Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

DICE delivery tools – Final version

Deliverable 5.3

Deliverable:	D5.3
Title:	DICE delivery tools - Final version
Editor(s):	Matej Artač (XLAB)
Contributor(s):	Giuliano Casale (IMP), Derek Ho Law (IMP, Tatiana Ustinova (IMP), Gabriel Iuhasz (IeAT), Matej Artač (XLAB), Tadej Borovšak (XLAB), Damian Andrew Tamburri (PMI)
Reviewers:	José Merseguer (ZAR), Ismael Torres (PRO)
Type (R/P/DEC):	Demonstrator
Version:	1.0
Date:	31-July-2017
Status:	Final
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2017, DICE consortium – All rights reserved

DICE partners

ATC:	Athens Technology Centre
FLEXI:	Flexiant Limited
IEAT:	Institutul E Austria Timisoara
IMP:	Imperial College of Science, Technology & Medicine
NETF:	Netfective Technology SA
PMI:	Politecnico di Milano
PRO:	Prodevelop SL
XLAB:	XLAB razvoj programske opreme in svetovanje d.o.o.
ZAR:	Universidad de Zaragoza

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

Executive summary

This document contains the report of the final release of the DICE delivery tools: DICE Deployment Tool, DICE Continuous Integration and DICE Configuration Optimisation. The purpose of these tools in the DICE methodology is to create a runtime of a DIA described in a DDSM / TOSCA blueprint, provide scheduled or on-commit execution of complex automated tasks on top of the DIA, and offer recommendations for the optimal configuration for the DIA's deployment. The report focuses on new functionalities and validations that we carried out between months 24 and 30 of the project.

In the DICE Deployment Tool, the new functionalities include strengthening of security of the tools themselves as they are installed to use HTTPS for communication, added support for MongoDB deployment, and ability of the new blueprints to freely migrate between platforms (OpenStack, Amazon EC2, …). The use of HTTPS will slightly slow down the installation of the tools, but significantly improve the security of the tools' operation. Applying security to the DIAs by design is the next step, where our DICE technology library creates and enables user accounts in MongoDB automatically. We provide validation of the final version of the tool through a custom city traffic data use case, showing a 4x speed-up when using DICE tools. In a logical sense, deployment is also related with growingly popular container technology. We argue that the technology is complementary to the DICE building blocks and can be viewed at the same level as user's custom components.

The main functionalities of the DICE Continuous Integration was already complete, but we extended the DICE Jenkins plug-in to support improved pipeline project types in Jenkins. This enables using a Continuous-Integration-as-Code approach, increasing flexibility of DIA automated deployment and testing. We took advantage of the approach to speed up ATC Topic Detector's Quality Testing process by only deploying the Storm cluster once, then reusing it in consecutive builds.

For the DICE Configuration Optimisation, we provide substantial usability upgrades by bringing all the controls into the Eclipse IDE: the new IDE plug-in now provides a guided creation of experiment configuration. Additional integration of IDE with Jenkins Continuous Integration enables an experiment execution that is driven from the IDE. On request, the outcome of the experiment can then be displayed in the IDE.

Glossary

DDSM	DICE Deployment Specific Model
DICE	Data-Intensive Cloud Applications with iterative quality enhancements
FCO	Flexiant Cloud Orchestrator
TOSCA	Topology and Orchestration Specification for Cloud Applications
IDE	Integrated Development Environment
CI	Continuous Integration
BO4CO	Bayesian Optimisation for Configuration Optimisation
DIA	Data Intensive Application
HDFS	Hadoop File System
GUI	Graphical User Interface
VCS	Version Control System
JSON	JavaScript Object Notation
YAML	YAML Ain't Markup Language[1]
SWT	Standard Widget Toolkit (a Java library for user interfaces)
CSRF	Cross-Site Request Forgery

[1] http://yaml.org/

Table of contents

Executive summary 3
Glossary 4
Table of contents 5
List of Figures 6
List of Tables 7
1 Introduction 8
1.1 What is new in Y3 8
1.1.1 DICE Deployment Tool 8
1.1.2 DICE Continuous Integration 9
1.1.3 DICE Configuration Optimisation 9
2 Requirements 11
3 Tools 13
3.1 DICE Deployment Tool 13
3.1.1 Main components 13
3.1.2 DICE Deployment Service 13
3.1.2.1 TOSCA technology library 16
3.1.2.2 Security by design 17
3.1.2.3 Multi-cloud support through unified approach 18
3.1.3 Supported technologies 19
3.1.4 Container technology perspective 22
3.1.5 Evaluation and validation: a city traffic use case 23
3.2 DICE Continuous Integration 25
3.3 Configuration Optimisation 29
3.3.1 Overview of integrated solution 29
3.3.2 CO Eclipse plugin 31
3.3.3 CO Jenkins integration 33
4 Conclusion 35
4.1 DICE Requirement compliance 36
References 38

List of Figures

Figure 1: Deployment Diagram of the DICE Deployment Tool 13
Figure 2: Architecture of the city traffic use case 23
Figure 3: Architecture of the Configuration Optimisation solution 30
Figure 4: Interface for building an experiment configuration with the Eclipse plug-in 31
Figure 5: Hierarchy of components used in the Parameter Selection tab 32
Figure 6: Interface for providing configuration for external services used by the Configuration Optimisation 33

List of Tables

Table 1: Summary of services and their required method of creating certificates 15
Table 2: Break down of the times required for applying DICE Deployment Tool to the city traffic use case 24
Table 3: Level of compliance of the initial version of the DICE delivery tools with the initial set of requirements 36

1 Introduction

The DICE delivery tools are able to consume blueprints that are based on the OASIS TOSCA standard, and produce in a fully automated and unattended way a running application. They unburden the users from the tedium of configuring individual hosts one by one, and speed up the deployment and testing workflow considerably. This enables a DevOps [1] workflow, which according to DICE Methodology [8] starts at modelling and offline validation, then transitions using the deployment tool into the realm of the DIA's runtime.

The tools are then capable of varying the configuration parameters of the deployment, while at the same time measuring performance of the DIA. This yields a recommendation on the best configuration for the specific DIA, automatically helping with an increased quality of the resulting application.

Both components of the tools are now available from Eclipse IDE, making them comfortably close to the developer's regular workflow. This is useful for hands-on experimentation. The DevOps approach, on the other hand, is enabled through the Jenkins Continuous Integration.

This report accompanies the final release of the tools that include **DICE Deployment Tool**, **DICE Continuous Integration**, and **DICE Configuration Optimisation**. The reported work is a part of the DICE project's WP5 on deployment and delivery. Specifically, it includes results from T5.1 Deployment plan execution and T5.2 Continuous integration. This report is the final one in a series starting with D5.1 DICE delivery tools – Initial version [2], where we introduced the tools, their initial architecture and functionalities. The D5.2 DICE delivery tools – Intermediate version [3] was then an update, reporting on GUI improvements, technologies added to the support, and acceleration of assessment of optimal configuration.

In the rest of this section we summarize the changes and improvements since Y2 (M24). The Section 2 summarizes the requirements for the DICE delivery tools, extracted from the D1.2 [4]. In the Section 3 we present the details of each tool's new features, also presenting their usage and validation results. Finally, in Section 4 we present the conclusions, including the analysis of the requirements fulfilled by the work.

1.1 What is new in Y3

For the final release of the tools in the M30, we have addressed all the issues that remained open in M24. Here, the major focus was on maximizing the stability and usability of the tools. In this section, we briefly highlight results by each tool.

1.1.1 DICE Deployment Tool

At the end of Y2, the DICE Deployment Tool has already provided to the users a great improvement in the develop-deploy-test cycle [3]. It demonstrated that spinning up complex clusters of Big Data technologies to support DIAs can be a quick, reliable and automatic. For our **Deployment Service**, we have since improved the following aspects:

- The installation process and the clients (i.e., command line tool, IDE plug-in) now support and indeed mandate communication across encrypted (HTTPS) channels. This is an important step towards security by design, which needs to start from the supporting services.

- New integration with the DICE Monitoring Tool allows that the DICE Deployment Service automatically registers the whole application with the monitoring tool, thus storing in the Monitoring Warehouse essential information about each application's deployment. This information is then valuable to all the downstream tools such as Enhancement Tools.
- An increased fault tolerance improves reliability of the tool, making the DevOps process work without human interventions.
- A better overall usability and user experience.

With the service side being stable and feature complete, we were able to focus on contributing with major improvements to the **DICE TOSCA technology library**:

- A unification across the underlying platforms now enables a true abstracting in the TOSCA, making the blueprints that exploit the DICE TOSCA technology library functional for any supported platform.
- Added support for Amazon's EC2 extends the possibility to deploy DIAs using DICE to a wider variety of public cloud platforms.
- Added support for MongoDB.
- All of the technologies supported now by the DICE Monitoring Tool get automatically connected for monitoring during deployment. After the undeployment, they are also capable of deregistering from the monitoring.

These changes enable a wider spectrum of possible uses of the DICE tools, both in terms of the technologies used in the DIA as well as by enabling new deployment targets.

1.1.2 DICE Continuous Integration

The most notable new aspect of the DICE Continuous Integration is that we have migrated from classic free-style Jenkins projects to newer and more flexible pipeline projects. To support them, we have updated the DICE Jenkins plug-in to its version 0.3.0. This means that the DevOps aspect now includes handling the Continuous Integration aspects of the project as well.

1.1.3 DICE Configuration Optimisation

The Configuration Optimisation (CO) tool provides a software mechanism to explore alternative configurations for a DIA and identify the optimal one with respect to a given performance metric (e.g., throughput, response time, ...). The intermediate version of this tool, presented in deliverable *D5.2 – DICE delivery tools – Intermediate version* [3], is based on an algorithm, called BO4CO, which drives the search for an optimal configuration using a technique known as Bayesian Optimisation, which can cope with variability in the measurements and allow to customize the optimal trade-off between exploitation of existing measurements and exploration of new configurations. A large-scale validation has been performed for Storm-based DIA and Apache Cassandra.

In Y3 we have consolidated the CO tool as follows. First, we have resolved minor bugs in the instantiation of the tool on testbeds with arbitrary combinations of Big Data technologies and in the initial generation of the Latin hypercube design.

