Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

Deployment Abstractions

Deliverable 2.4

Version: Final

Deliverable:	D2.4
Title:	Deployment Abstractions - Final Version
Editor(s):	Damian A. Tamburri (PMI), Elisabetta Di Nitto (PMI)
Contributor(s):	Michele Guerriero (PMI), Matej Artac, Tadej Borovsak (XLAB)
Reviewers:	Dana Petcu (IEAT), Vasilis Papanikolaou (ATC)
Type (R/P/DEC):	Report
Version:	1.0
Date:	31-April-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 Union's Horizon 2020 research and innovation programme under grant agreement No. 644869

Executive summary

Data-intensive applications (DIAs) and the Big Data assets these manipulate are key to industrial innovation. However going data-intensive requires much effort not only in design, but also in system/infrastructure configuration and deployment - these still happen via heavy manual fine-tuning and trial-and-error.

In the scope of this deliverable, we outline abstractions and automations that support data-intensive deployment and operation in an automated DevOps fashion, The sum of these abstractions and automations are realised in the form of DICER, which stands for “Data-Intensive Continuous Engineering and Rollout". DICER is an automated tool enabling fast prototyping of DIAs and their deployment in a production-like environment via model-driven Infrastructure-as-Code. DICER is a result of the joined WP2-WP5 effort over task T2.3 whose goals are to provide, on one hand, complete and coherent deployment abstractions to support precise and rigorous infrastructure design and, on the other hand, automations to build fully deployable Infrastructure-as-code blueprints expressed in TOSCA.

To address these objectives, DICER allows to: (a) express the deployment of DIAs on the Cloud, using Unified Modelling Language (UML); (b) automatically generate DIA-specific infrastructure blueprints conforming to the TOSCA (Topology and Orchestration Specification for Cloud Applications) standard and (c) 1-click deploy them in the cloud using a DICER-customised fork of the Cloudify open-source orchestration engine. Evaluating DICER in 3 medium real-life industries we conclude that it does hasten up to 70% the work of prototyping DIAs in the cloud.

Glossary

DICE	Data-Intensive Cloud Applications with iterative quality enhancements
IDE	Integrated Development Environment
MDE	Model-Driven Engineering
UML	Unified Modelling Language
OCL	Object-Constraint Language
DICER	Data-Intensive Continuous End-to-end Rollout
TOSCA	Topology and Orchestration Specification for Cloud Applications
DPIM	DICE Platform Independent Model
DTSM	DICE Technology Specific Model
DDSM	DICE Deployment Specific Model

Contents

List of Figures

List of Tables

1 Introduction

According to the “Worldwide Big Data Technology and Services, 2012-2015 Forecast" by IDC [1], Big Data or *Data-Intensive* applications (i.e., applications acquiring, manipulating and/or generating Big Data), are expected to grow worldwide at an annual growth rate of 40% — about 7 times that of the ICT market as a whole. However, Data-Intensive expertise is scarce, expensive - designing, developing and deploying DIAs is more and more difficult [**surveyBD**].

In this technical report for the scope of the work we conducted in WP2 - T2.3, we focus on the problem of continuously architecting [2] & deploying DIAs on the Cloud. In so doing, this technical report outlines the final version of: (a) the deployment and automation abstractions that provide for DICER notations and automations; (b) the evaluation of DICE abstractions and automations in action. In the previous version of this report (see D2.3) we provided a general overview of three DICE WP2::T2.3 assets: (a) a core set of meta-models we inherited to prepare the production of T2.3 abstractions and automations; (b) the preliminary design of T2.3 automations; (3) the preliminary design of the multi-model perspective for the DDSM abstraction layer. This technical report concludes our work with the above assets, providing for the technical explanations, design choices, and insights over their final version, as integrated in the DICE IDE.

