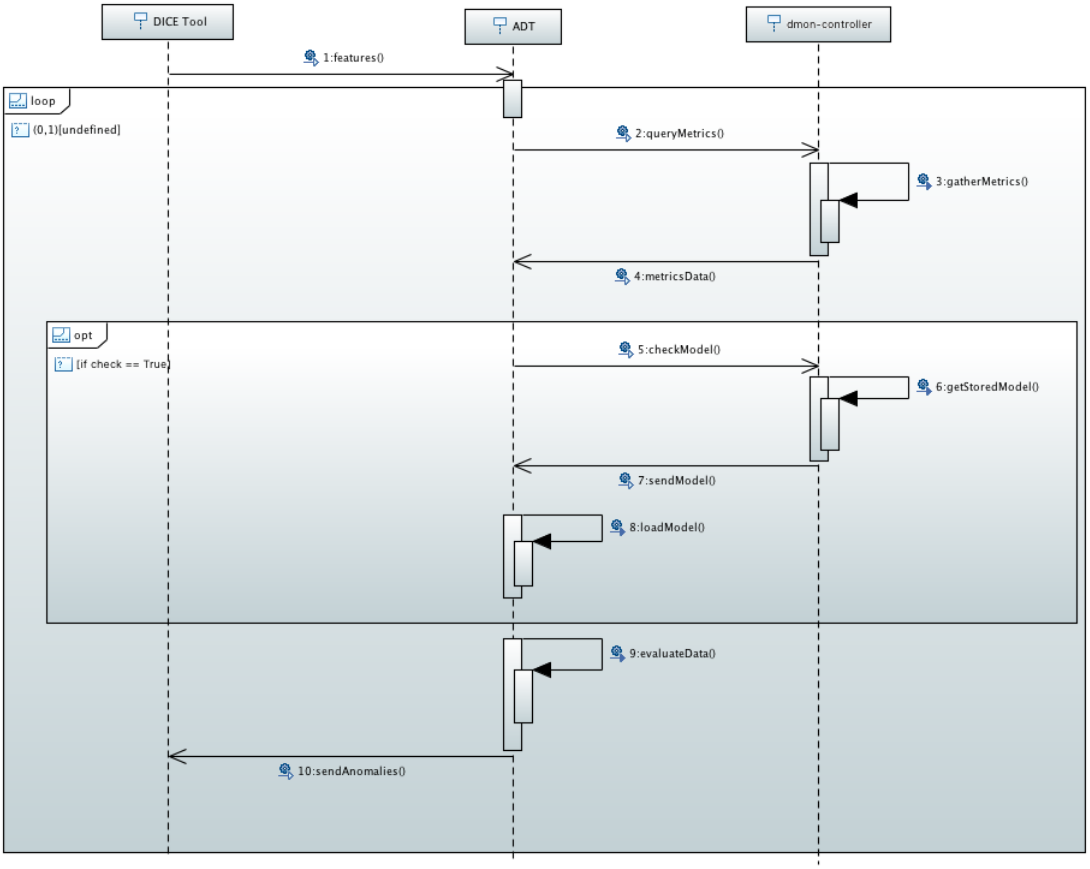

Figure 4: ADT Sequence diagram.

2.1.4 Configuration

In order to run ADP we must execute the following command:

```
$ python dmonadp.py <args>
```

There are currently two ways of configuring ADT. First we have the command line arguments and second we have the configuration file. The following paragraphs will deal with the command line arguments.

```
$ python dmonadp.py -h
```

Where the h argument lists a short help message detailing some basic usage of for ADT.

```
$ python dmonadp.py -f <file_location>
```

Where the f argument will define the configuration file is location. If the config file is not defined it will use a default location with the name *dmonadp.ini*.

```
$ python dmonadp.py -e <es_endpoint>
```

Where the e argument allows the setting for the Elasticsearch endpoint. It is important to note that ADT is tightly integrated with DMon and requires both the controller and Elasticsearch endpoints.

```
$ python dmonadp.py -a <query> -t -m <method> -v <folds> -x <model>
```


The query string is represented by a and is used to generate the final DMon query. The resulting data will be used for training. The query is a standard elasticsearch query[8] containing also the desired timeframe for the data. Also it can be in the form of the aggregated preset queries detailed later in this deliverable.

The t parameter if set instructs ADT to initiate the training of the predictive model. While m represent the method name used to create the predictive model (or clusterer). In order to run cross-validation the v parameter has to be set and will run for the number of fold defined by the use. In order to export the model in PMML format the x parameter has to be set together with the model name. This user defined model name is used together with the model to generate the final model name. It is important to note that this name will be used to identify the model thus the user has the obligation to give it a suitable name. The last two arguments namely v and x, are optional and can be omitted.

```
$ python dmonadp.py -a <query> -d <model>
```

By setting the d parameter ADT will predict use the user defined predictive model to detect anomalies in the incoming monitoring event stream.

2.1.5 Configuration File

The configuration file allows the definition of all of the arguments already listed. The settings passed as command line arguments will override any settings written in the configuration file. The file uses an ini format which makes integration into the DICE IDE much simpler as this file format is extensively used in Eclipse. Listing 1 contains an example configuration file.

Connector

The **Connector** section sets the parameters for use in connecting and querying DMon:

-
- *ESEndpoint* - sets the current endpoint for DMon, it can be also in the form of a list if more than one elasticsearch instance is used by DMon
- *ESPort* - sets the port for the elasticsearch instances. It is important to note that this setting is important for anomaly index creation and updating.
- *DMonPort* - sets the port for DMon
- *From* - sets the first timestamp for query, the use of time arithmetic of the form "now-2h" is also supported.
- *To* - sets the second timestamp for query, lower bound
- *Query* - defines what metrics context to query from DMon
  - each metric context is divided into subfields as follows:
    - **yarn** - cluster, nn, nm, dfs, dn, mr
    - **system** - memory, load, network
    - **spark** - executor, driver
    - **storm** - all
    - **cassandra** - all
    - **mongodb** - all
    - **userquery** - the user can define a manually created elasticsearch query which will be use instead of the aforementioned aggregated queries
- *Nodes* - list of desired nodes, if nothing specified than uses all available nodes

[8] https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html

- *QSize* - sets the query size (number of instances), if set to 0 then no limit is set
- *QInterval* - sets aggregation interval

Each large context is delimited by ";" while each subfield is divided by ",". Also QInterval must be set to the largest value if query contains both system and platform specific metrics. If the values are two far apart it may cause issues while merging the metrics. In this case ADT will issue a warning to the user. Another important point is that in some circumstances if the interval value is set to low DMon will not be able to successfully aggregate the Elasticsearch response which might cause unwanted behavior. In order to fix this the interval should never be set lower than the interval set for the monitoring auxiliary components from DMon.

The *userquery* aggregation option was added in the final version of ADT so that users which might use different technologies than the ones officially supported by DICE can use the anomaly detection platform. The query must be in JSON format and be stored in the queries directory. An example query can be found in listing 2.

Mode

The *Mode* section defines the mode in which ADP will run. The options are as follows:

- *Training* - If set to True the selected method will be trained using metrics collected from DMON
- *Validate* - If set to True the trained methods are compared and validated
- *Detect* - If set to True the trained model is used to decide if a given data instance is an anomaly or not.

Filter

The *Filter* section is used to filter the collected data and can be used to select which features are used for anomaly detection. The options are as follows:

- *Columns* - Defines the columns to be used during training and/or detecting. Columns are delimited by ";"
- *Rows* - Defines the minimum (using ld) and maximum (using gd) of the metrics. The time format used is utc.
- *DColumns* - Defines the columns to be removed from the dataset.

Detect

The *Detect* section is used for selecting the anomaly detection methods for both training and detecting as follows:

- *Method* - sets the desired anomaly detection method to be used (available 'skm', 'em', 'dbscan', isolationforest, randomforest, decisiontree, adabost, neural)
- *Type* - type of anomaly detection (available 'clustering', 'classification)
- *Export* - name of the exported predictive/clustering model
- *Load* - name of the predictive/clustering model to be loaded

If 'Export' and 'Load' is set to the same value then one a model is trained it will be automatically loaded and used to detect anomalies. If the model is not yet available then detect will try every 30 seconds until it detects a model with the given name. Since D4.3 [9] we have implemented a set of classification algorithms in addition to the clustering and density based ones.

Point

The Point section is used to set threshold values for memory, load and network metrics to be used during point anomaly detection. This type of anomaly detection is run even if 'Train' and 'Detect' is set to False.

Miscellaneous

The *Misc* section is used to set miscellaneous settings which are as follows:

- *heap* - sets heap space max value for all JVM based AD method implementations
- *checkpoint* - If set to false, all metrics will be saved as csv files into the data directory otherwise all data will be kept in memory as dataframes. It is important to note the if the data is kept in memory processing will be much faster. If checkpointing is activated each pre-processing and processing step will yield a csv file which serves as a checkpoint. If an operation fails or the platform crashes it will resume from the last checkpoint.
- *delay* - sets the query delay for point anomalies
- *interval* - similar to delay however it sets the query interval for complex anomalies
- *resetindex* - if set to True the 'anomalies' index will be reset and all previously detected anomalies will be deleted.

2.1.6 Method Settings

The *MethodSettings* section of the configuration files allows the setting of different parameters of the chosen training method. These parameters can't be set using the command line arguments and if not defined will use default values. Some algorithms such as KMeans and EM will not be covered in detail in this deliverable as during testing these algorithms have proven to be of limited use for detecting anomalies. There available parameters can be found in the official ADT wiki[9].