Next, we observed that the configuration parameters of big data frameworks fall into these four categories: 1) Integer – the parameter may take any integer value between a set of lower and upper bounds. 2) Percentage – the parameter may take any value between 0 and 1. 3) Boolean – the parameter value may be true or false. 4) Categorical – the parameter may take any value from a list of Strings options. In the original release, BO4CO tool only supported numerical parameter values,

i.e. Integer and Percentage parameters. In order to have "fully automatic" configuration optimisation, all parameters should be supported. Hence, we extended the BO4CO tool to support Categorical and Boolean types.

Third, we have developed an Eclipse plugin to instantiate runs of BO4CO directly from the DICE IDE, mediated by communication with the Jenkins instance in the DICE continuous integration toolchain. In this deliverable, we primarily focus on discussing the implementation of this plugin.

With these updates, the DICE Configuration Optimisation has now become better accessible to the users, because they can include setting up and managing the Configuration Optimization experiments directly in their IDEs. The support for additional configuration categories further extends the possible technologies addressed by the tool.

2 Requirements

With the Deliverable D1.4 [5], we have provided a Companion document with all the updates presented the requirement analysis for the DICE project. This section includes summaries of the prominent requirements in their state at the end of M30.

ID	R5.4.5
Title	Deployment tools transparency
Priority	Should have
Description:	The DEPLOYMENT_TOOLS SHOULD NOT require from ADMINISTRATOR to take part in any individual deployment.

ID	R5.4.6
Title	Deployment plans extendability
Priority	Could have
Description:	The DEPLOYMENT_TOOLS MAY be extended by the ADMINISTRATOR with other building blocks not in the core set.

ID	R5.7.1
Title	Data loading hook
Priority	Should have
Description:	DEPLOYMENT_TOOLS SHOULD provide a well-defined way to accept the initial bulk data that they can load.

ID	R5.4.9
Title	Deployment plans portability
Priority	Should have
Description:	The DEPLOYMENT_TOOLS SHOULD be able to support more than one vendor's IaaS.

ID	R5.27.1
Title	Brute-force approach for CONFIGURATION_OPTIMISATION deployment
Priority	Should have
Description:	CONFIGURATION_OPTIMISATION SHOULD apply intelligent ML methods in order to enable a sequential decision making approach that selects a promising configuration setting at each iteration. CONFIGURATION_OPTIMISATION should find the best possible configuration at the end within the

ID	R5.27.6
Title	CONFIGURATION_OPTIMISATION experiment runs
Priority	Must have
Description:	CONFIGURATION_OPTIMISATION MUST be able to derive the experiment by running the application under test with specific configuration setting by contacting DEPLOYMENT_TOOL. CONFIGURATION_OPTIMISATION MUST be able to retrieve the monitoring data regarding the experiments by contacting MONITORING_PLATFORM.

ID	R5.27.7
Title	Configuration optimisation of the system under test over different versions
Priority	Should have
Description:	CONFIGURATION_OPTIMISATION SHOULD be able to utilize the performance data that have been collected regarding previous versions of the system under test in the delivery pipeline.

ID	R5.27.8
Title	Configuration Optimisation's input and output
Priority	Must have
Description:	CONFIGURATION_OPTIMISATION MUST be able to receive a TOSCA blueprint, which describes the application under test including any initial configuration. It MUST return a TOSCA blueprint updated with optimal parameters, or a stand-alone configuration file.

ID	R5.43
Title	Practices and patterns for security and privacy
Priority	Must have
Description:	The DEPLOYMENT_TOOLS MUST enable applying practices and patterns to ensure that the deployed application is reasonably secure and protecting privacy.

3 Tools

3.1 DICE Deployment Tool

3.1.1 Main components

At the end of M30, the DICE deployment tool is a collection of the following components:

- DICE deployment service version 0.3.4
- DICE TOSCA technology library version 0.7.0
- DICE Chef Cookbooks version 0.1.12 (in use by the DICE TOSCA technology library)
- Cloudify 3.4.2 (provided by the GigaSpaces).

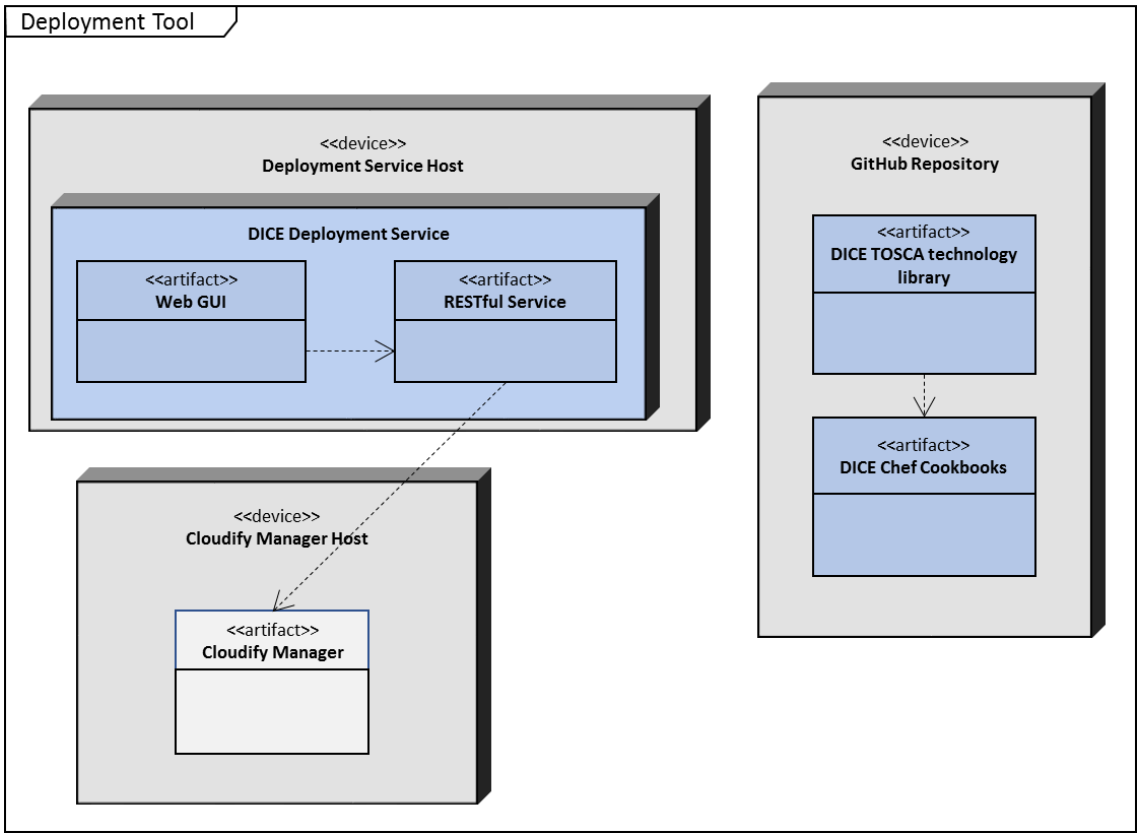

Figure 1: Deployment Diagram of the DICE Deployment Tool

As the deployment diagram on Figure 1 shows, the architecture of the DICE Deployment Tool remains unchanged since M24. The entities marked in blue are from DICE, while the others are from third parties.

3.1.2 DICE Deployment Service

The DICE Deployment Service is an abstraction layer, which we created on top of a Cloud orchestration layer with one purpose in mind: to simplify the inclusion of the continuous deployments into a DevOps workflow. We have achieved this by providing an interface with simple actions: **deploy** a blueprint, **redeploy/replace** the previous blueprint with a new one, and **undeploy** a previously deployed blueprint. Through applying these actions onto a specific virtual deployment container, the tools for DevOps such as the Continuous Integration may rely on a clear separation of applications and their roles. For example, a specific virtual deployment container may be dedicated to main DIA's branch testing, another one to an experimental branch, a third one might

be manually deployed (i.e., by manually submitting a blueprint) and used for manual acceptance tests, etc.

Of course, the DICE Deployment Service offers a wider variety of API calls, which allow for managing a list of the virtual deployment containers, and setting up testbed-specific input values. We consider these calls to be for administration purposes. As already presented in [3], delegation of platform parameters into the central service essentially enables that the blueprints themselves can be platform-agnostic and thus highly portable across private and public cloud providers. Input parameters related to the target platform such as cloud account credentials and identifiers of cloud images are still needed by the blueprint, but their values will be dynamically supplied by the DICE Deployment Service. The service is now also less strict when validating the submitted blueprint's input list, such that it does not require any of the inputs that already have default values set. As a result of these improvements, the usability of the deployment tool has already increased considerably.

Integration with DICE Monitoring Service. In M30, the DICE Deployment Service is aware of the functionality of the DICE Monitoring Service [9], which allows an application to be registered at the beginning of its runtime. The integration is controlled by two aspects. First, the Administrator needs to assign input parameters to the service, which provide locations of specific DICE Monitoring Service's access points. Listing 1 shows the list of the needed parameters. These will cue the DICE Deployment Service to submit an application (deployment) ID to the DICE Monitoring Service, which stores it in its history. Optionally, the DICE Deployment Service can supply the deployment metadata (such as the name of the application, the version of the DIA being deployed, etc.) with this registration. The action of registration thus becomes searchable by the metadata, helping various anomaly detection and enhancement tools to easier browse or discover past DIA runtime instances.

```
dmon_address: Main dmon address (eg. 10.50.51.4:5001).
logstash_graphite_address: Graphite address (eg. 10.50.51.4:5002).
logstash_lumberjack_address: Lumberjack address (eg. 10.50.51:5003).
logstash_lumberjack_crt: Lumberjack certificate.
logstash_udp_address: Logstash udp address (eg. 10.50.51.4:25826).
```

Listing 1: Inputs used from DICE Deployment Service to enable registration of applications and services with the DICE Monitoring Service

The second aspect comes from the DICE TOSCA technology library. Many of the node types declare property `monitoring`, which lets individual node template to indicate whether it needs to be monitored. Listing 2 shows an example usage of the property. For such nodes, the library defines configuration steps for setting up the DICE Monitoring tool's local agent, configuring the services to enable logging, and notifying DICE Monitoring Service about the monitored nodes.

```
node_templates:

  master:
    type: dice.components.spark.Master
    properties:
      monitoring:
        enabled: true
    # ...
```

Listing 2: Example of how to enable in a blueprint that a Spark master node is automatically monitored

Security of the services. It is important to be aware that with exchange of parameters and blueprints during deployment, sensitive data get exchanged between clients and services. It is therefore essential to **secure communication channels** using encryption. Up until Y3, our approach to this issue has been relaxed in the sense that all exchanges were in clear-text HTTP. By working with prototypes and in a non-mission-critical environment, the risk or consequences of potential leaking of information was acceptable. In exchange, we were able to focus on functionality, while postponing the effort of establishing secure connections to the later time.