As previously stated, the contents of this deliverable rotate heavily around DICER, a tool to speed up the trial-and-error process that DIA developers have to go through to find optimal deployment configurations when it comes to moving applications into production. DICER is the implementation of DICE DDSM abstraction and automation support worked as part of T2.3 jointly with WP5 to provide automated continuous architecting for DIAs. To this aim, DICER allows design of complex data-intensive infrastructures using UML models customised with DDSM profile notations and abstractions, which are used to automatically generate so called Infrastructure-as-Code. Along this path, DICER brings both DIA design and operations concerns within the same environment, in a typical DevOps fashion. DICER automations use the DDSM abstractions in action to prepare an actionable TOSCA blueprint. DICER is designed combining Infrastructure-as-Code [**toscayaml**] (IasC) and Model-Driven Engineering (MDE) [3].

On one hand, IasC is a typical DevOps tactic that offers standard ways to specify the deployment infrastructure and support operations concerns using human-readable notations [**morris2016infrastructure**]. The IasC paradigm features: (a) domain-specific languages (DSLs) for Cloud application specification such as TOSCA, i.e., the “Topology and Orchestration Specification for Cloud Applications" standard, to program the way a Cloud application should be deployed; (b) specific executors, called *orchestrators*, that consume IasC blueprints and automate the deployment based on those IasC blueprints. DICER comes with our own augmented version of the Cloudify orchestration engine, which supports many of the most popular DIA technologies and is completely transparent to DICER users. Nevertheless, DICER’s automation architecture is independent of any TOSCA-specific orchestrator.

On the other hand, Model-Driven Engineering (MDE) predicates the usage of models to design and semi-automatically develop software systems. DICER allows to create a IasC blueprint from a simple UML deployment model by offering a new UML profile. Also, DICER offers assisted modelling employing OCL constraints: users can model the desired DIA components, using continuous OCL validation to improve their model by adding/editing modelling elements until all constraints are satisfied. DICER’s automation approach is implemented using the Eclipse modelling framework and supports all of the Big Data technologies supported by our underlying Cloud orchestrator, such as Hadoop, Storm, Spark, Cassandra and, soon, MongoDB (unplanned but support is 80% complete).

All of the above happens within the same IDE, an Eclipse installation provided with DICER’s plugins, which are freely available for download[1], soon also on the Eclipse Marketplace.

Evaluating DICE deployment abstractions and automations in action in three medium-sized industrial case-studies as part of the DICE scope, we observed that they do indeed speed up the iterative trial-and-error process developers and operators usually perform, letting them gain back over 70% of

[1] `https://github.com/dice-project/DICER` & `https://github.com/dice-project/DICE-Deployment-Service`

their time. We observed that although DICE deployment abstractions have their own learning curve for practitioners with no prior DIA experience, DICER-assisted design and deployment times grow linearly with the complexity of a DIA, since times reflect a fixed and iterative model-driven procedure. This is a considerable gain against the observed exponential increase of design and deployment times in manual configuration and installation scenarios.

In summary, this technical report outlines 4 novel contributions: (a) a UML profile enabling for designing the deployment of DIAs on the Cloud, (b) a new DIA-enabled implementation of TOSCA, (c) the elaboration of the DICER tool as an automated, model-driven solution and (d) its evaluation over 3 industrial case-studies featuring action research [**ares**], ethnography [4], technology acceptance [**tat**], and design science principles [**hevner2004design**].

1.1 Objectives

This deliverable has the following objectives:

Objective	Description
DDSM Meta-Models	elaborate on the final meta-modelling and UML DICE profile foundations with which we support automated continuous end-to-end rollout of DIAs.
DICER Tool	elaborate the final DICER logic and IasC automations.

1.2 Structure of the deliverable

The rest of this deliverable is structured as follows.

First, Section 2 outlines the main DICE achievements addressed in this deliverable. Second, Section 3 outlines a state of the art overview for us to contextualise the DICE results addressed in this deliverable. Further on, Sections 4, 5 and 6 outline our research solution, namely the DDSM DICE meta-model layer with its refined counterpart the UML DDSM profile, now integrally part of the DICE profile solution. Also, we showcase the final version of the DICER tool and the methods in which the contributions were obtained, while Section 7 evaluates our solution by means of mixed-methods research. Finally, Section 8 concludes the deliverable by elaborating on the future work we intend on the DDSM layer and DICER tool, respectively.