Unsupervised Methods

As mentioned in section DBSCAN is a density based data clustering algorithm that marks outliers based on the density of the region they are located in. For this algorithm we support two versions. The first one is base on the Weka implementation of the algorithm while the second one is based on the scikit-learn implementation. During testing we have found that the Weka implementations are not that computationally efficient especially when considering the fact that they require the starting of a Weka JVM as a subprocess from inside ADT which can then execute training and validation. The two implementation have the following method settings:

DBSCAN Weka[10]

- *E* - epsilon which denotes the maximum distance between two samples for them to be considered as in the same neighborhood (default 0.9)
- *M* - the number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself (default 6)
- *D* - distance measure (default weka.clusterers.forOPTICSAndDBScan.DataObjects.EuclideanDataObject)

DBSCAN scikit-learn[11]

- *eps* - epsilon which denotes the maximum distance between two samples for them to be considered as in the same neighborhood (default 0.5)

[9] https://github.com/dice-project/DICE-Anomaly-Detection-Tool/wiki
[10] http://weka.sourceforge.net/doc.packages/optics_dbScan/weka/clusterers/DBScan.html
[11] http://scikit-learn.org/stable/modules/generated/sklearn.cluster.DBSCAN.html

- *min_samples* - The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself (default 5)
- *metric* - metric used when calculating distance between instances in a feature array (default euclidean)
- *Algorithm* - the algorithm used by the nearest-neighbor module to compute pointwise distance and find nearest neighbor (default auto)
- *leaf_size* - leaf size passed to BallTree or cKDTree, this can affect the speed of the construction and query, as well as the memory required to store the tree (default 30)
- *p* - the power of the Minkowski metric used to calculate distance between points (default None)
- *n_jobs* - the number of parallel jobs to run (default 1, if -1 all cores are used)

IsolationForest[12] implementation used is the one found in the scikit-learn library. It has the wollowing parameters:

- *n_estimators* - number of base estimators in the ensemble (default 100)
- *max_samples* - number of samples to draw to train each base estimator (default auto)
- *contamination* - the amount of contamination of the dataset, used when fitting to defined threshold on the decision function (default 0.1)
- *max_features* - the number of features to draw to train each base estimator (default 1.0)
- *boostrap* - if true individual trees are fit on random subsets of the training data sample with replacements, if false sampling without replacement is performed (default false)
- *n_jobs* - the number of jobs to run in parallel for both fit and predict, (default 1, if -1 all cores are used)

Supervised Methods

Random Forest parameters are:

- *n_estimators* - number of trees in the forest (default 10)
- *criterion* - function to measure the quality of the split; supported criteria are ”gini” for Gini impurity and ”entropy” for the information gain (default gini)
- *min_sample_split* - minimum number of samples required to split an internal node; if *int* it is considered the minimum sample if *float* it is a percentage (default 2)
- *min_sample_leaf* - minimum number of samples required to be at a leaf; if *int* it is considered the minimum sample if *float* it is a percentage (default 1)
- *min_weight_fraction_leaf* - minimum weighted fraction of the sum total of weights required to be a leaf node (default 0)
- *bootstrap* - whether bootstrap samples are used when building trees (default True)
- *n_jobs* - the number of jobs to run in parallel for both fit and predict, (default 1, if -1 all cores are used)
- *random_state* - if *int* it is the seed used by the random number generator if None uses np.random (default None)

[12] http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html

Decision Tree parameters are:

- *criterion* - function to measure the quality of the split; supported criteria are "gini" for Gini impurity and "entropy" for the information gain (default gini)
- *splitter* - strategy used to chose the split at each node; strategies supported are "best" for chose the best split and "random" to choose the best random split.
- *max_features* - number of features to consider when looking for the best split (default None):
  - if *int* represents the number of features
  - if *float* represent the percentage of features at each split
  - if "auto" then $max_features = sqrt(n_features)$
  - if "sqrt" then $max_features = sqrt(n_features)$
  - if "log2" then $max_features = log2(n_features)$
  - if None then $max_features = n_features$
- *min_sample_split* - the minimum number of samples reqired to split an internal node; if *int* it is considered the minimum sample if *float* it is a percentage (default 2)
- *min_weight_fraction_leaf* - minimum weighted fraction of the sum total of weights required to be a leaf node (default 0)
- *random_state* - if *int* it is the seed used by the random number generator if None uses np.random (default None)

AdaBoost parameters are:

- *n_estimator* - number of trees in the forest (default 10)
- *random_state* - if *int* it is the seed used by the random number generator if None uses np.random (default None)
- *learning_rate* - learning rate shrinks the contribution of each classifier (default 1.)

Neural Network parameters are:

- *hidden_layer_sizes* - tuple, represents the number of elements in the hidden layer (default (100,))
- *max_iter* - maximum number of iterations until convergence ($1e-4$) or this number of iterations
- *activation* - activation function for hidden layers (default relu):
  - 'identity', no-op activation $f(x) = x$
  - 'logistic, sigmoid function $f(x) = 1/(1 + exp(-x))$
  - 'tanh', hyperbolic tan function, $f(x) = tanh(x)$
  - 'relu', rectified linear unit function, $f(x) = max(0, x)$
- *solver* - solver for weight optimization:
  - 'lbfgs' is a quasi-Newton method optimizer
  - 'sgd' stochastic gradient descent
  - 'adam' stochastic gradient-based optimizer proposed by Kingma, Diederik and Jimmy Ba
- *batch_size* - size of mini-batches for stochastic optimizer
- *learning_rate*- learning rate schedule for weight updates (default constant):

  - 'constant' constant learning rate
  - 'invscaling' gradually decreases the learning rate at each timestep
  - 'adaptive' keeps the learning rate constant as long as training loss keeps decreasing

- *momentum* - momentum for gradient descent updates (default 0.9)
- *alpha* - L2 penalty (regularization term) parameter (default 0.0001)

This tool is still a work in progress. All commands and their behaviors are subject to changes. Please consult the official DICE repository changelog[13] to see any significant changes.

2.1.7 Requirements

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	✓	Using the data formatter
R4.24.5	Predictive Model saving for AD	MUST	✓	Is capable of generating PMML or serialized predictive models
R4.24.6	Semi-automated data labeling	Could	✓	Can be done via dmon-gen component.
R4.24.7	Adaptation of thresholding	Could	✓	User defined treasholding
R4.26.2	Report generation of analysis results	Should	✓	Local generation of report is possible.
R4.36	AD between two versions of DIA	MUST	✓	Querying based on DIA tags
R4.37	ADT should get input parameters from IDE	MUST	✓	DICE IDE plugin

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

In Deliverable D4.3 DICE-TraCT has already been presented as a stand-alone application, taking part of the definition of the Anomaly Detection component in the DICE ecosystem. DICE-TraCT has the purpose of enabling log analysis of the DIA implemented in Apache Storm by means of trace-checking techniques. To this end, DICE-TraCT exploits a direct connection to the monitoring platform (DMon) which is used to retrieve and collect log traces of the application currently analysed. The architecture presented in D4.3 has already introduced the fundamental connections between DICE-TraCT and the IDE component and between DICE-TraCT and Dmon, not yet implemented at the time of delivering the

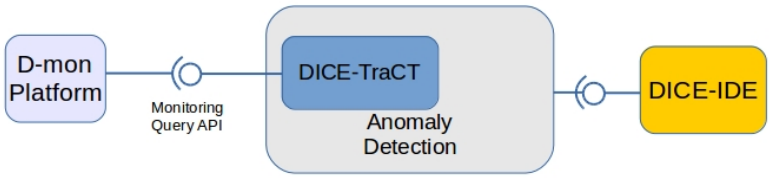

Figure 5: Dependencies between DICE-TraCT, DMon and the IDE component

document. The following Figure 5 shows the architectural connections among the three components.

By means of suitable API methods, available from Dmon (depicted on the left-hand side), DICE-TraCT can:

- activate a log monitoring session on the current application running in the deployed Storm topology by calling

  ```
  POST /dmon/v1/overlord/storm/logs
  ```

- visualize the list of the available logs that can be obtained from Dmon by calling

  ```
  GET /dmon/v1/overlord/storm/logs
  ```

- collect the logs of the application that were stored in the specified session and that are available in the platform

  ```
  GET /dmon/v1/overlord/storm/logs/{log_name}
  ```

The REST calls required to implement such functionalities are arguments of deliverable D4.3 and will not further explained in this document.

On the right-hand side of Figure 5, the connection between DICE-TraCT and the IDE has the purpose of enabling the trace-checking analysis by making the trace-checking service available to the DICE user, who enforces the iterative refinement of the DIA application through the quality analysis and enhancement tools.

This section elaborates on the implementation of the two connectors and shows the definition of the core API that DICE-TraCT supplies as a REST service. In addition, besides the architectural relations existing among the components required by DICE-TraCT to perform the log analysis, the second argument of the section is related to the extensions implemented in the last version of the tool and that have been included in the M30 release. In particular, these improvements affect one subcomponent of DICE-TraCT called Trace Checking Engine (TCE). According to Figure 6 in D4.3, DICE-TraCT is composed of three main components: DICE-TraCTor, Trace Checking Engine and Log Manager (LM).

- Trace Checking Engine (TCE) actually performs the trace analysis. The input is a time stamped log of events and a property to analyze; the output is the result of the evaluation of the property over the specified log.

- Log Merger (LM) collects the logs that undergo the analysis and splits them into various smaller logs, each one specific for a node of the Storm topology in analysis. Storm worker log might contain more than one sequence of events, each one associated with an executor spawned in that worker. Therefore, to reduce the size of the analyzed logs, that can entail an onerous trace-checking analysis, DICE-TraCT first manipulates the collected logs to aggregate all the relevant events generated by a node (or a subset of the nodes in the topology) into a new smaller log trace.