With the M29 release 0.3.4 of the DICE Deployment Service, we have upgraded the **service deployment blueprints** to **setting up the secure (HTTPS) connections by default**. As a consequence, no eavesdropping is now possible for any of the API calls, be it on the private or a public network. Additionally, the set up of the services creates user accounts, unsolicited visitors from accessing the deployments or administration interfaces.

This change has a small impact on the ease of use of the services. Now, the users need to supply their user credentials, and configure their clients with the public certificate of the service. For convenience (which should not considerably diminish the security benefits), all of our clients provide the means to locally store these credentials: the command line interface creates a hidden configuration file, the IDE plug-in uses Eclipse's secure data store, and the web user interface uses a standard token (cookie) based authentication once the user provides their credentials.

On the other hand, the added security does require more effort of the Administrators when setting up and configuring the services. In particular, the Administrators need to create and maintain a **Certificate Authority** to be able to create public certificates and private keys. There should be one such key pair per a service to be set up. This process adds to up to 30 minutes of the Administrator's time when first setting up the Cloudify orchestrator and the DICE Deployment Service. We estimate that each certificate needed to be created manually then takes additional 15 minutes of time, including the time for creating the private key and certificate signing request, signing the public key, and configuring the service's blueprint with the certificate and private key. The process uses standard open source OpenSSL tool, and we have provided quick instructions[2]. Table 1 summarizes the services and the methods of creating service certificates.

On the clients' side, the user needs to install the public certificate so that the client will trust the service. This incurs a one-time cost of 5 minutes to the user. The installation steps are the same regardless of the certificate creation method used for the server.

Table 1: Summary of services and their required method of creating certificates

Service	Method of certificate creation
Cloudify Manager	Manual
DICE Deployment Service	Automatic

In our understanding, the cost of creating the certificates is relatively low comparing to the gained benefits of sensitive data protection. Also, the process is by nature only needed once (or, when

[2] https://github.com/dice-project/DICE-Deployment-Service/blob/master/doc/certificates.md

renewing the certificates, infrequent), and therefore negligible in comparison to the high frequency of accesses and use of these services in a day-to-day DevOps methodology.

As shown in Table 1, we built into the DICE Deployment Service's bootstrap process an automatic creation of the service's certificates. This is a convenience measure, which saves Administrator's time at a cost of transferring all control of the certificate creation process to the orchestrator.

3.1.2.1 TOSCA technology library

While the DICE Deployment Service provides the means for deployment, the content of the DICE project comes from the TOSCA technology library. The goal of the library is to provide every DICE-supported building block's configuration and deployment capabilities and wrap them into easy to use elements in arbitrary DIA topologies.

As already reported [7][3], there are three components making up our TOSCA technology library. First, the **TOSCA YAML definitions** provide component-specific node types and relationships. We built them by inheriting node types from the ones provided by Cloudify. For example, a blueprint for deploying a stand-alone MongoDB cluster involves the following node types:

- `dice.hosts.ubuntu.Medium`: represents a compute host of a medium size.
- `dice.firewall_rules.mongo.Common`: a node type for defining a networking security group or firewall, such that only the ports needed for MongoDB to communicate are accessible, and this includes peer engine services and any clients.
- `dice.components.mongo.Server`: a component containing all the MongoDB-related modules that comprise a stand-alone instance of the MongoDB engine.
- `dice.components.mongo.DB`: represents a database in a MongoDB engine.
- `dice.components.mongo.User`: a user in a MongoDB cluster.

Many of the node templates need to be in a relationship with another node template. We do this using the following relationship types:

- `dice.relationships.ProtectedBy`: the source of this relationship is a compute host, and the target is a `dice.firewall_rules` node template defining the secure group or a firewall.
- `dice.relationships.ContainedIn`: may connect a service-related node template to a compute host, or a database to a database engine such as MongoDB.
- `dice.relationships.mongo.HasRightsToUse`: enables permission of the source user node template with the target database node template.

The TOSCA YAML definitions provide a foundation for declarative representation of the DIA at the same level as the DDSM [7]. We have made sure that these concepts can be mapped directly from the DICE metamodel into a TOSCA blueprint, because in this way a DICE UML model created in the IDE becomes fully actionable. This means that suitable orchestrators will accurately and consistently turn the DIA's model into the DIA's runtime.

The second aspect of the library are **Chef cookbooks** for each building block. Cookbooks are composed of recipes, and their purpose is to implement one or more components' lifecycle steps (e.g., install, start, configure, stop). While granularity of Chef is much smaller than that of the cloud orchestrator, focussing on files and services of a compute host, the structure of the Chef cookbook recipes also has a strong declarative nature. Ultimately, the steps are implemented as imperative

components, but they are embedded into higher-level structures. This property lets us segment recipes into stages of the orchestration workflow, granting us a good level of flexibility. As an exercise of reusability, we were able to use our Chef recipes to implement a working Ubuntu JuJu charm[3].

The final elements of the technology library are **plug-ins and extensions** for the Cloudify orchestration engine. Many of the workflow steps and relationship implementations in the TOSCA library require tailored approaches and careful handling of installation sequences. We implemented these as Python scripts and registered them to be triggered at certain times of the workflow. For example, some of the clustered services require that all of the peers are installed, configured individually and running first. Then they need to enter a common cluster one by one. Such workflow needs a special logic implemented as a Python script to access and update the state (context) of a blueprint's deployment.

This context becomes available to the Chef recipes (which themselves are able to manipulate), thus enabling insight into orchestrator's wider picture to the configuration management. While the shape and format of this context is Cloudify-specific, it was trivial for us to recreate it in the aforementioned Ubuntu JuJu charm, proving a possibility to migrate to other orchestration engines (e.g., ARIA TOSCA) with low effort.

3.1.2.2 Security by design

When surveying the installation and configuration instructions of various Big Data technologies, we found **a common theme** that **security aspects** of the components are often **secondary** or are even more **neglected**. Protecting a new or an existing cluster requires many seemingly arcane and cumbersome steps that strictly speaking are not necessary during development and test phases. However, it is quite likely that the Ops keep the development-level set-up also when going to production, putting systems and data at risk. Many of these systems end up on **publicly available network interfaces**, where a specialized search tool such as Shodan[4] is able to index them en masse. Just recently, media[5] reported of 5.12 Petabytes of Hadoop Distributed File System datasets being uncovered in such a way. Earlier in the year, a high number of MongoDB and ElasticSearch datasets were stolen, held ransom or deleted[6].

Systems that are **secure by design** start already at the **modelling phase** with security and, possibly, privacy of the DIAs in mind. In D2.2 [6], we have already covered the modelling aspects, where the architect can express **security policies** through an interplay of: resources (what needs to be protected), actions (what can be done to/with the resources), actors and roles (who is doing the actions against the resources) and permissions (specific actors and roles allowed to do specific actions against specific resources).

Transferring these capabilities into the context of the deployment and configuration is beyond the scope of the DICE project. The long-term goal would be two-fold: wherever possible, the deployed DIA has to be deployed from the ground up in such a way that it prevents any unauthorized users

[3] https://github.com/xlab-si/DICE-Juju-Charms
[4] https://www.shodan.io/
[5] http://thehackernews.com/2017/06/secure-hadoop-cluster.html
[6] http://thehackernews.com/2017/01/mongodb-database-security.html

from gaining access to the restricted data or functionality of the DIA. Where that is not possible, it should at least be possible to detect anomalous usage from the logs or other monitoring approaches.

It is important to note that policies expressed in the DIA's models need to be **applied at various levels of the application's deployment**. At the level of a DIA's supporting cluster, we can typically create **database engine-level users** and assign **permissions to datasets** such as tables or keyspaces. However, this is a relatively coarse-grained access control, which cannot enforce policies that work at the level of individual records in a dataset. This, in turn, is a **responsibility of the user's custom application**.

By M30, we chose MongoDB as the technology to demonstrate concepts of the security by design. As noted earlier, this NoSQL database engine is often set up using default configurations and thus unprotected from any actors within the engine's network. As a minimum security measure, all our MongoDB deployments will have **created an administrator account** with a **strong password** that is randomly **generated** during each deployment. Additional users may be defined in the TOSCA blueprint, and the orchestrator will generate passwords for them that are random and strong as well. Any **clients** needing to access the MongoDB datasets then **must know these credentials** (obtainable either through orchestrator's dynamic attributes or manually by copying them from deployment outputs). **Anonymous access** is therefore **disabled**.

The work related with security by design has uncovered an important aspect of orchestration: generation and exchange of secrets. This includes pieces of information such as passwords, API keys, service or host private keys and any other sensitive items. They are essential for a successful deployment orchestration, but the challenge is to keep them away from the eyes of the users of the deployment services. In Cloudify, this functionality is subject to commercial and paid licenses, while in the community version, users need to be careful with who can use the services and for what purposes. As a part of future work, we plan to include third party solutions such as HashiCorp Vault[7] to solve this problem.

3.1.2.3 Multi-cloud support through unified approach

The version 0.7.0 of the DICE TOSCA technology library[8] released in M29 has enabled truly multi-platform blueprints. Originally, the particular platform plug-ins in Cloudify exposed node types that were specific to that platform not only in the name of the node type, but also in structure of properties. DICE unifies all the supported computation and networking concepts, encapsulated in the following base node types:

- `dice.firewall_rules.Base`
- `dice.VirtualIP`
- `dice.hosts.centos.Base`
- `dice.hosts.ubuntu.Base`

In terms of the node type names, this is only a slight improvement over the M24 release. In particular, we introduced an explicit selection of the compute host's OS distribution: Ubuntu or CentOS. Any existing blueprints would therefore simply need to replace occurrences of

[7] https://www.vaultproject.io/
[8] https://github.com/dice-project/DICE-Deployment-Cloudify/releases/tag/0.7.0

"`dice.hosts.`" with "`dice.hosts.ubuntu.`" to become functional with the version 0.7.0 of the library.