The technical report is accompanied by appendices that provide a complete reference over the DDSM meta-modelling and profile notations, the DICER tool automations, and the accompanying IDE logic behind.

2 Achievements

This section briefly describes the main achievements of this deliverable.

2.1 Achievement 1

We have achieved a final version of the DICE Deployment Modelling abstractions (DDSM) by combining our previously edited and updated version of the MODACloudsML language grammar (called MODAClouds4DICE) with the TOSCA standard v 1.1 YAML profile grammar. Also, The DDSM layer now features a complete outset of UML profiling facilities which terminate the technical specification of the DICER tool. These modelling abstractions have been augmented and tested to support 100% of the technologies required in the DICE technical space; as a result, our notations now contain the necessary concepts required for the full technical support originally claimed by DICE deployment modelling and automation task T2.3 (see the DoW).

In their final version, the above abstractions allow designers to quickly produce a deployable map for the implementable view of the big data application design realised and refined within the DTSM component. Said map essentially relies on core-constructs that are common to any cloud-based application (of which big data is a subset). Similarly to the related DTSM abstraction layer (see Deliverable D2.1), DDSM abstractions come with ad-hoc deployment configuration packages which are specific per every technology specified in the DTSM component library. Designers that are satisfied with their DTSM model may use this abstraction layer to evaluate several deployment alternatives, e.g., matching ad-hoc infrastructure needs. For example, the MapReduce framework typically consists of a single master JobTracker and one slave TaskTracker per cluster-node. Besides configuring details needed to actually deploy the MapReduce job, designers may change the default operational configurations behind the MapReduce framework. Also, the designer and infrastructure engineers may define how additional Hadoop Map Reduce components such as Apache Yarn may actively affect the deployment.

Appendix A contains an overview of all concepts and relations captured within the DDSM-specific meta-models and augments these with their UML DDSM counterparts. Meta-models are outlined in tabular form.

2.2 Achievement 2

We have achieved a fully-integrated and fully-working implementation of the above-mentioned deployment abstractions into the rich automations tool we call DICER, which blends: (a) Model-To-Model transformations that transmute models from a DTSM specification stereotyped with UML DDSM constructs into a TOSCA intermediate and editable format (e.g., for experienced and ad-hoc fine-tuning) as well as (b) a Model-2-Text transformation to produce an actionable TOSCA blueprint profiled with the TOSCA YAML 1.2 TOSCA version. These results constitute the DICER tool - evaluating DICER, we observe that it does in fact speed up the work of our DICE case-study owners by over 70%.

Our evaluation also shows that DICER has reached the stage of a successful final implementation of MDE applied to continuous modelling and continuous deployment of Data-Intensive Applications to be supported in the DICE project.

Appendix B contains a general overview of the DICER tool and its internal logic using snippets of commented code.

3 DDSM and deployment modelling: State of the art overview

There are several works that offer foundational approaches we considered in developing the meta-modelling and domain-specific notations required to support the DDSM meta-modelling layer and the supporting DICER tool. Said works mainly reside in model-driven engineering as well as deployment modelling & automation domains.

A number of works leverage Model-Driven Engineering (MDE) for Big Data. For example, [5] offers facilities for MDE of Hadoop applications. After defining a metamodel for Hadoop the same can be used to define a model of the application; the approach proceeds with automatic code generation. The end result is a complete code scaffold to be complemented by DIA developers with actual implementation of Hadoop MR components (e.g., Map and Reduce functions) instead of placeholders in generated code. The main goal is to show how MDE can reduce the accidental complexity of developing Hadoop MR DIAs. Similar support is offered by Stormgen [6], a DSL for Storm-based *topologies*.