[13] https://github.com/dice-project/DICE-Anomaly-Detection-Tool/wiki/Changelog

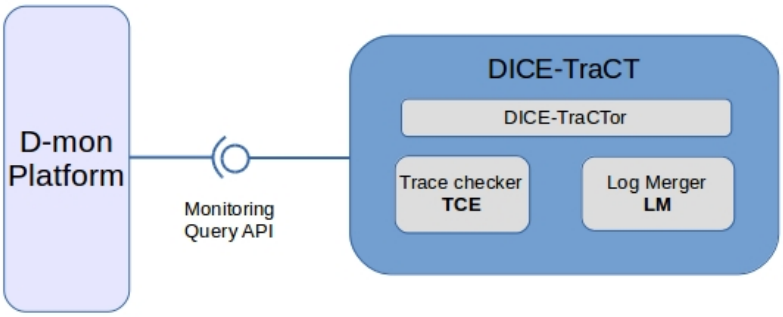

Figure 6: DICE-TraCT architecture

- DICE-TraCTor coordinates all of the activities carried out by DICE-TraCT. It manages the trace-checking request from the IDE, instructs LM and TCE and handles the communication with the monitoring platform.

The modification affecting TCE allows the use of different trace-checking engines. In the previous version, TCE only dealt with one external solver whereas in M30 release TCE is able to select the most appropriate engine based on the log analysis that the user wants to carry out.

2.2.1 DICE-TraCT REST API

DICE-TraCT can be accessed through a POST call with the following input data:

- parameter *ip* and *port* specify the IP address and the port where the DICE-TraCT service is deployed and running;
- the payload is a json descriptor which defines the trace-checking analysis to be carried out. The format of the json descriptor has been already shown in deliverable D4.3 and it is examined in detail afterwords.

The blue component in Figure 6 is implemented by a Flask [14] module `dicetractservice.py`, shown in Figure 7. DICE-TraCT is implemented by function `dicetract()` which is associated with the POST method `/run`. Line 1 of Figure 7 specifies that `/run` is a POST method. DICE-TraCTor component of DICE-TraCT is called at line 6 through function `dicetractor()`, which is invoked with parameter `request.get_json()` returning the JSON payload of the call.

Assuming that DICE-TraCT is deployed at *dicetracturl* on port 5050 and that the monitoring platform is deployed at *dmonurl* at port *dmonport*, an example of a method call can be the following:

```
POST http://dicetracturl:5050/run?ip=dmonurl&port=dmonport
```

The payload associated with the POST call is obtained, at line 6, through the `request.get_json()` function which is specified as a parameter of `dicetractor()` function. An example of a possible payload that DICE-TraCT can manage is shown if Figure 8. It specifies two trace-checking tasks for the nodes “word” and “exclaim1” of ExclamationTopology, a simple Storm benchmark application which is distributed within the Storm framework and that was used to validate DICE-TraCT.

The json descriptor consists of two main items: the topology name, specified by the key “topology-name”, and a list of descriptors of the trace-checking instances for each node in the tuple associated with the key “nodes”. Each node descriptor provides the following data.

- The name of the node to analyze is specified in the “name” string. The specified identifiers must match the names that are used in the deployed topology and that appear in the log files associated with the events of the nodes in the topology.

[14] `http://flask.pocoo.org/`


```
@app.route('/run', methods=['POST'])
def dicetract():
        result = None

        if (request.args.get('ip') ... ):
                result = dicetractor(..., request.get_json())
        else:
                result = -1

        # return a string version of the list containing results
        if (result == -1):
                # send error to the caller
        else
                # send the response to the caller
```

Figure 7: POST method implementing DICE-TraCT main service in `dicetractservice.py`.

```
{
        "topologyname": "ATopology",
        "nodes": [
        {
                "name": "exclaim1",
                "type": "bolt",
                "parameter": "sigma",
                "timewindow": 3600,
                "method": "counting",
                "relation": "=",
                "designvalue": "0.8"
        },
        {
                "name": "word",
                "type": "spout",
                "parameter": "spoutrate",
                "timewindow": 3600,
                "timesubwindow": "36",
                "method": "average",
                "relation": "<",
                "designvalue": "0.3"
        }
        ]
}
```

Figure 8: An example of payload specifying two trace-checking analysis on spout "word" and bolt "exclaim1" of ExclamationTopology.

- The "type" of node can be either a "spout" or a "bolt" (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 parameter (or property) that the analysis has to consider for the node. The current implemen-

tation of DICE-TraCT only deals two two distinct parameters for the quality analysis assessment and they are tightly related to the verification analysis that is carried out by D-VerT according to the line presented in deliverables D3.5 DICE Verification tool Initial version. In particular, the two possible parameters are: "spoutrate" and "sigma". "Spoutrate" is the emitting rate of a spout node, defined as the number of tuple emitted per second, and "sigma" is the ratio between the number of tuples produced by a bolt node in output and the number of tuples that it has are received in input.

- The length of the time window limiting the trace-checking analysis. The size, specified by the value of the key "window", determines how many log events are considered to carry out the evaluation of the property defined by "parameter". The unit measurement is the millisecond.

- The kind of the analysis, that DICE-TraCT has to use to calculate the value of the property, is specified in the method keyword. The analysis is performed by running the engine that is specific for the property to be verified. This feature was not implemented in the first release of DICE-TraCT and it represents one improvement that is implemented for the final release. The first implementation of DICE-TraCT exploited a trace-checking procedure that is based on a logical approach. Logical languages involved in the trace checking analysis are usually extensions of metric temporal logics which offer special operators called aggregating modalities. These operators are useful to count events in the logs or calculate an average on the occurrences over a certain time window and compare the value with a given threshold. Therefore, the outcome is always a boolean answer. DICE-TraCT is based on Soloist language[3]. The current implementation of DICE-TraCT can also call a simpler trace-checker (called SimpleTC) than the one for Soloist, that is developed to provide quantitative information calculated from the logs: specifically, the value of the emit rate for the spouts and the sigma of bolts can be directly extracted from the logs and provided to the user. The possible values of the key "method" are:

  counting : it enables the evaluation of the "parameter" by means of the so called Counting formulae of the Soloist logic. This choice will call the Soloists trace-checker (`https://bitbucket.org/krle/mtlmapreduce/overview`).

  "average" : it enables the evaluation of the "parameter" by means of the so called Average formulae of the Soloist logic. This choice will call the Soloists trace-checker (`https://bitbucket.org/krle/mtlmapreduce/overview`).

  quantitative : it enables the quantitative analysis of the logs which extracts the value of "parameter". This method will call SimpleTC engine.

- The analysis instantiated by the Soloist trace checker is designed to provide a boolean result calculated by comparing the number of occurrences of an event in the log with a user defined threshold. The relation "$<$", "$=$" or "$>$" is specified by "relation" and the threshold value is provided in "designvalue" key.

For each instance descriptor in the JSON payload, DICE-TraCT carries out the proper trace-checking analysis. The trace-checking engine and the Soloist formula for the analysis are identified in DICE-TraCT by means of a specific hierachy of classes defined in the module `formula.py`.

- Formula is an abstract class and the root of the hierarchy.

- SpoutRateAverage and SpoutRateCounting are the subclass of Formula representing the analysis is the emit rate of spout nodes by means of average and counting criteria.

- SigmaAverage and SigmaCounting are the subclass of Formula representing the analysis is the sigma value of bolt nodes by means of average and conting criteria.

- SigmaQuantitative and SpoutRateQuantitative are the two classes enabling the quantitative analysis of logs.

2.2.2 Trace Checking Engine - TCE

The core part of DICE-TraCT hinges on a specific object implementing class TCRunner, which is defined based on the trace-checking analysis specified in the payload of the REST call `/run`. The portion of code shown in Figure 9 is a part of function `dicestractor()` which in responsible of performing all the trace-checking instances in the `tc_instances`. Variable `tc_instances` is a TCRunner object which is instantiated with the information derived from the payload in variable `tc_descriptor`. `tc_instances` is an iterable object and each element in `tc_instances` is a (solvable) trace-checking instance that can be actually carried out by calling the engine associated with it by means of the function call `i.run()` at line 4. If no errors occurred while calling the trace-checking engine, the outcome of the analysis is appended to the result list in variable `result`, which is returned to the caller at the end of all the trace-checking tasks at line 12.

```
1  tc_instances = TCRunner(tc_descriptor)
2  for i in tc_instances:
3          try:
4                  i.run()
5                  if (i.getResult()):
6                          result.append(i.getResult())
7                  else:
8                          # return error
9          except Exception, err_msg:
10                 # return error
11 return result
```

Figure 9: Portion of code showing the use of the iterator pattern on the trace-checking instances obtained from the TCRunner object `tc_instances`.

Component TCE in DICE-TraCT is implemented by TCRunner class in the `dicetract.py` module. TCE has been improved in the M30 release to allow trace-checking analysis with Soloist trace-checking engine and with SimpleTC engine, the latter specifically developed to allow the extraction of quantitative values of emit rate for spout and sigma for bolts.

2.2.3 Trace-checking Runner

TCRunner returns an iterator on a set of trace-checking instances, each one associated with an element of the list in the value of the key "nodes" of the payloads provided with the call to `/run`. Each trace-checking instance represents a trace-checking problem that can be solved with one of the available engines. The solver is selected based on the subclass of Formula that contribute to the definition of the problem instance and that is derived from the description of the analysis in the payload (through the keys "parameter" and "method"). TCRunner includes a chain-of-responsibility defined by the available trace-checking engines that can be used to select the most appropriate engine to solve an instance.