To support this change, DICE needed to also implement its own plug-in (i.e. Python code components) for platform operations. Using platform native libraries (please see justification below), the selection of the target platform is now a matter of providing an appropriate `platform` property name:

- `fco`: the instance will be provisioned in Flexiant Cloud Orchestrator
- `openstack`: supports OpenStack version Icehouse or newer (tested also on Mitaka and Newton)
- `aws`: deploys in Amazon's EC2
- If the property is not explicitly set in the node template, the DICE Deployment Service will pick the default platform, usually in its own testbed environment.

The library therefore provides for interesting and highly flexible deployments. In the common scenario, also used by all the use cases in DICE, the DIAs' blueprints will not prescribe any platform. This will make the blueprint deployable without any change into any of the supported platforms.

Additionally, we can envision DIAs spanning multiple types of clouds, e.g. a combination of public and private clouds. In this scenario, some or all of the relevant node templates will have the `platform` property set to the relevant platform type. It is then a matter of supplying the proper inputs containing each platform's access credentials, image IDs and other parameters for such deployment to work. This scenario, however, is out of scope of the DICE project and is thus subject to potential future work.

A note on the choice of the third party libraries used to support a platform: we were careful to evaluate whether it is better to use the target platform's native client library, or an abstraction such as Apache Libcloud9. With an abstracted library, our expectation was that it would make extending support for new platforms simpler. In practice it turned out that support for OpenStack in Libcloud was not fully available and thus the approach was not usable for us. Additionally, we still needed to study and understand intricacies, parameters and structures particular to each platform, which diminished any benefits of having a single common library.

This experience has shown us from another perspective that only through abstraction at a higher level is it possible to achieve a multi-cloud operation. Cloud orchestration engines are such an abstraction, and while they internally need to handle specific interfaces of the supported platforms, the blueprint writers and users do not have to.

3.1.3 Supported technologies

This section briefly presents the technologies supported by the DICE Technology Library in its M29 version 0.7.0.

Apache Zookeeper. This component is a support service for many of the other Big Data services, and might also be used by some of the DIAs. It is a distributed service, which helps coordinate other distributed services and keep their local or global configurations. The distributed services are instances of the `dice.components.zookeeper.Server` node type. Clusters of Zookeeper servers need to be connected using the `dice.relationships.zookeeper.MemberOfQuorum`

[9] https://libcloud.apache.org/

relationship to a common node template of type `dice.components.zookeeper.Quorum`. All the other node templates that require connection to the Zookeeper need to use `dice.relationships.zookeeper.MemberOfQuorum` relationship to connect to the quorum (and *not* to the Zookeeper itself).

Apache Storm. A Big Data service for stream processing of data streams and batches. Each Storm cluster needs to contain at least one instance of node template belonging to type `dice.components.storm.Nimbus`, which serves both as the nimbus (i.e., supervisory) node and the web interface to the Storm cluster. The actual work is performed by instances of the node templates of type `dice.components.storm.Worker`. Both the Nimbus and the Worker nodes require connection `dice.relationships.storm.ConnectedToZookeeperQuorum` to the Zookeeper's quorum node template. Each Worker also needs to connect to its Nimbus using `dice.relationships.storm.ConnectedToNimbus`.

User applications in Storm are Storm topologies. These are represented by node templates of type `dice.components.storm.Topology`. They are in a relationship with the Nimbus node by `dice.relationships.storm.SubmittedBy`. The custom applications will therefore be started by the Cloudify orchestrator, to be then run independently in the Storm.

Apache Spark. This technology is suitable for executing batches or micro-batches (to simulate stream processing) in a distributed manner on top of Big Data. A Spark topology is composed of Master nodes instantiated from node templates of type `dice.components.spark.Master`, and of Worker nodes from `dice.components.spark.Worker`. Workers need to be connected to the Master node template using `dice.relationships.spark.ConnectedToMaster`.

User applications are presented as `dice.components.spark.Application` node templates, related to the Master through `dice.relationships.spark.SubmittedBy`.

Apache Kafka. Messaging bus such as the one provided by Kafka is an important element in DIAs, offering message passing and queueing, publishing messages and content to multiple listeners, balancing load between resources, etc. In a DIA blueprint, we include them as node templates of type `dice.components.kafka.Broker`. Each node template has to be connected to Zookeeper Quorum using `dice.relationships.zookeeper.ConnectedToZookeeperQuorum`.

Hadoop File System (HDFS). A cluster for the object data store is composed of name nodes and data nodes. To define the name nodes, add a node template of type `dice.components.hadoop.NameNode`. The data nodes are defined by a node template of type `dice.components.hadoop.DataNode`, which needs to be in the `dice.relationships.hadoop.ConnectedToNameNode` relationship with the name node's node template.

Hadoop YARN. The popular distributed application task runtime management needs two components in its cluster: resource managers and node managers. The node type `dice.components.hadoop.ResourceManager` defines a node template of a resource manager. The blueprint then needs to contain a node template of type `dice.components.hadoop.NodeManager`, which is in a relationship

`dice.relationships.hadoop.ConnectedToResourceManager` with the resource manager node.

Apache Cassandra. Cassandra is a distributed highly available and fault resistant NoSQL database engine. A cluster of Cassandra service composes a seed node (of multiple instances) represented by node template of type `dice.components.cassandra.Seed`, and worker nodes of type `dice.components.cassandra.Worker`. A worker needs to be in a relationship `dice.relationships.cassandra.ConnectedToSeed` with a seed node.

MongoDB. Another NoSQL database engine based entirely on the concept of document storage. A MongoDB cluster can take many forms, including a stand-alone server, a replicated cluster, or a shared cluster.

For the **stand-alone mode**, the blueprint simply needs to provide a node template of type `dice.components.mongo.Server`.

In the **replicated cluster mode**, instead of a simple server, the blueprint needs to contain a replicated server represented by a node template inherited from the node type `dice.components.mongo.ReplicaServer`. This node template then belongs in a MongoDB group represented as `dice.components.mongo.Group`, which connects to the replica server using `dice.relationships.mongo.ComposedOf`.

The **sharded cluster mode** needs to contain configuration servers, represented by the `dice.components.mongo.ConfigServer` node type's template instances. A group of configuration service replicas is represented by `dice.components.mongo.Group`, and the relationship between a replica group and a configuration server is `dice.relationships.mongo.ComposedOf`. The actual data is stored in shards represented by multiple node templates of type `dice.components.mongo.ShardServer`, where each shard server node template must be in its own relationship `dice.relationships.mongo.ComposedOf` with its own node template for the `dice.components.mongo.Group`. To connect the whole structure, the cluster also needs a router node, represented as a `dice.components.mongo.Router` node template type. The router works with configuration data stored in the configuration server, so its node template connects using `dice.relationships.mongo.ConfigurationStoredIn` to that of the configuration server's node template. The router then routes requests to the shard server replicas, therefore for each target replica group there has to be a relationship of type `dice.relationships.mongo.RoutesTo`.

The DICE TOSCA technology library also provides definition for MongoDB databases and users. A node template for a database uses node type `dice.components.mongo.DB`. It needs to be connected using `dice.relationships.ContainedIn` to the appropriate database server or server replica: `dice.components.mongo.Server` in the stand-alone mode, `dice.components.mongo.ReplicaServer` in a replicated cluster mode, and `dice.components.mongo.ShardServer` in a sharded cluster mode. To define a user, add a node template of type `dice.components.mongo.User` and use `dice.relationships.ContainedIn` to apply it to the database server or server replica. To set

a user's permission to use a database, add to the user's node template a relationship of type `dice.relationships.mongo.HasRightsToUse` with the target database.

Custom script. For the DIA components that do not conform to any of the above building blocks, it is possible to use a simple custom script building block. `dice.components.misc.ScriptRunner` node type provides properties to set a language of the script (either `python` or `bash`), to list command line arguments for the script, and enumerate any additional resources (files, scripts, libraries, etc.) that need to be in the file system of the host running the script. These files need to be present in the blueprint bundle when submitting the blueprint for deployment. The node template is limited in that the library doesn't have any relationships defined to connect a script node with other node templates. However, TOSCA enables using built-in functions to obtain inputs, properties and dynamic runtime attributes to supply information on other node template instances to the script.

3.1.4 Container technology perspective

Containers are a special form of computer virtualization, enabled by operating system containers, where multiple applications run in isolation from any other containers or processes within the same user space. Container technologies are gaining a growing interest and support in the industry. The advantages include a high level of portability, the fact that the containers are lightweight and are also quickly deployed. They can be used to package single services, composite parts of an application such as micro services, or whole applications.

In terms of DICE deployment and configuration, we view individual containers as a special type of user's custom application or component. These were not in focus of the DICE project, which instead gave the majority of its attention to Big Data technologies. In principle, containers could provide an alternative approach to configuring and deploying the building blocks supported by DICE, such as Spark, Storm or Cassandra. While we consider this an interesting possibility, we believe that our existing approach already works well and has enough flexibility for serious use. The container approach might provide some deployment speed, but it also increases the complexity of the DICE tools implementation, and likely reduces the component's performance.

That said, DICE Delivery Tools offer a good basis for possible future support of container technologies. For a container to be able to run, a TOSCA blueprint (as well as the deployment document) would need to specify: 1) a template for a host such as a virtual machine, 2) a capability of this host to receive containers of a particular brand (Docker, Linux kernel containers – LXC, etc.) either through a special type of the host template or an associated node type, and 3) a template for the actual container to be deployed. The DICE TOSCA technology library would need to contain the needed building blocks. In this schema, the orchestrator such as Cloudify in the back-end of the DICE Deployment Service treats containers equally as any other component, and it also takes responsibility for the containers' lifecycle.

Many container technologies are now complemented by orchestrators specialized to handle containers. Kubernetes and Docker Swarm are just two of such solutions. Typically, they expose an interface that makes a cluster of container hosts appear as a single virtual container host. For DICE Deployment Service, a container orchestrator then appears like a persistent implicit (i.e., not explicitly represented in a TOSCA blueprint) platform for hosting containers. This concept is not very different from how we represent data centres and cloud providers in DICE TOSCA.