While the above approaches provide a first evidence of the usefulness of MDE in the context of DIAs, both focus on a single technology, while the key challenge in Big Data is the necessity of assembling many technologies at the same time. Moreover, the focus is on development phases with no target to deployment, which is indeed a cost- and time-consuming activity for DIAs. Also, various works explore how to apply model-driven for configuration & deployment but never previously for DIAs.

For example in [**autosar1**] authors elaborate on automated deployment featuring AUTOSAR where the target are complex embedded software systems from the automotive domain. Similarly, in the Cloud domain, Ferry et al. [7], argue for the need for MDE techniques and methods facilitating the specification of provisioning, deployment, monitoring, and adaptation concerns of multi-Cloud systems at design-time and their enactment at run-time. This last work acts as a baseline for our approach, but while in [7] the focus is on general Cloud-applications, we argue that data-intensive applications, although Cloud-based, deserve their own room and there is currently no prior effort to specifically target the configuration and deployment issues of such kind of applications via Model-Driven DevOps.

Moreover, from the UML perspective, there are few initial works which investigate on the usage of UML for designing modern data-intensive applications [**gomez**]. Along this direction, in this work we discuss how we exploited UML in the particular context of designing and automating DIAs deployment.

Finally, from the TOSCA side, being TOSCA an emerging standard, there are several works connected to previous and current research projects both in EU and abroad [**SeaClouds**], [**wettinger**] that discuss how to seamlessly enable the model-based orchestration of application topologies by automatically transforming them into TOSCA and by adopting suitable TOSCA-enabled Cloud orchestrators. Again these works focus on general Cloud-based applications, while it is well recognized by the TOSCA community (e.g., in the recent standard draft examples[2]) that the TOSCA support for DIAs may need further research [**toscapaper**]. Moreover, although there has been some initial effor towards trying to define a connection between UML and TOSCA [**bergmayr14**] [**bergmayr16**], further investigation needs to be done to better address interoperability concerns between these two standards.

[2] `http://tinyurl.com/z3e9x9z`

4 DICE Approach for Deployment Modelling

This section highlights how the DICE DDSM abstractions can combine with the DICER tool to provide modelling foundations for automated IasC production and operation. From an outsider's perspective, the DICE DDSM and DICER abstractions and tools look fairly straightforward to any generic end-user new to DIAs, for example, to Mr. Ben Andjerrys, a senior QA engineer at Icecream Software Inc. Let's assume Ben needs to evaluate various quality aspects of his DIA in a production-like environment, with little to no experience over Big Data technologies and Cloud computing concepts. From Ben's outsider perspective, DICER offers the possibility to do 3 things: a) create the deployment model representation of a DIA, b) automatically generate the TOSCA blueprint corresponding to that deployment model, with the possibility to, c) executing 1-click deployment of that blueprint in a matter minutes. In particular, Ben can go through the following:

1. After installing Eclipse UML Papyrus and the DICER Eclipse plugins, Ben starts by designing the UML Deployment Diagram of its DIA using the DICER UML profile to enrich the model with DIA-specific deployment concepts by applying standard UML stereotypes.

2. Using the DICER UML profile Ben can model his target Cloud infrastructure and map the deployment diagram contents to their respective execution platform, adapting appropriate configurations, along with setting the various Big Data platforms and DIAs that will run on top of it.

3. Using the available UML stereotypes Ben can configure various properties of his model, both at the infrastructural level (e.g. the number of available virtual machines and their sizes, required firewalls, etc) and at the platform level (e.g. the many configuration parameters of the deployed DIA technologies).

4. DICER assists Ben during this model refinement phase by providing modelling constraints (defined in OCL) to be checked continuously until the model is valid. A key aspect captured by the OCL constraints are the technological dependencies; for instance the Apache Storm runtime environment often needs a Zookeeper cluster in order to operate.

5. Using an Eclipse launch configuration, Ben can generate a deployable TOSCA blueprint which can be sent to DICER's own deployment engine.