The portion of code in Figure 10 is an excerpt of the implementation of the class TCRunner. The constructor method instantiates, at line 5 and 6 respectively, the two solvers: the first one is `sparkTC`, with an attrbute of class with class `SparkTCSolver()`, and the second one is `simpleTC`, with an attribute of class `SimpleTCSolver()`. In addition, it *declares* a chain-of-responsibility composed by means of the two solvers, at line 8. TCRunner implements method `next()`, which returns a trace-checking instance properly set with the formula to analyze and the node undergoing the analysis. This method actually realizes the iterator behavior of the class. The method call to `getRunnableTCInstance()`, at line 15, allows TCRunner to visit the chain-of-responsibility and obtain the suitable trace-checking solver that can manage the problem instance defined for node and the formula in analysis.


```
class TCRunner():

  def __init__(self, tc_descriptor):
    # Declaration of Chain-Of-Responsibility handlers
    self.sparkTC = SparkTCSolver()
    self.simpleTC = SimpleTCSolver()

    self.sparkTC.setSuccessor(self.simpleTC)

  def next(self):
    if (there are trace-checking instances to solve):
      # define the formula to solve based on tc_descriptor

      # return a TCInstance specifying the node and the formula
      return self.sparkTC.getRunnableTCInstance(node['name'], formula)
    else:
      raise StopIteration()
```

Figure 10: Class TCRunner and (i) the declaration of the chain-of-responsibility with `sparkTC` and `simpleTC` instances, (ii) the method `next()` implementing the iterator on the solvable trace-checking instances.

2.2.4 Trace-checking Instance

A trace-checking instance is represented by the class TCInstance. It *implements* the design pattern Chain-of-Responsibility to allow the selection of the trace-checking engine to be used for the resolution. Therefore, TCInstance is an abstract class providing

- two methods `setSuccessor()` and `getRunnableTCInstance()` implementing the pattern and
- the abstract method `run()`, that actually executes the solver on the problem instance and that is implemented by the concrete classes SparkTCSolver and SimpleTCSolver, being them associated with an existing trace-checking engine.

Figure 11 shows the main structure of the class

SparkTCSolver and SimpleTCSolver are the two implementations of TCSolver representing the Soloist trace-checker and the Simple trace-checker. Both implement the abstract method `canProcess()` based on their solving capabilities. The outcome of the method is a boolean that determines if the formula provided in the parameter of the call can be undertaken by the solver.

SparkTCSolver deals with the trace-checking instances that can be solved by the Soloist trace-checker, which is implemented in Spark; whereas SimpleTCSolver captures the instances that are solved by SimpleTC. For the sake of brevity, only an excerpt of SimpleTC is provided in Figure 12, as details on the Soloist trace-checker was already described in the previous deliverable D3.4. SimpleTC can handle formulae of type SigmaQuantitative and SpoutRateQuantitative. Hence, method `canProcess()` checks that the trace-checking instance, represented by the formula to be verified, has type SigmaQuantitative and SpoutRateQuantitative. Method `run()` actually launches the SimpleTC on the logs that has been obtained by component LM (see Figure 6), according to what has already been explained in deliverable D4.3.

2.2.5 Monitoring connectors

DICE-TraCT exploits two different log retrieval features. The first one is developed for testing purposes and does not require a running monitoring platform (“nodmon” mode). The second one, on the


```
class TCInstance():
  ...

  @abc.abstractmethod
  def canProcess(self, formula):
    pass

  @abc.abstractmethod
  def run(self):
    pass

  def setSuccessor(self, successor):
    self.successor = successor

  def getRunnableTCInstance(self, node, formula):
    if (self.canProcess(formula)):
      self.setTCInstance(node, formula)
      return self
    else:
      return self.successor.getRunnableTCInstance(node, formula)
```

Figure 11: Class TCInstance and the implementation of the Chain-of-Responsibility pattern methods `setSuccessor()` and `getRunnableTCInstance()`.

```
class SimpleTCSolver( TCInstance ):

  def __init__(self, nodename=None, formula=None):
  ...
    # declaration of a SimpleTC solver
    self.tc = SimpleTC()

  ...

  def canProcess(self, formula):
    if ( (type(formula) is SigmaQuantitative) or
         (type(formula) is SpoutRateQuantitative) ):
      return True
    else:
      return False

  def run(self):
    self.tc.run(self.__nodename, self.__formula)
```

Figure 12: Class SimpleTCSolver and the two methods `canProcess()` and `run()`.

other hand, implements the communication with a running instance of DMon that might be working locally or remotely. `RemoteDMonConnector` is the class in module `dicetract.py` that instruments the REST calls to DMon to retrieve the application logs from the monitoring platform. Figure 13 shows the fundamental methods that realize the functionality of the connector to the DMon platform.

- `getTopologyDescriptor()` is the main method that builds the topology descriptor associated with the analysis to be performed. A topology descriptor is a json file specifying the list

of all the log files where each node of the topology appers. The information included therein are fundamental to carry out the functionality of component LM (already detailed in deliverable D4.3).

- `availableStormLogs()` retrieves the list of all the collected logs available in DMon, by calling the method `GET /v1/overlord/storm/logs/`.
- `whichLog()` provides the capability for selecting the proper log files archive from DMon based on the characteristic of the trace-checking analysis specified by the user.
- `getStormLog()`: calls the method `GET /v1/overlord/storm/logs/{workerlog}` specifiying the "workerlog" to download. The obtained file is then used to perform the analysis.

```
1  class RemoteDMonConnector( AbstractDMonConnector ):
2
3    def getTopologyDescriptor(self, jsonscript, regexp_descriptor):
4
5      topology_descriptor = None
6
7      if (self.checkDMonOn()):
8        # build a topology_descriptor
9
10     return topology_descriptor
11
12   def whichLog(self, jsonscript):
13     # select the proper workerlog file to be downloaded from Dmon,
14     # based on the requirement in the jsonscript parameter
15
16   def availableStormLogs(self):
17     # call GET /v1/overlord/storm/logs/ of DMon to retrieve
18     # the list of available workerlogs
19
20   def getStormLog(self, workerlogToCheck):
21     # call GET /v1/overlord/storm/logs/{workerlogs} of DMon
```

Figure 13: `RemoteDMonConnector` class in `dicetract.py`

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: Configuration file for ADT

```
[Connector]
ESEndpoint:85.120.206.27
ESPort:9200
DMonPort:5001
From:1479105362284
To:1479119769978
Query:yarn:cluster, nn, nm, dfs, dn, mr;system
Nodes:
QSize:0
QInterval:10s

[Mode]
Training:true
Validate:False
Detect:false

[Filter]
Columns:colname;colname2;colname3
Rows:ld:145607979;gd:145607979
DColumns:colname;colname2;colname3

[Detect]
Method:skm
Type:clustering
Export:test1
Load:test1

[MethodSettings]
n:10
s:10

[Point]
Memory: cached:gd:231313;buffered:ld:312123;used:ld:12313;free:gd:23123
Load: shortterm:gd:2.0;midterm:ld:0.1;longterm:gd:1.0
Network: tx:gd:34344;rx:ld:323434

[Misc]
heap:512m
checkpoint:false
delay:2m
interval:15m
resetindex:false
```

Listing 2: Example user defined query

```
{
  "size": 0,
  "query": {
    "filtered": {
      "query": {
        "query_string": {
          "query": "collectd_type:\"load\" AND host:\"vm-cluster-storm4\"",
          "analyze_wildcard": true
        }
      },
      "filter": {
        "bool": {
          "must": [
            {
              "range": {
                "@timestamp": {
                  "gte": 1496940356413,
                  "lte": 1496954756413,
                  "format": "epoch_millis"
                }
              }
            }
          ],
          "must_not": []
        }
      }
    }
  },
  "aggs": {
    "2": {
      "date_histogram": {
        "field": "@timestamp",
        "interval": "1s",
        "time_zone": "Europe/Helsinki",
        "min_doc_count": 1,
        "extended_bounds": {
          "min": 1496940356413,
          "max": 1496954756413
        }
      },
      "aggs": {
        "1": {
          "avg": {
            "field": "shortterm"
          }
        },
        "3": {
          "avg": {
            "field": "midterm"
          }
        },
        "4": {
          "avg": {
            "field": "longterm"
          }
        }
      }
    }
  }
}
```

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 shown in the DICE IDE.
R4.28	Safety and privacy properties loading	MUST	✓	Log analysis is based on the kind of property the user chooses.
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.

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) [17]. 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 to specify the query string and the desired time frame. There are several operations made available by ADT which enable the reencoding and filtering of features as detailed in section 2.1.5. It is important to remember that the bigger the time frame specified the more accurate the anomaly detection becomes. The tradeoff is that querying as well as cluster creation become computationally more expensive the bigger the query time frame is.

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 labeled in order to create a viable training and validating data set. Once this is done the resulting predictive models can be easily applied at runtime.