Specifically, the host node templates do not have to explicitly specify the infrastructure that they are guests to. Instead, this is handled by Cloudify as an orchestrator, and the DICE Deployment Service, which supplies credentials and configuration values through TOSCA inputs.

This last scenario is also interesting because we could implement a TOSCA blueprint to deploy a container orchestrator, then update the inputs in the DICE Deployment Service to have the subsequent blueprints use this orchestrator as an extended platform.

3.1.5 Evaluation and validation: a city traffic use case

We validated the DICE Deployment Tool in an industrial use case developed internally at XLAB. The use case collects various publicly available data on traffic conditions in the Slovenian capital city Ljubljana. This use case was developed in collaboration with the TIMON[10] H2020 project.

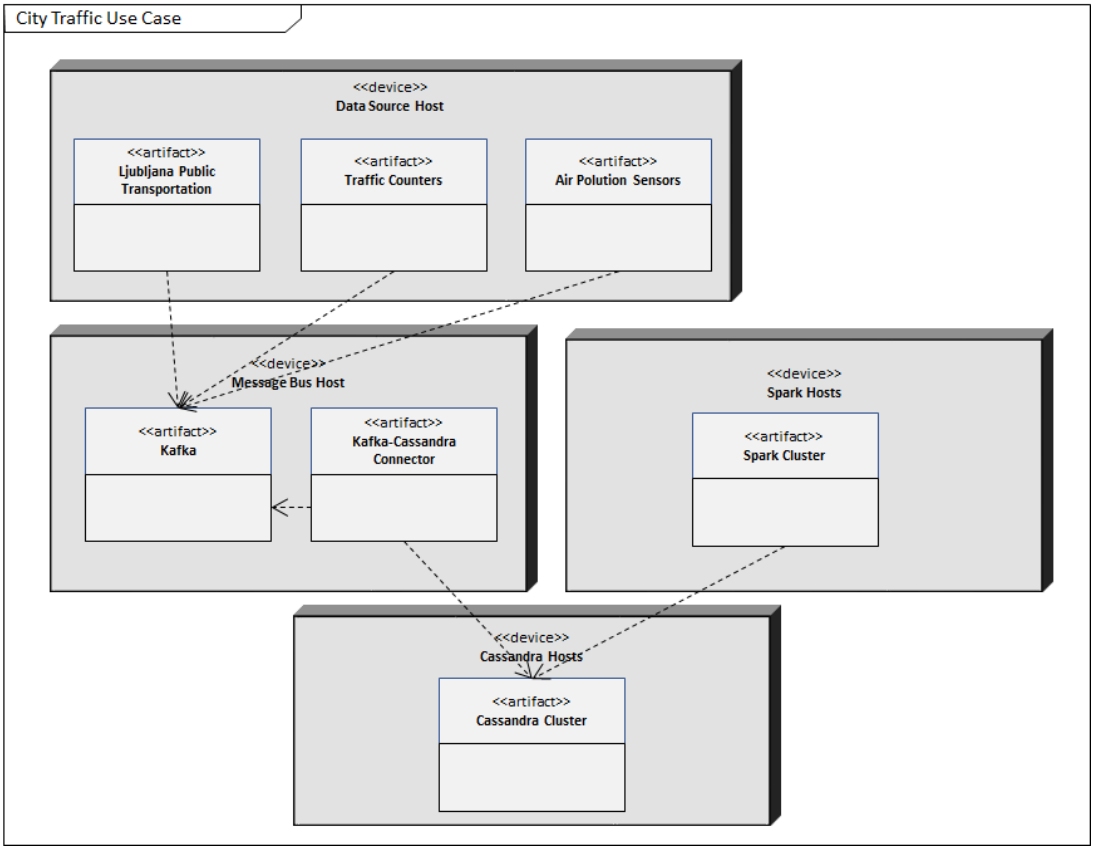

Figure 2: Architecture of the city traffic use case

The city traffic DIA architecture is shown on Figure 2. As shown, it is composed mostly of services that are supported by DICE and are thus available in the TOSCA technology library: Kafka, Spark and Cassandra. For additional components we developed custom Chef cookbook recipes. The goal of the validation was to show that DICE tools provide a speed-up in the development and deployment process of the use case DIA.

In the first phase, we created a **skeleton blueprint** with all the nodes that are already supported in DICE, and we added to the blueprint additional host nodes with no services assigned to them. The resulting cluster would be suitable for manual installation of the use case's custom components.

[10] https://www.timon-project.eu/

Next, we extended the skeleton blueprint with the node templates that would represent the custom and unsupported components of the city traffic DIA. In particular, we needed to accommodate for installation of:

- the data collector component, which periodically queries the data sources and sends results to Kafka topics, custom made for the use case,
- Stream Reactor[11], a connector for Kafka that receives data from a pre-configured topic and writes the contents into a Cassandra keyspace, a third-party solution with configuration specific to the use case.

As the descriptions of these components suggest, each component has an external dependency to Kafka or Cassandra, the access points of which need to be written into an appropriate configuration file during the deployment. In the blueprint, we therefore created **custom relationships** to express such dependencies. In the deployment workflow, these relationships also make sure that the values needed in configuration of either of the custom components are available at the time the orchestrator triggers configuration of the components. Our relationships were derived from `dice.relationships.Needs` and reused interface implementation available in the TOSCA technology library for storing the target node host's FQDN into a runtime attribute specified by the relationship.

The DICE TOSCA technology library does not yet provide any means for describing service nodes with custom Chef recipe runlists. For creating **custom node templates**, we therefore needed to create a custom node type, which we derived from `dice.chef.SoftwareComponent` and provided the ability to assign arbitrary recipes. This ensured that Cloudify would execute our Chef recipes for the use case's custom components. Next, we **adapted our Chef recipes** slightly to use Cloudify's context and its dynamic attributes, which enabled them to use Kafka and Cassandra access point addresses in the configuration process.

The work we have described was carried out by a senior engineer. The Table 2 shows estimates of the time they needed to create and fully test the city traffic DIA TOSCA blueprint. The entire process of creating deployable application (excluding the implementation time of the DIA's functionality itself) took around 40 hours. Of that time, only **10 hours** were applied to the use of DICE tools.

In comparison, if we didn't have DICE, but wanted to create deployable TOSCA blueprints, we would have spent at least 38 hours on the same activities as described above except for the skeleton blueprint creation. We would then need to create and implement blueprints that would deploy Kafka, Spark and Cassandra, which would take a senior engineer at least 3 weeks (120 hours) to implement. This means that with DICE we have achieved at minimum an almost **4x speed-up** of the use case deployment implementation.

Table 2: Break down of the times required for applying DICE Deployment Tool to the city traffic use case

Task	Time required [engineering hours]
Skeleton blueprint creation	2 h
Custom relationships implementation	1 h
Custom node templates implementation	5 h
Adaptation of Chef recipes	2 h

[11] https://github.com/datamountaineer/stream-reactor

Total	**10 h**

3.2 DICE Continuous Integration

In DICE, the aim of the Continuous Integration task is to provide execution of automated tools and processes (from DICE or from third parties) in a manner that is itself automated: it happens when a developer pushes a new version to the VCS, or at specific times of day or week. By M24, we have implemented a Jenkins plug-in to help visualize the progress the developers make in terms of quality of the DIA. By M30, we have refined the method of setting up the Jenkins jobs, and enabled speed-ups of the Jenkins projects executions. In the following paragraphs, we will present the updated approach on the example of the ATC's NewsAssets' Topic Detector application.

The agile methods encourage the use of Everything as Code. The DDSM, Chef cookbooks and TOSCA blueprints all are great examples of Infrastructure as Code. In the context of Continuous Integration, Jenkins also provides the means to present test jobs in a DSL that is based on the Apache Groovy[12] language. The developers need to include a file named `Jenkinsfile` in their project, and the Administrator only needs to add a project in Jenkins of type **pipeline**, such that it reads the VCS and executes the selected `Jenkinsfile`. The developers are then free to build and modify the test definition, evolving it organically just as the DIA project evolves, without having to involve the Administrator any further.

From the methodological point of view, we now also want to separate Continuous Integration jobs into stages by the frequency of their required execution and the time they take in each of their cycles. As the analysis in the previous deliverable [3] has shown, a single cycle of deployment and teardown could take up to 10 minutes on a reasonably occupied testbed. From the point of view of deployment, we consider this to be fast. However, in the context of application testing, we strive to bring the Continuous Integration job execution times to as low a number as possible. In looking for reserves and through noting the typical DevOps team's behaviour, we have established that modelling and updating the cluster takes up a small number of commits. In particular, for the Posidonia Operations, less than 5% of commits influence the models. Similarly, the Netfective Technologies teams practically never change the model once it is finalized, while the DIA's logic commits occur daily. To test these, it is normally acceptable to reuse the cluster deployed in the previous deployment runs.

Note: *the approach is only safe as long as the developers implement removal of all traces of the previous tests' runs, e.g., by making sure that any datasets created and populated during each job's run get fully purged before beginning of the next runs.*

The Jenkins pipeline definition's `Jenkinsfile` is composed of stages, which correspond to the various phases that the developer of an application wants to have Continuous Integration run. In Listing 3, we see a skeleton of such a `Jenkinsfile`. The skeleton represents a mandatory `pipeline` block, indicating that this file uses the declarative syntax. The pipeline stages will run on any of the available agents as declared with the `agent any` declaration. This means that the job will run on any of the currently available Jenkins nodes. The `stages` directive declares the stages to be executed during the main job's execution. They will be executed in sequence until one of them

[12] http://www.groovy-lang.org/

either fails or until all of them succeed. The `post` block tells about what should happen after the stages are finished.

```
pipeline {
    agent any

    stages {

        stage('build-test') {
        }

        stage('deploy') {
        }

        stage('quality-testing') {
        }
    }

    post {
        always {
        }
        success {
        }
    }
}
```

Listing 3: Jenkinsfile skeleton for multi-stage testing and deployment

We could easily populate this skeleton for any type of DIA. We will illustrate this on a Storm application, which we based on the ATC's Topic Detector [11] use case.

The first stage is named build-test. Its goal is to compile the application and test it, as shown in Listing 4. The outcome is a jar file containing the user's compiled application, stored in the project's workspace on Jenkins (thus it will be available for subsequent stages). The stage also runs unit tests, so if anything is wrong in the code, the execution stops here.

```
stage('build-test') {
    steps {
        sh 'mvn clean assembly:assembly test'
    }
}
```

Listing 4: Declaration of stage for building the DIA and running unit tests.

Next, we provide the details in the `deployment` block. As discussed before, we want to save time and execute it only if really needed. To this end, we add a pre-stage named `pre-deploy`, as shown on Listing 5: we note down a hash of the current blueprint bundle. For the edge case of the stage running on a fresh workspace, we also initialize a file containing the previous blueprint's hash. Then we read the two and note down if they are equal or not.

In the actual `deploy` stage, also shown in Listing 5, we use a `when` block to make sure the stage is run only if the blueprints have changed. If yes, we submit the blueprint bundle to the DICE Deployment Service and wait for the deployment to be finished. Please note that the environment variables for the DICE Deployment Service's configuration path `DDS_CONFIG`, and the UUID of the virtual deployment container to be used by this DIA `STORM_APP_CONTAINER` need to be assigned by the ADMINISTRATOR in the Jenkins configuration.

```
stage('pre-deploy') {
    steps {
        sh 'sha256sum blueprint.tar.gz > current-hash.txt'
        sh 'touch last-successful-hash.txt'
        script {
            env.CURRENT_HASH = readFile('current-hash.txt')
            env.LAST_HASH = readFile('last-successful-hash.txt')
            env.CHANGED = env.CURRENT_HASH != env.LAST_HASH
        }
    }
}

stage('deploy') {
    when {
        expression { return "${env.CHANGED}" == "true" }
    }
    steps {
        sh '''
            dice-deploy-cli deploy --config $DDS_CONFIG \
                $STORM_APP_CONTAINER \
                blueprint.tar.gz

            dice-deploy-cli wait-deploy --config $DDS_CONFIG \
                $STORM_APP_CONTAINER
        '''
    }
}
```

Listing 5: The stages for deploying the cluster needed by the DIA onto the testbed

If the previous stage succeeds or was skipped, we assume that the Big Data cluster for the DIA is available[13]. The Listing 6 then shows the steps needed to run the DIA in the Quality Testing [10] mode. Here, we obtain the address of the Nimbus service using a convenience wrapper around the DICE Deployment Service client tool, and we store the result in `STORM_NIMBUS_HOST` variable. The rest of the variables are pre-set and DIA specific: `TOPOLOGY_CLASSPATH` provides the full name of the Storm application's class to be executed, `TOPOLOGY_NAME` is the name to be used in Storm for the topology, and `METRICS_FILE_PATH` is the path in the workspace to contain an outcome of the quality testing runs. Only the `STORM_UI_URL` is dynamic and composed of the Nimbus host address obtained earlier.

The last command in the step executes the actual submission of the DIA into the cluster. Here we assume that the application is written such that it employs QT-LIB and performs the needed querying of the monitoring subsystem of the Storm. At the end, it also needs to return the metrics to be displayed at the Continuous Integration.

[13] Please note that in the presented example, the target Big Data cluster might be off-line even if the pre-deploy check assumes that it is available. We skipped this condition for brevity, but it can be easily addressed by adding a check using DICE Deployment Service's RESTful calls to confirm that the cluster is indeed deployed in the virtual deployment container.

```
stage('quality-testing') {
    steps {
        sh '''
            STORM_NIMBUS_HOST=$(dice-get-output.sh $STORM_APP_CONTAINER \
                    storm_nimbus_host "$DDS_CONFIG")

            TOPOLOGY_CLASSPATH="gr.iti.mklab.focused.crawler.QTTopicsDetector"
            TOPOLOGY_NAME="topic-detector"
            TOPOLOGY_JAR="target/focused-crawler-jar-with-dependencies.jar"
            STORM_UI_URL="http://$STORM_NIMBUS_HOST:8080"
            METRICS_FILE_PATH="output/result.json"


            bash submit-topology.sh \
                "$TOPOLOGY_JAR" \
                "$TOPOLOGY_NAME" \
                "$TOPOLOGY_CLASSPATH" \
                "$STORM_NIMBUS_HOST" \
                "$STORM_UI_URL" \
                "$DMON_URL" \
                "$METRICS_FILE_PATH"
        '''
    }
}
```

Listing 6: Definition of the quality testing stage block

The `post` block defines how to wrap up the job. As shown in Listing 7, we are interested in addressing two conditions: the `always` block executes regardless of any of the steps failing or succeeding. Here, we collect and archive the compiled `.jar` files, if they exist. By archiving the results, we ensure that they will be accessible using a permanent URL composed separately for each build. The `success` block executes only if all the stages succeed. Here we archive also the outcome of the Quality Testing, and we supply this same file as an input to the DICE Jenkins plug-in.

```
post {
    always {
        archive 'target/*.jar'
    }
    success {
        archive 'output/result.json'
        DICEQualityCheck(pathToResults: 'output/result.json')
    }
}
```

Listing 7: Definition of actions to happen after the build

A `Jenkinsfile` as shown here then needs to be committed into a VCS in the DIA's repository. Jenkins will execute it on schedule as it is obtained from the VCS. After a number of the job's executions, the project's dashboard in Jenkins will display a summary similar to Listing 8. The view contains two important components: on the top right, DICE Jenkins plug-in shows a chart of metrics for each build. The rest of the view is occupied by the table, where rows represent individual builds, and columns represent stages of the pipeline. Notice that the timing in the `deploy` column varies depending on whether a build involved a changed blueprint or not.

Listing 8: Sample execution history view in Jenkins

As shown, Jenkins displays timings of each stage. Like at M24, DICE plug-in shows a chart of the metrics in the past Quality Testing executions.

3.3 Configuration Optimisation

3.3.1 Overview of integrated solution

The implementation of the CO Eclipse Plugin consisted of two main parts:

1) Provide a development environment in the Eclipse IDE, where developers can explore configuration optimisation and the performance of their applications.

2) Fully integrate the development environment to trigger configuration optimisation on remote automation server and run tests on remote testbed.

The challenges addressed in Y3 for the first part focused on the Eclipse IDE were as follows:

- Allow selection of configuration parameters of corresponding Big Data technology for optimisation.
- Allow specification of parameter values, ranges and intervals to experiment upon – Extend BO4CO tool to support broader types of input: boolean, categorical, ranges and intervals.
- Allow configuration of experiment set-up, e.g.: test application to run, numbers of iterations and experiment time.

- Allow setting of connections to remote Jenkins server, remote testbed and monitoring services.

The challenges addressed in Y3 for the second part focused on the integrated toolchain were as follows:

- Integrate Eclipse and Jenkins for triggering parameterised builds with configuration files remotely.
- Integrate Jenkins automation server and MATLAB-based BO4CO tool to start experiment.
- Integrate BO4CO tool and remote Storm testbed to deploy tests with different configuration parameters and retrieve performance metrics.
- Integrate Eclipse and Jenkins to retrieve and display BO4CO configuration results.

A schematic view of the overall CO solution is shown below in Figure 3.

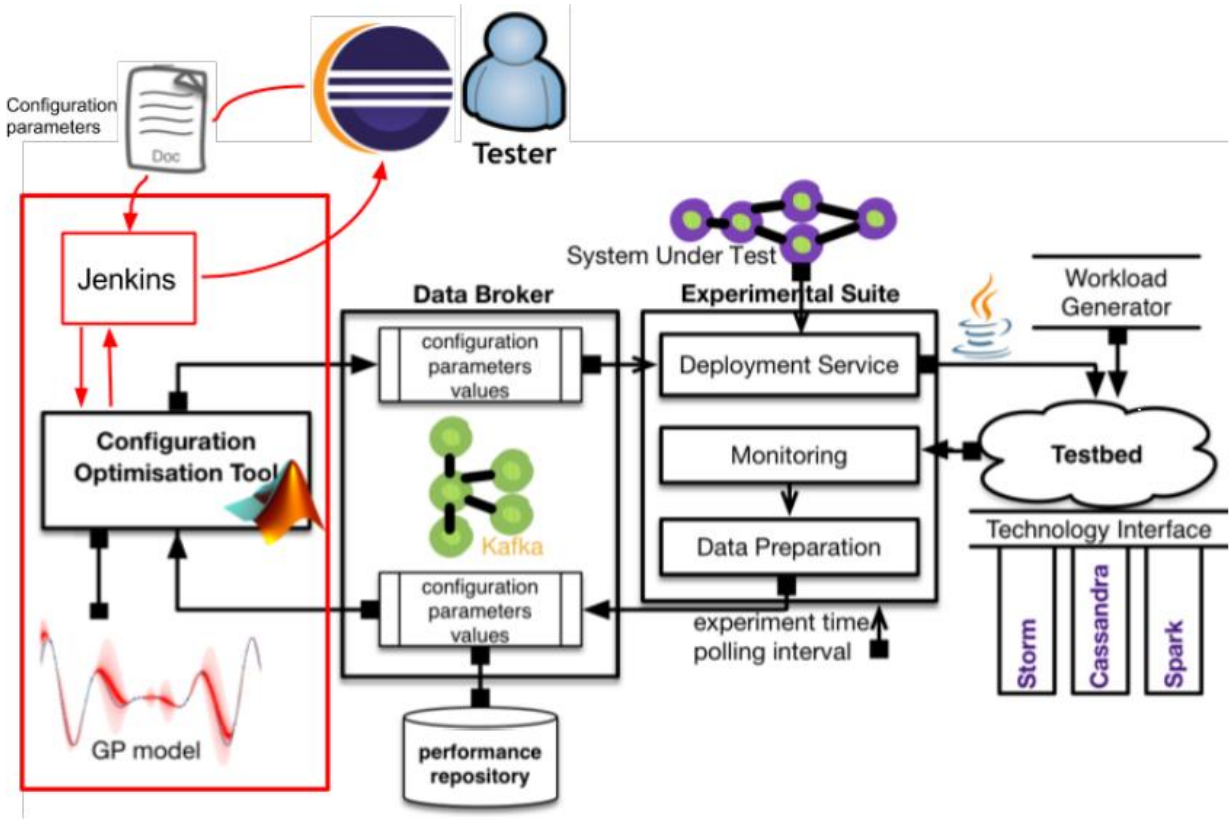

Figure 3: Architecture of the Configuration Optimisation solution

Initially, the developer selects the configuration parameters to optimise via the user-friendly interface on the Eclipse IDE. The experiment configuration file is generated according to the selections, and sent to the remote Jenkins CI server as it triggers the experiment to run. Jenkins executes the Configuration Optimisation tool, and monitors the status of the experiment. When the CO tool terminates, Jenkins retrieves the results and the optimised configuration.