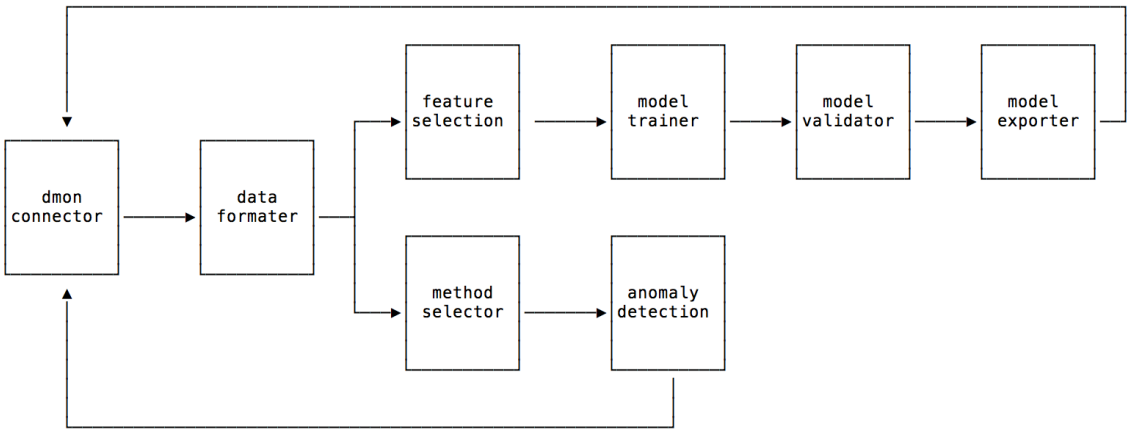

Figure 14: Anomaly Detection flow.

Figure 14 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 signaling 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.

3.1.1 Training Data

In deliverable D5.5 we have introduced the *dmon-gen* too which is used to define different Big Data and DIAs related jobs thus ensuring semi-supervised labeling of training data. In some instances this is not available. For instance if a developer is using the DICE solution to develop and deploy his DIA the

first versions of the application will not have known anomalies. In these kinds of instances unsupervised methods are extremely useful allowing the detection of anomalous events soley based on the avilable data.

There are several problems with this approach. The first problem is that we cannot distinguish between different types of anomalies, we only know that a particular event is anomalous when compared to the others. Secondly, the incidence of false positives can be significant. Because of this we have developed a method of boostraping labeled training data suing the output of unsupervised methods.

First we run the Isolation forest algorithm in order to detect anomalous events. Then we isolate the resulting anomalies and cluster these using DBSCAN. This will create clusters of anomalies, in essence creating labels for them. This is not an optimal solution as the clustered anomalies are still largely meaningless for the developer. It is meant to help the developer or architect in identifying the underlying cause of the anomalies. Finally, the clustered information is added back into the original data set thus creating a first iteration of a training set. This can later be fine tuned, anomalies can be added or removed as required.

Thus ADT facility aids in creating valid training sets for predictive model training. ADT is also able to accept manually created training and validation data sets. If no validation set is defined ADT can split the user defined labeled data set into two parts training and validation. At the same time the user can define what percentage of the original set is to be defined as the validation sets. All datapoints selected to be in the validation set are chosen randomly ensuring that during cross validation we do not fall prey to cherry picking.

3.1.2 Parameter Selection

During the development of ADT we have found that most algorithms have a large set of available parameters that can be fine tuned. This fine tuning requires extensive knowledge of the underlying algorithms which can result in low uptake of ADT as most novice user will be overwhelmed. In order to mitigate this as much as possible we have implemented a GridSearch method which allows the definition of the parameter space for any given method which is then used for an exhaustive search. At the end of this optimization step the best combaination is retained and used for the final predictive model creation.

This step has to be run only once as long as the data set has the same feature dimensionality. The resulting model has a very good chance to outperform any naively chose parameters. It requires no futher input from the user while at the same time guaranteeing exelent detection performance of all trained models. Sadly there is one downside to this method. It requires a lot of computational resources for large parameter search spaces. One solution is to define a set of parameter vectors to the grid search method which it then uses to perform the optimizations instead of just individual parameter values.

3.2 Trace Checking tool

DICE-TraCT has been developed to perform log analysis of Storm applications. It is used to assess the runtime behavior of a deployed application as log traces collected from the monitoring platform are analyzed to certify the adherence 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 new certification of correctness.

Trace-checking can be applied to supply information to the verification task carried out by D-VerT (details can be found in DICE Verification Tool - Initial version [19]). Verification of Storm topologies takes place at DTSM level on UML models enriched with information needed to represent the application behavior over time. To verify DTSM models with D-VerT, the designer must provide some parameter values that abstract the (temporal) behavior of spouts and bolts. Trace checking extracts from real executions the parameter values of the model used by D-VerT that are not available from the monitoring service of the framework, as they are inherently specific of the modeling adopted for the 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 might

be specified at the DPIM level by means of suitable classes, annotations and new ad-hoc constraints that impose the accessing restrictions applied to the users of the resources in a database. Checking the integrity of the deployed and running application can be achieved through the analysis of the application logs against suitable properties that are annotated by the user in the application model. An examination of this problem and a solution based on trace-checking techniques has been presented in [15] and also reported in the final methodology document D2.4 - Deployment abstractions - Final Version.

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

Integration

ADT has 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 15.

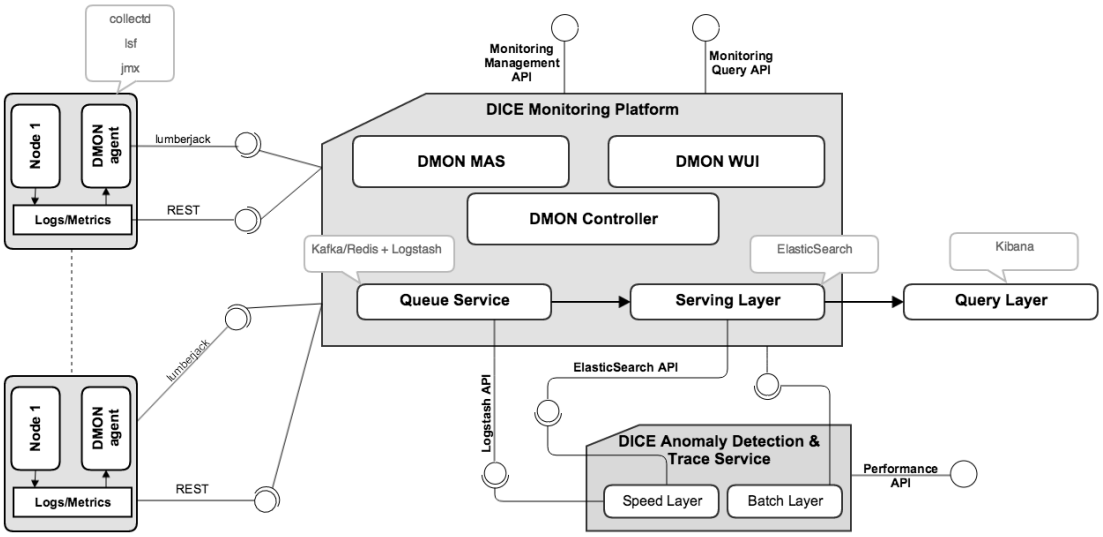

Figure 15: Anomaly detection integration with DMON.

As mentioned before the detected anomalies are 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 to serve the results of both the monitoring and anomaly detection on the same endpoint (DMon).

All anomaly detection methods used have their predictive models (be they clusterers or predictive models for classification) serialized or saved in PMML format. In order to completely implement a lambda type architecture we have to ensure that the trained models are versioned and saved inside DMon (serving layer). Figure 16 shows the REST API resources used for saving and versioning. This feature from DMon is a heavily modified version of the Artifact Repository from the FP7 MODAClouds research project [22].

The artifact are stored directly on the file system. The file hierarchy is directly mirrored from the URL structure shown in figure 16. This means that the folder structure will include folders for repositories, artifacts, versions and the models. Thus making interrogation extremely intuitive. Another bonus of using a simple file system based approach is the ability to use *rsync* as the synchronization mechanism between artifact repository models. In order to support multicloud environments the artifact repository is able to sync contents between different instances of DMon.

For the artifact repository to be fully usable in DICE we had to make 2 major modifications. First, we added the capability to store some metadata about the models stored in the repository pertaining to the

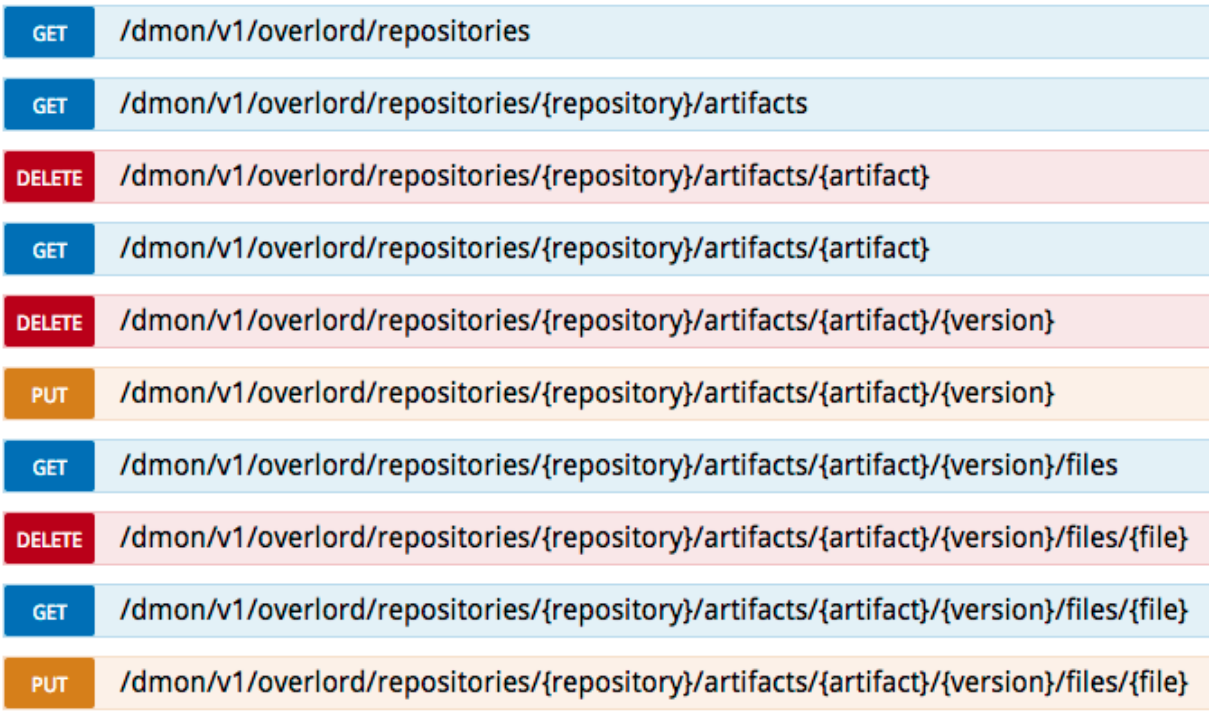

Figure 16: DMon Predictive model saving resources

shape of the data set (rows and columns), the method parameter settings used and finally the performance or score. This enables the fast checking of weather a stored predictive mode is usable on a particular data set and what it's overall score is. Versioning of the models is done via the naming convention used for the models. The first part of the name represents the model used for training, the second part is the unique identifier of the model (this can be user defined or by default is set as the application tag from DMon). The last part of a name is the timestamp of the model creation.

Second, we completely integrated the artifact repository into DMon as opposed to having a separate service. This allows a vastly simplified usage pattern and integration. Tools such as the ADT will only require the access point of DMon for both data fetching and anomaly reporting. From figure 16 we can see that inside DMon we organize models into repositories which can have several artifact which can have different versions.

For this version of ADT we have also implemented an Eclipse plugin) that is part of the integration with the DICE IDE as seen in figure 17. The plugins main goal is to generate the ADT configuration file and start the execution. All outputs from ADT (which are not sent to DMon) are shown inside this plugin. At this point we should mention that if ADT is run from Windows in some circumstance it will suffer from slow performance.

This is due to the multiprocessing module which behaves differently on Linux and Windows. In Linux multiprocessing uses the fork system call to create a child process with all the same resources as the parent while on Windows a new python interpreter has to be spawned and all the information needed by the child has to be passed as an argument. This difference is only evident during one preprocessing operation in ADT, namely the reencoding of a data set.

Validation

As mentioned in section some anomaly detection methods, more precisely the ones using supervised learning techniques, need labeled 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 labeling them is extremely impractical. We have detailed several method of ensuring that novice users are able to train and use anomaly detection methods using ADT, see section 3.1 for more details.

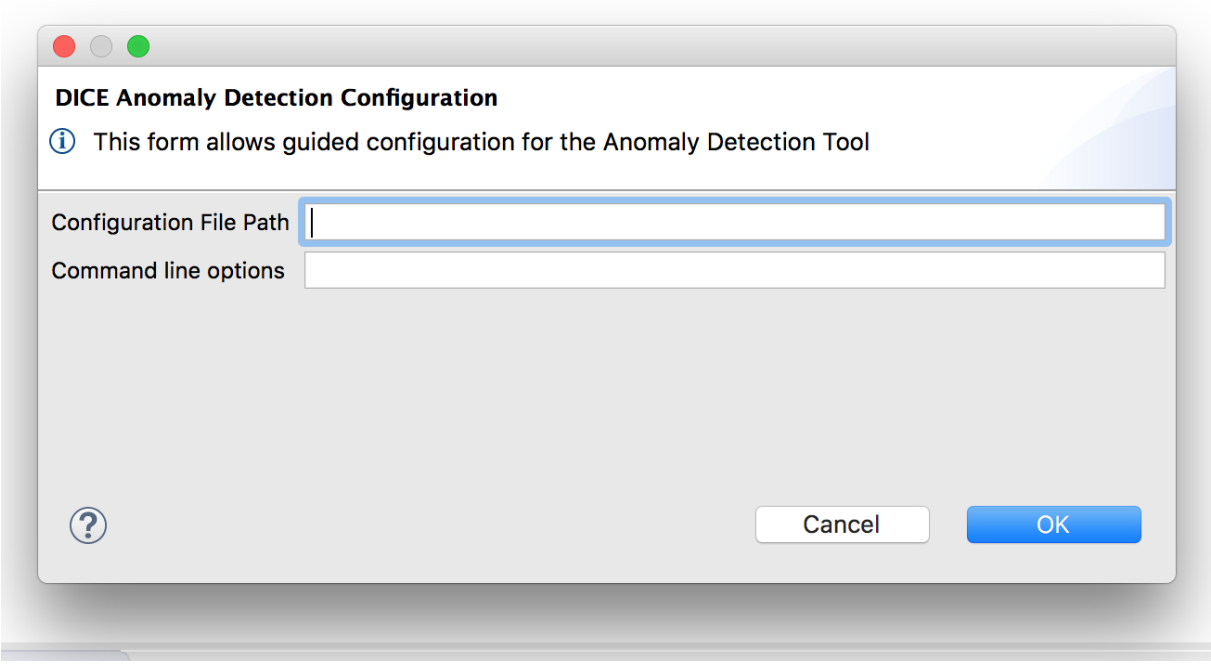

Figure 17: DICE IDE plugin for ADT

POSIDONIA Operations is an integrated port operations management system. Its mission consists on globally monitor vessels positions in real time to improve and automatize port authoritys operations. In the use case, some scenarios are considered. The deployment or the Posidonia system on the cloud considering different parameters, supporting different vessel traffic intensities, adding new business rules (high CPU demand), running simulation scenario to evaluate performance and quality metrics. Posidonia Operations core functionality is based on analysing a real-time stream of messages that represent vessels positions to detect and emit events that occur on the real world (a berthing, an anchorage, a bunkering, etc.).

One of the software components of the Posidonia Operations is the CEP, Complex Event Processing, engine which analyse all the AIS messages in order to detect patterns and create the corresponding events. One of the types of requirements of the Posidonia Operations use case is the Assessment of the impact in performance after changes in software or conditions. The changes in software and conditions include: (1) Adding, modifying and removing the rules of the existing CEPs,(2) Adding or removing CEPs (since we can have more than one CEP processing the incoming flow of AIS messages), (3)Increasing or decreasing the input message rate. The Anomaly Detection Tool is used to fulfill these requirements. During the development phases of the Posidonia use case, the Anomaly Detection Tool has been validated to detect anomalies related with the cost execution time of the different events that the CEP component analyses, this cost impact directly in the performance of the system.

As validation for ADT we used the POSIDONIA use case [11] as well as the Wikistat toy applications Storm topology related metrics.

The data available for the POSIDONIA usecase has 6 features in it, all extracted by DMon from raw log information. It is important to mention that not all features are useful when it comes to testing ADT. We eliminated/filtered the *host* and *ship* features as the former one is essentially a constant in the current deployment (having a cardinality of one). The latter represents the unique identifier of the ship to which the metrics are relating. ADT is designed to offer DIA performance related anomaly detection so we only need metrics related to what rules are being activate inside the Complex Event Processor (CEP) not what it processes.

Two features from the data set are categorical (component and method) because of this they required re-encoding. We first tried label encoding however it resulted in poor performance, this type of encoding usually leads to a significant loss of relationships between different data points. Much better results

Method	BScore	BTime	ParamSearch	CV Mean	CV STD	CV Time	Fscore	FTime
RF	0.68	0.0611	185.446	99.98%	0.05%	27.5004	1	4.5732
DT	0.53	0.0049	5.0348	99.97%	0.09%	0.0478	0.9989	0.0036
AD	0.51	0.2358	62.6346	100.00%	0.00%	0.4151	1	0.0548
NN	0.34	0.5640	1771.0435	100.00%	0.00%	0.2695	1.0	0.03283

Table 4: Experimental runs for CEP component

Metric	AdaBoost	DecisionTree	Random Forest
AIS_SENTENCE_LISTENER	0.1	0.193142068	0.153848567
RETRACT_OLD_AISGEOMDATA	0.1	0.000700426	0.005668693
SESSION	0.1	0.00990615	0.032303538
SIMPLE_ANCHOR_IN	0.1	0.052707564	0.196569172
SIMPLE_DOCK_START_OUT	0.1	0.003373742	0.035556067
SIMPLE_DOCK_STOP	0.1	0.091526082	0.208327863
STOP_OVER_IN	0.1	0.526665234	0.194793414
ms	0.3	0.121978734	0.172932687

Table 5: Feature Importance for different methods CEP

where obtained using the one hot encoding detailed in section 2.1.3. All subsequent experiments detail in this section are using this type of encoding.

With the help of the developers from the POSIDONIA use case we where able to manually label a dataset comprising over 4800 data points. For validation purposes we ran all supervised and unsupervised methods on this data set. We can see in table 4 the results of the first validations. First we ran a baseline where all methods had their parameters set to default values. After which we ran parameter optimization on all methods and executed a 10 fold crossvalidation with 30% of the dataset used for validation. We can see that the parameter optimization not only allowed us to optimize the predictive performance but also the required training time (*BScore* and *BTime* for the baseline and *FScore* and *FTime* for the best performing).