The developer can access the results conveniently from the Eclipse IDE. The Big Data Auto-tuning Tool provides a fully integrated solution in the development environment to performance test and tune big data applications, in line with the DevOps principles of frequent testing in deployment environment.

In comparison to the existing approach to directly work with the Configuration Optimisation tool, the developer can now run optimisation and view results without leaving the IDE. The developer no longer manually creates an experiment configuration file for the tool. The GUI lists the configuration parameters available for selecting the corresponding big data framework, provides helpful descriptions to aid non-expert developers. It simplifies the process for the developer, without requiring that they understand the format of the CO tool's experiment configuration file. It also eliminates human error when creating the file and guarantees no parsing problem when it is executed.

3.3.2 CO Eclipse plugin

A screenshot of the CO Eclipse plugin is shown in Figure 4.

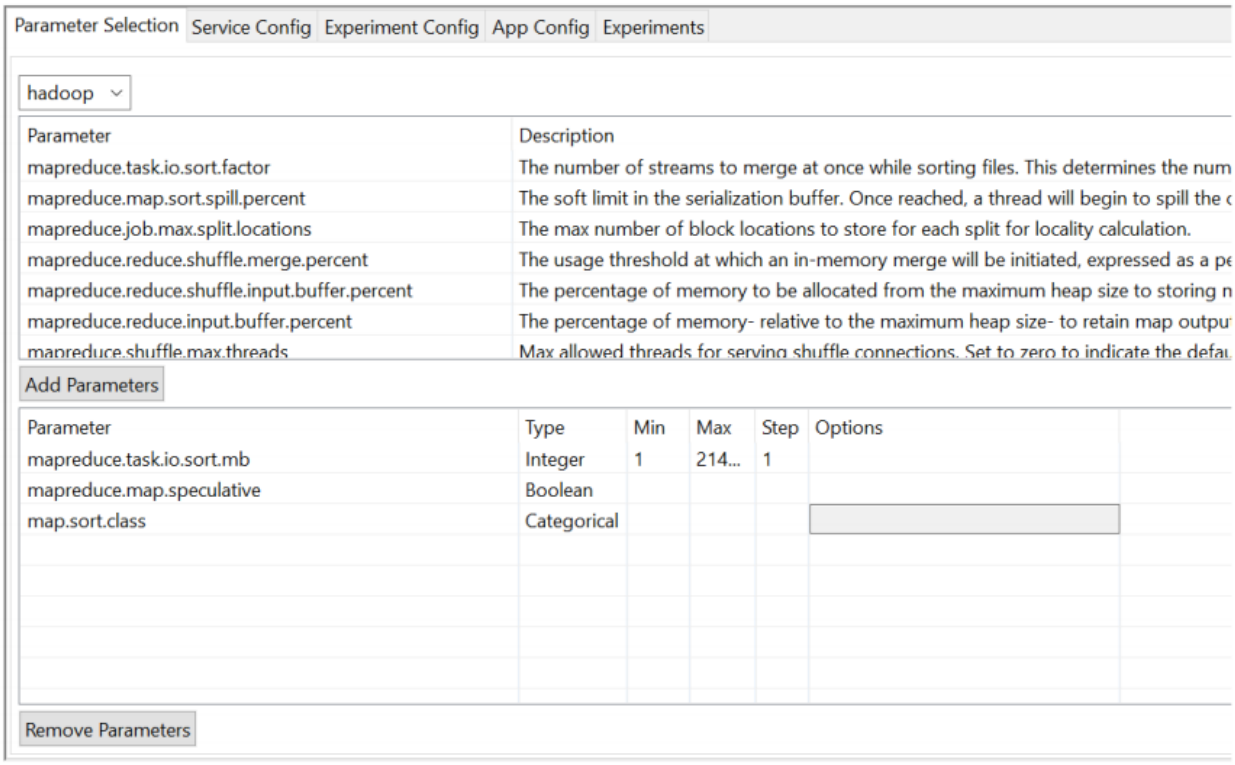

Figure 4: Interface for building an experiment configuration with the Eclipse plug-in

The interface consists of several top-level tabs. *Parameter Selection* provides a list of standard configuration parameters for DICE-supported technologies and allows the developer to specify an arbitrary numerical range for integer and percentage type parameters, and choose from the available options of categorical parameter.

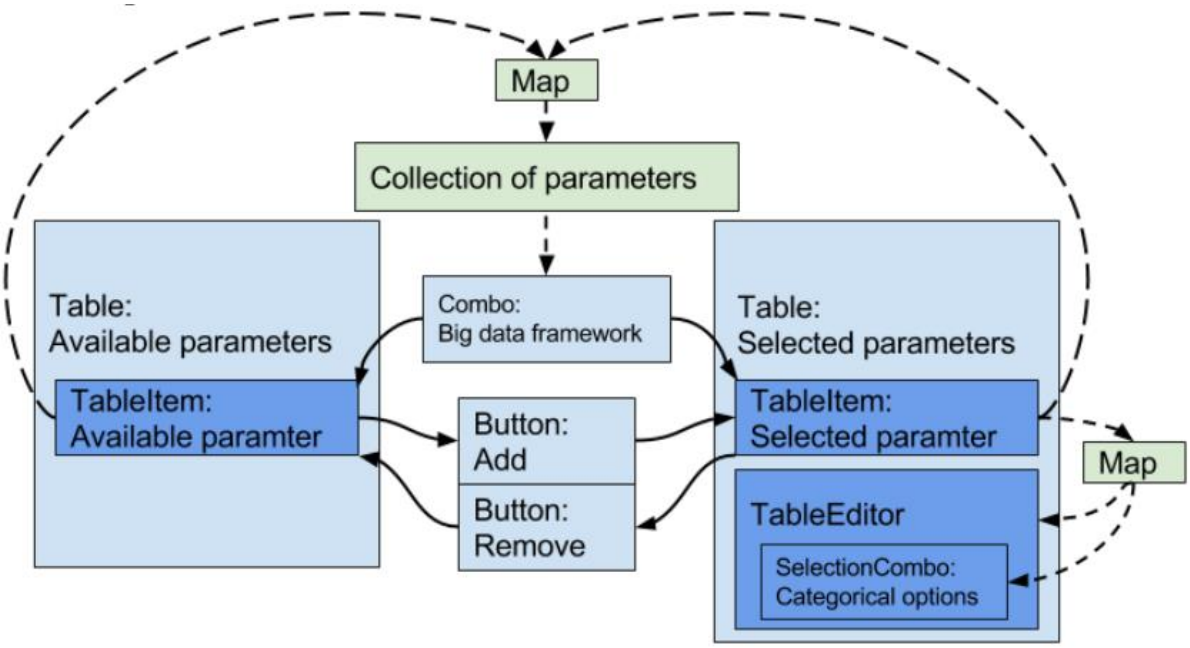

Figure 5: Hierarchy of components used in the Parameter Selection tab

Figure 5 shows how the different components in the *Parameter Selection* tab are connected:

- Blue components are Standard Widget Toolkit (SWT) widgets that appear on the user interface.
- Green components are underlying data collections and objects.
- Deeper shade of blue is used to highlight that multiple instance of the component may exist.
- A SWT widget is the parent of SWT widgets contained inside its box on the diagram.

When the plugin is launched, a collection of parameter objects is created from reading the *params.xml* file shipped with the plugin. The SWT Table of available parameters is created, along with the drop-down list (SWT Combo[14]) that controls which parameters should be displayed for the selected big data framework. The SWT Table of selected parameters is initially empty, along with SWT Buttons for "Add parameter" and "Remove parameter", which allow the user to include the parameters that BO4CO should optimize during the experimentation. The Services, Experiment, and Application configuration tabs are similar in that all of the user inputs are simple single text fields that specify some required information about the services that need to be tested, such as their URL and login credentials.

[14] http://www.eclipse.org/swt/widgets/

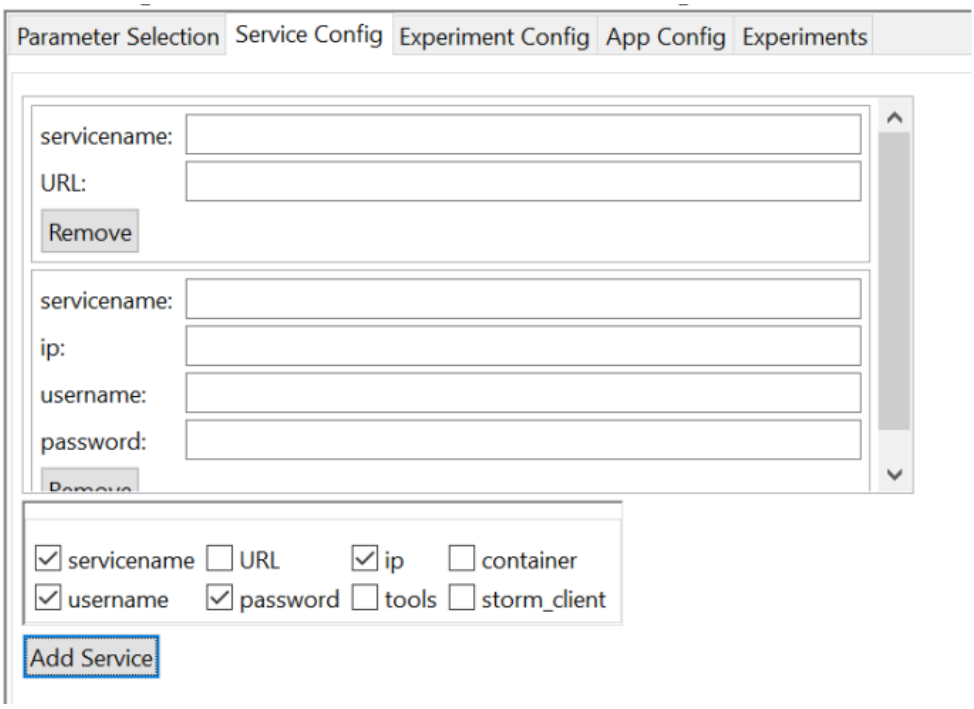

Figure 6: Interface for providing configuration for external services used by the Configuration Optimisation

3.3.3 CO Jenkins integration

The aim of this contribution is to provide integration between the Eclipse plugin described in the above section, and a remote Jenkins server containing the Configuration optimisation tool instance. The tool requires an experiment configuration file to execute, and hence there is a need to remotely trigger Jenkins' parameterised build with a file parameter. A Jenkins project has to be created, with remote triggering enabled. This opens the option to specify a token used for added security when remotely executing the build from the Jenkins API. The parameterised trigger option has to be selected, with a file parameter specified.

We used in this part an existing Eclipse plugin that is capable of running builds and monitoring the status of remote Jenkins server, called the Hudson/Jenkins Mylyn Builds Connector[15]. However, this plugin does not support triggering parameterised builds and does not support sending file (or any) parameters. The possibility of reusing code from the plugin was investigated, but it was found that the code required to trigger parameterised build significantly differed from the existing code that only supported simple builds. Therefore, the integration was built from scratch. Jenkins offers an API that opens functionality to remote access.

The API requires three security components to remotely trigger builds:

- User-defined project remote trigger token. The project token is a simple plain text token that is sent as one of the arguments in the HTTP request URL string.
- Credentials of an authorised user – username and password. Pre-emptive authentication is used to authenticate the user with provided username and password. The pre-emptive authentication for Apache HTTP client is done by implementing a `HTTPRequestInterceptor` that intercepts all HTTP requests and injects the authentication component.

[15] https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/examples/RandomTextWriter.html

- CSRF. There is no available implementation of the Cross-Site Request Forgery (CSRF) authentication for Jenkins and Apache HTTP client, possibly due to the fact that CSRF was only enforced recently for new installations of Jenkins 2.x upwards. The CSRF crumb can be obtained by requesting the Jenkins API's `CrumbIssuer`, and the response message is parsed to extract the crumb value. It is stored and used in subsequent requests. Remotely triggering parameterised builds is documented in the Jenkins remote access API.

Text based parameters such as strings and integers were simply appended onto the trigger HTML request URL as parameters. To retrieve the result from Jenkins Server to Eclipse plugin when the "Show results" button is pressed, an HTTP request is sent. Jenkins API provides a plaintext HTTP response that contains the entire console log output of the last successful build. The configuration optimisation results are extracted from this output.

4 Conclusion

In this report, we have presented the final release of the DICE deployment and configuration tools. We consider the tools to now be feature complete, at least in terms of the requirements gathered at the beginning of the project and adjusted during the project's progress. In fact, several of the features that we implemented were not planned initially, but we recognised their importance as we worked with our own tools, such as security by design, and Continuous Integration as Code. We have validated the individual tools internally as respective tool owners, and provided the results of the validation in this report.

The self-validation that we have conducted has shown a high value of the DICE Deployment Tool even for DIAs that include custom components or technologies not supported by the DICE TOSCA technology library. We have demonstrated that inclusion of the DICE building blocks is simple and quick, while connecting custom parts works well. Customization and extension of the DICE TOSCA technology, however, remains an advanced topic that needs to be done by the experts. The reason for this is not in the way the technology library is created, but much more due to the fact that automating any components takes effort and time. Nevertheless, some of the existing elements of the technology library can readily help speed up the customization work.

An important aspect of usability of a tool is also that it produces results that can be applied in a variety of contexts. To this end, the DICE TOSCA technology library uses a new unified approach to handling client side of the hosting platforms, which the cloud orchestrator uses to control the cloud platforms such as OpenStack or Amazon’s EC2. The DICE Deployment Service also stores and handles the platform-related parameters, injecting them on the fly into the blueprint being deployed. This means that the information about target platforms are removed from the blueprint, thereby substantially widening the possible targets of deploying the same blueprint, enabling relatively effortless DIA migration between various test beds and even cloud providers.

Having finished the essential features of the Deployment Tool, we have recognised that services such as Deployment Service and Cloud orchestra tor manager may live outside a relative safety of the development environment. When they operate on public addresses and in public clouds, they become discoverable and accessible to random visitors and potential attackers. To mitigate that, we have ensured that security of the DICE Deployment Tool is present from the moment of its bootstrapping.

Security of the DIAs is a joint responsibility of the support tools and the teams designing and developing them. In this direction, DICE has created a good foundation for the DIAs being designed with elements like encryption of communication channels between services, user-based access restrictions and security features enabled in engines that don't enable them by default. In this area, more work will be required in the future to address the need for creating and exchanging secrets that are involved in Big Data clusters.

Containers are gaining a growing interest of the DevOps communities. In DICE, we do not perceive this technology as a threat, but rather as a complementing technology. Through demonstrators and our own validation, we have shown that the developed features are complete in terms of usability even before containers. Nevertheless, our existing tools can be extended to support Docker, Kubernetes or any other interesting technologies for future commercial versions of the tools.

The Continuous Integration is an important element in the DevOps toolbox. It is therefore not surprising that Jenkins offers a built-in support for projects that are defined in code – a logical extension of the infrastructure as code. Our documented examples of Jenkins pipeline definitions represent a starting point for further streamlining and customization of the end users' projects, where the integration testing definitions live and evolve at the same place as the code being integrated. The only downside of the approach is that the pipeline definitions have to be aware of a specific set-up of the Jenkins masters and slaves topology, which reduces their portability.

We have also updated the final release of the Configuration Optimisation tool, with the primary innovation compared to earlier version being its new Eclipse IDE plugin and Jenkins integration. This addition allows the developer to conveniently start batch executions of BO4CO from within Eclipse.

4.1 DICE Requirement compliance

In the Section 2, we provided a summary of the requirements. indicates the level that the DICE Delivery Tools comply in their initial release. The *Level of fulfilment* column has the following values:

- ✗ – not supported
- ✓ – initial support
- ✓✓ – medium level support
- ✓✓✓ – fully supported

Table 3: Level of compliance of the initial version of the DICE delivery tools with the initial set of requirements

Requirement	Title	Priority	Level of fulfilment
R5.3	Continuous integration tools deployment	SHOULD	✓✓✓
R5.4	TOSCA format for blueprints	MUST	✓✓✓
R5.4.1	Big Data technology support	MUST	✓✓✓
R5.4.2	Translation tools autonomy	MUST	✓✓✓
R5.4.5	Deployment tools transparency	SHOULD	✓✓✓
R5.4.6	Deployment plans extendability	SHOULD	✓✓
R5.4.7	Deployment of the application in a test environment	MUST	✓✓✓
R5.4.8	Starting the monitoring tools	MUST	✓✓✓
R5.5	User-provided initial data retrieval	MUST	✓✓✓
R5.7.1	Data loading hook	SHOULD	✓✓
R5.16	Provide monitoring of the quality aspect of the development evolution (quality regression)	MUST	✓✓✓
R5.19	Deployment configuration review	COULD	✓
R5.20	Build acceptance	MUST	✓✓✓
R5.27	Configuration Optimisation	MUST	✓✓✓
R5.27.1	Brute-force approach for CONFIGURATION_OPTIMISATION deployment	SHOULD	✓✓✓
R5.27.6	CONFIGURATION_OPTIMISATION experiment runs	MUST	✓✓✓
R5.27.7	Configuration optimisation of the system under test over different versions	SHOULD	✓✓✓
R5.27.8	Configuration Optimisation's input and output	MUST	✓✓✓
R5.43	Practices and patterns for security and privacy	MUST	✓✓

In the final version, we can see that we have addressed a great majority of the requirements of priority MUST. As already commented and demonstrated by the use case providers [11], the coverage is satisfactory. We are aware of a few limitations, shown in the requirements with less than full support:

- R5.4.6: the support for this requirement is at medium level. We have provided a validation and evaluation of this aspect in Section 3.1.5. We believe that such support is an advanced functionality, which is not in the main scope of DICE project.
- R5.7.1: while we did not provide a fully built-in support for loading data into Cassandra and MongoDB, there is already a capability for this functionality using the scripting support in DICE TOSCA technology library.
- R5.19: this feature has priority COULD because use cases did not express the need for this requirement. Further, it is already possible to employ a third party code review tool such as Gerrit[16], then protect the main deployment/production in Git, and configuring Jenkins projects to only proceed with release after an ADMINISTRATOR approves changes in Gerrit.
- R5.43: we have demonstrated the basis of security by design on MongoDB as described in Section 3.1.2.2.

[16] https://www.gerritcodereview.com/

References

[1] Balalaie A., Heydarnoori A., Jamshidi P., *Microservices Architecture Enables DevOps: an Experience Report on Migration to a Cloud-Native Architecture*. IEEE Software, 2016.

[2] DICE consortium, *DICE deliverable 5.1: DICE delivery tools – Initial version*, January 2016

[3] DICE consortium, *DICE deliverable 5.2: DICE delivery tools – Intermediate version*, January 2017

[4] DICE consortium, *DICE deliverable 1.2 Requirement Specification*, July 2015

[5] DICE consortium, *DICE deliverable 1.4 Architecture definition and integration plan - Final version*, January 2017

[6] DICE consortium, *DICE deliverable 2.2 Design and quality abstractions - Final version*, January 2017

[7] DICE consortium, *DICE deliverable 2.4 Deployment abstractions – final version*, April 2017

[8] DICE consortium, *DICE deliverable 2.5 DICE methodology*, July 2017

[9] DICE consortium, *DICE deliverable 4.2 Monitoring and Data warehousing tools - Final version*, January 2017

[10] DICE consortium, *DICE deliverable 5.4 DICE testing tools – Initial version*, January 2017

[11] DICE consortium, *DICE deliverable 6.3 Consolidated implementation and evaluation*, July 2017