An interesting observation which can be made using ADT is the so called feature importance. It is in fact showing what the impact of each feature from the data set has on the classification model. Table 5 show the feature importance for the tree based classification methods. One surprising fact evident in this table is that although logically the most important feature in detecting anomalies for CEP would be the *ms* feature indicating how long a particular rule takes to fire. However, we can see in table 5 that although this feature has quite an impact on the predictive model it is not the most representative. This can be explained by the fact that some rules are much more prone to have abnormal behavior and are a very good indicator of anomalies.

ADT provides in addition to the anomaly reporting inside DMon a more complete report for the trained models It creates files with the confusion metrics, training related parameters(time it took to train and validate, parameter optimization information etc.) and some visualizations of the trained model. In figure 18 we see the best performing decision tree structure learned.

In the case of the Wikistat toy application we used Storm topology related metrics. As opposed to the POSIDONIA use case data we do not have any categorical information, reencoding the data is not necessary. For these experiments we used a dataset with 60218 data points with a total of 54 features. This is a much larger data set both in event count and feature number. As opposed to the POSIDONIA data set this data was labeled using the *dmon-gen* tool.

We have run the same battery of experiments as before. Table 6 shows the overall performance and computational time of all supervised methods. As before we see that the performance of each resulting model is much better after the parameter search. We can also see that it took much longer for the parameter search to complete, this is in large part due to the size of the data set.

An interesting observation can be made from the feature importance metrics. Table 7 shows those

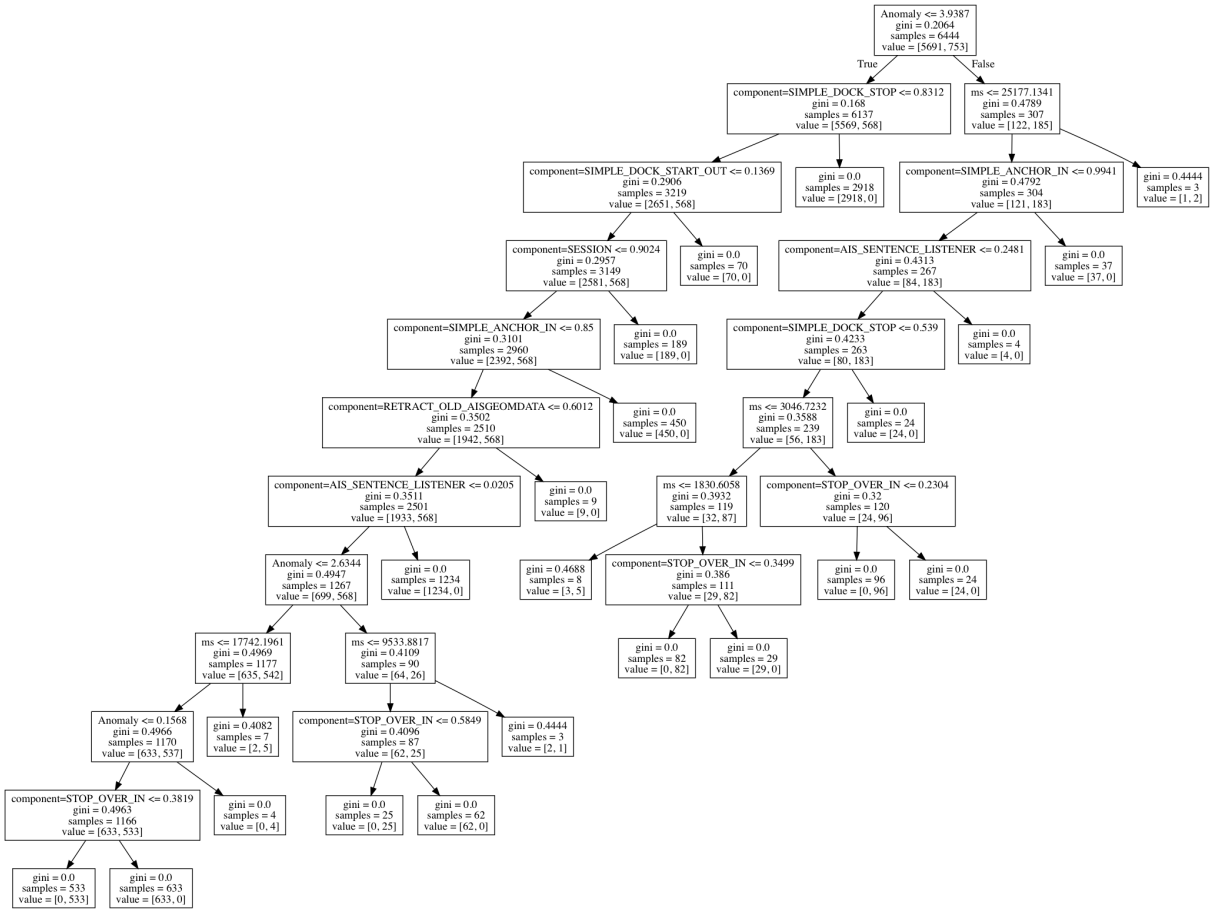

Figure 18: Decision Tree Model for CEP

Method	BScore	BTime	ParamSearch	CV Mean	CV STD	CV Time	FScore	FTime
RF	0.62	1.5985	1118.2022	93.61	0.27	24.8098	0.968	6.3571
DT	0.52	1.9713	100.0836	91.63	0.74	0.4723	0.9539	0.0663
AD	0.51	10.5293	1629.4891	89	21.15	32.0148	0.949	5.3124
NN	0.43	20.5565	4256.4321	93.1	0.94	49.2121	0.9522	7.2346

Table 6: Experimental runs for Wikistat Storm

Storm Metrics	AdaBoost	DecisionTree	Random Forest
bolts_0_acked	0	0.261248981	0.07353724
bolts_0_capacity	0	0	0.013144028
bolts_0_emitted	0	0.042070332	0.091775431
bolts_0_executed	0	1.91E-07	0.074560628
bolts_0_processLatency	0	0.004960664	0.008871831
bolts_0_transferred	0.05	0.013922383	0.062518511
bolts_1_acked	0.1	0	0.088776415
bolts_1_capacity	0.45	0	0.012296491
bolts_1_emitted	0.05	0.353170154	0.093166881
bolts_1_executed	0	0.136724359	0.097394063
bolts_1_executeLatency	0.05	0	0.03206146
spouts_0_emitted	0	0.023710512	0.059667492
spouts_0_transferred	0	0.033190521	0.079343435
topologyStats_10m_emitted	0.25	0	0.009834129
topologyStats_10m_transferred	0	9.64E-05	0.010155682
topologyStats_1d_emitted	0	0	0.006489571
topologyStats_1d_transferred	0	0	0.006286803
topologyStats_3h_emitted	0	0	0.003205677
topologyStats_3h_transferred	0	2.35E-05	0.003470178
topologyStats_all_emitted	0	0	0.086787035
topologyStats_all_transferred	0.05	0.130881988	0.086657019

Table 7: Feature importance for different methods Wikistat

Metric	CEP	Storm
Labeled anomalies	1447	6624
Detected anomalies	999	4516
False Positives	58	217
Good Anomalies	941	4299
Percentage labeled	22.4	11
Percentage detected	15.5	7.5
Accuracy	93.4	95.2

Table 8: Isolation Forest performance on labeled data set

features which have some degree of impact on the predictive performance of the learned models. Unlike the POSIDONIA data set where only two features can define an anomaly in this case we have a much wider set of possibilities which can give context to any anomaly.

The last set of validation experiments where done for Isolation Forest unsupervised method. Because we have already labeled data we can run the unsupervised method and see if it identifies the correct anomalies. Of course Isolation Forest is not able to distinguish between different types of anomalies it can mark events as normal or anomalous however, this is enough to test the ratio of false positives to true positives.

Table 8 shows the performance of Isolation Forest on both labeled dataset. Although at first glance one might conclude that it has poor performance a closer inspection shows that in fact the number of false positives is under 6% of the total identified for both data sets. It is true that not all anomalies have been found but this is of little practical significance as it is more important for the algorithm to detect true anomalies then to identify false ones. Keeping in mind that this methods precision increases with the amount of data it has to work with we are confident that this method is extremely useful for detecting anomalies during the initial stages of DIA development. In the latter stages of DIA development more data will be available and most likely examples of anomalous behavior These can be used for the training

of supervised predictive models which in turn can yield a much better predictive performance.

4.2 Regression based Anomaly Detection

In year 3, the regression based anomaly detection method (RBAD) was detailed and validated in the previous deliverable *D4.3* [9]. As discussed therein, the purpose of the RBAD tool is to detect presence of performance anomalies by accept a set of input performance measurs, train regression models describing application behaviour for the given metric(s) of interest, compare the data 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 of the application.

Since then we have integrated this method into the ADT. Because the initial prototype of RBAD was based on Matlab, we had to adapt it in order to be released with the Matlab compiler runtime for Java.

In the latest version of ADT, we start RBAD as a suprocess which is controlled from the main control thread. Some functionalities such as data querying and anomaly reporting are done directly by ADT, while in the previous version RBAD was guiding the user through the interactive analysis. RBAD is therefore responsible only for instantiating the regression model parameters and generating the anomaly predictions. Currently the data from DMon is dumped into the data folder from ADT in CSV format. Once the file is detected by a specialized process within ADT it will load the CSV file into memory using *cStringIO* module. After RBAD finishes its processing, it saves the output in JSON format which is the sent into DMon to be indexed by ADT. There were some challenges in integrating the Java Matlab runtime into ADT.

Summarizing, the main changes for the regression-based anomaly detection tool in year 3 are:

- It is now a method in the ADT, not an independent tool.
- It can be invoked by setting Method:regression in the Detect section of the ADT configuration file (please see below for more information).
- The tool reads all necessary input parameters from the ADT configuration file.
- The method now works with predictive regression models stored in the PMML format.
- The tool performs basic root-cause analysis by reporting which predictors (if any) contributed to the significant change in a metric of interest (improvement or degradation).
- The report the tool generates will be available to view in the D-Mon.

The integration between ADT and RBAD has been successfully tested with small to medium size data set up to 100 MB in size.

4.3 Trace Checking tool

Trace-checking tool was initially set up to be validated with ProDevelop use case, as DiceTraCT and ADT are brought together by similar capabilities of runtime data analysis. However, through a careful analysis of the scenario involving Posidonia system, we concluded that trace-checking could not be directly applied to this use case. The log traces of the application include, in fact, only those events that are related to the ships (like “DOCKING ship A on position X”) operating in the ports monitored by the system. All these events, however, have no direct mapping with the concepts available in the DICE profile and that can be applied for modeling a data-intensive applications like those implemented with the main technologies supported by DICE. In other words, in the example of event “DOCKING ship A on position X” the notion of “docking”, ship and position do not have a counterpart in the DPIM/DTSM/DDSM diagrams used to model the DIAs, being all these information application dependent. Therefore, the use of trace-checking for the log analysis of the Posidonia scenario would have been a customization of a log analysis technique for a specific application, lacking of a connection with the overall modeling environment and, in particular, with the verification tool D-verT that, together with DiceTraCT, constitute a compound toolkit supporting both the modeling and the iterative design refinement.

In conclusion, validation of trace-checking is therefore postponed and it will be realized in collaboration with the DICE industrial partner ATC on the Storm use case.

Nonetheless, the validation of DICE-TraCT has been achieved by running the trace-checking analysis on a benchmark application called *ExclamationTopology*, that was already used to test the solution in the previous deliverable D4.3. ExclamationTopology is a standard and very intuitive Storm application that is distributed in the Storm framework and that simply consists of three node: one spout emitting strings of characters, called "word", and two bolts "exclaim1" and "exclaim2" that elaborate the messages of "word" and "exclaim1", respectively, by simply adding a suffix with some symbols "!". To validate the functionality of DICE-TraCT, the topology was deployed in a local cluster running three Storm workers and one master node coordinating the topology. Beside the Storm topology an instance of the monitoring platform was executed to provide the monitoring functionalities that are required to carry out the trace-checking analysis. The trace-checking service has been deployed as a local service at 127.0.0.1 listening the port 5050.

The topology was first defined in the design panel of the DICE IDE with the UML classes and the DICE stereotypes `<<StormSpout>>` and `<<StormBolt>>` by means of the same procedure that a user follows to carry out verification with D-VerT. Then, the running Storm topology was registered in DMon through the web interface and DICE-TraCT was launched from the DICE IDE through the "Run configuration" window that is shown in Figure 19. A monitoring session was activated by clicking the button ''Activate" in the DICE-TraCT window and, afterwards, loading the Xml descriptor of the topology previously designed allowed the DICE-TraCT client to visit the structure of the application and show the nodes that can be analyzed. The value of parameters "sigma" and "emit rate" of the nodes "exclaim1" and ''word", respectively, were selected and then the trace-checking analysis was activated.

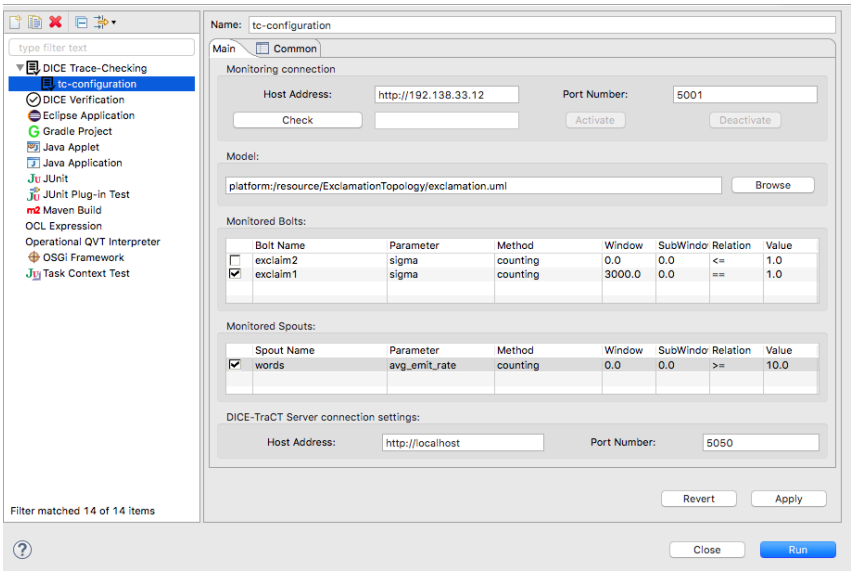

Figure 19: Launch configuration for a trace-checking analysis on two topology nodes

5 Conclusions

5.1 Summary

This deliverable presented the final versions of ADT as well as DICE-TraCT. The goal of the final version (M30) of these tools is 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 MS5 “DICE Integrated framework Second release Currently ADT has been tested on all DICE supported Big data technologies as well as on of the use case DIAs. DICE-TraCT has been completely integrated in the DICE framework and has been tested with a benchmark Storm application.

Furthermore we have finalized the 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 (M30) this integration is fully functional, data sets can be created, anomalies are transmitted to DMon and finally predictive models are saved and versioned.

5.2 Further work

At this stage the ADT and DICE-TraCT tools are fully operational an are an integral part of the DICE solution. Further work on these tools would be to extend their functionality past the one needed in DICE, to other problem domains. For example in the case of ADT problem domains such as IoT, Cloud and HPC can be easily included into the tool by extending some of its functionality (the connector component). Also, the integration of novel anomaly detection techniques can also be done by adding them to the ADT engine component. A little effort will be earmarked for the validation of DICE-TraCT with ATC use case in the next months.

References

[1] 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.

[2] Giuseppe Bombara, Cristian-Ioan Vasile, Francisco Penedo, Hirotoshi Yasuoka, and Calin Belta. A decision tree approach to data classification using signal temporal logic. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*, HSCC '16, pages 1–10, New York, NY, USA, 2016. ACM.

[3] Mihaela Elena Breaban and Henri Luchian. Pso aided k-means clustering: Introducing connectivity in k-means. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, pages 1227–1234, New York, NY, USA, 2011. ACM.

[4] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001.

[5] Giuliano Casale, Pooyan Jamshidi, Tatiana Ustinova, Gabriel Iuhasz, Matej Arta, Tadej Borovak, and Matic Pajni. Dice delivery tools initial version.

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

[7] Yu-Shu Chen and Yi-Ming Chen. Combining incremental hidden markov model and adaboost algorithm for anomaly intrusion detection. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, CSI-KDD '09, pages 3–9, New York, NY, USA, 2009. ACM.

[8] Martin Ester, Hans-Peter Kriegel, Jrg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. pages 226–231. AAAI Press, 1996.

[9] Tatiana Ustinova Marcello Bersani Gabriel Iuhasz, Ioan Dragan. Quality anomaly detection and trace checking tools - initial version. *DICE Deliverable D*, 4.3, 2016.

[10] 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.

[11] Youssef Ridene George Giotis, Christophe Joubert. D6.1 dice demonstrators implementation plan.

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

[13] R. G. Goss and G. S. Nitschke. Automated pattern identification and classification: Anomaly detection case study. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '17, pages 59–60, New York, NY, USA, 2017. ACM.

[14] 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.

[15] Michele Guerriero, Damian Andrew Tamburri, Youssef Ridene, Francesco Marconi, Marcello M. Bersani, and Matej Artac. Towards devops for privacy-by-design in data-intensive applications: A research roadmap. In *Companion Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, ICPE 2017, L'Aquila, Italy, April 22-26, 2017*, pages 139–144, 2017.

[16] 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.

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

[18] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation-based anomaly detection. *ACM Trans. Knowl. Discov. Data*, 6(1):3:1–3:39, March 2012.

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

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

[21] 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.

[22] Daniel Pop, Gabriel Iuhasz, Ciprian Craciun, and Silviu Panica. Support services for applications execution in multi-clouds environments. In *Autonomic Computing (ICAC), 2016 IEEE International Conference on*, pages 343–348. IEEE, 2016.

[23] 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.

[24] Ofer Schwartz, Sharon Gannot, and Emanuël A. P. Habets. An expectation-maximization algorithm for multimicrophone speech dereverberation and noise reduction with coherence matrix estimation. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 24(9):1491–1506, September 2016.

[25] Rinku Sen, Manojit Chattopadhyay, and Nilanjan Sen. An efficient approach to develop an intrusion detection system based on multi layer backpropagation neural network algorithm: Ids using bpnn algorithm. In *Proceedings of the 2015 ACM SIGMIS Conference on Computers and People Research*, SIGMIS-CPR '15, pages 105–108, New York, NY, USA, 2015. ACM.

[26] Manoj Kumar Sharma, Debdoot Sheet, and Prabir Kumar Biswas. Abnormality detecting deep belief network. In *Proceedings of the International Conference on Advances in Information Communication Technology & Computing*, AICTC '16, pages 11:1–11:6, New York, NY, USA, 2016. ACM.

[27] Xiuyao Song, Mingxi Wu, Christopher Jermaine, and Sanjay Ranka. Conditional anomaly detection. *IEEE Trans. on Knowl. and Data Eng.*, 19(5):631–645, May 2007.