In this Section we focus the automation ingredients of DICER, that are, (a) its UML profile and (b) the model-driven automations DICER uses to generate deployable TOSCA blueprints. Further on, Section 5 outlines the DICER orchestration service and the TOSCA technology library (for further details see D5.2) at the core of DICER's IasC strategy.

4.1 DICER Explained: Design Principles and Core Elements

To introduce the DICER UML profile we first outline the key design principles behind it as follows:

1. UML-Based Modelling: DICER modelling features UML, a well-known general purpose modelling language offering solid support for the extensions we designed. DICER proposes to tailor UML models towards the specific domain of DIAs through the UML profiling mechanism. The DICER UML profile implementation is an extension of the UML metamodel with DIA-specific *stereotypes*, i.e. tags that can be applied on standard UML elements to capture domain-specific features and constraints. Along this path, DICER models are defined by stereotyping elements in standard, familiar UML models. In particular the DICER profile extends the UML Deployment Diagrams metamodel [3], as the natural choice when dealing with applications' deployment.
2. DIA Runtime Environments: DIA runtime environments feature multiple distributed platforms (e.g.

[3] http://www.uml-diagrams.org/deployment-diagrams.html

execution engines, databases, messaging systems) that are used by running DIAs; each platform runs on a (typically homogeneous) *cluster* of virtual machines (VMs); from the DIA point of view, the resulting cluster appears as a single logical unit. In other words, the actual distribution of the underlying runtime environment is abstracted, resulting in a set of services that developers can combine within a DIA. Most of these distributed platforms simplistically adopt two major styles: (1) peer-to-peer (each node has the same role and privileges) or (2) master-slave (slaves communicate with the master, which in turn coordinates all). The purpose of each DIA platform varies (e.g., processing engines, databases, messaging layers). A complex DIA potentially needs many of them to fully operate. Distributed execution engines (e.g. Hadoop, Spark, Storm) cover a particular role - they are in charge of executing DIAs in a parallel and distributed fashion. DICER modelling offers support for this entire meta-design, to cover for all DIAs and DIA technologies following the above pattern.

3. Composite DIAs: A complex DIA can be composed of multiple sub-DIAs, or “jobs", potentially executed on different platforms. For instance, according to the so-called “Lambda" architecture, a widely adopted architecture style for DIAs, a DIA needs: a) a messaging system to decouple the application from the various data sources, b) a stream processing engine, for executing streaming applications over real time data, c) a batch processing engine for periodically executing batch jobs, d) at least a database for storing data processing results. More in general, a complex DIA can be seen as a set of different *Big Data jobs*, whose execution is adequately orchestrated. DICER is designed to support deployment in these scenarios.

4. DIA Cloud Infrastructure Design: As DIAs typically run on top of Cloud infrastructures, it follows that we need a way to model Cloud infrastructures. DICER draws from MODACloudsML [7], a modelling language for multi-Cloud applications, enabling designers to model the way their applications exploit Cloud resources from different providers. From the MODACloudsML’s perspective, DIAs are Cloud applications organised in various components running on virtual machines (VMs). The key difference with general Cloud applications is that some of these components may operate in *cluster mode*, meaning that they run on a cluster of VMs in a distributed fashion, but externally they appear as a single component.

Given the above considerations, the relevant part of the DICER UML profile implemented in Eclipse UML Papyrus is shown in Figure 1. In the following we elaborate on essential DICER stereotypes; further fine-grained detail can be found in Appendix A.

First, DICER distinguishes between *InternalNode*, or services that are managed and deployed by the application owner, and *ExternalNode* that are owned and managed by a third-party provider (see the providerType property of the ExternalNode stereotype). Both the InternalNode and ExternalNode stereotypes extend the the UML meta-class *Node*.

The *VMsCluster* stereotype is defined as a specialisation of ExternalNode, as renting computational resources such as virtual machines is one of the main services (so called Infrastructure-as-a-Service) offered by Cloud providers.

VMsCluster also extends the *Device* UML meta-class, since a cluster of VMs logically represents a single computational resource with processing capabilities, upon which applications and services may be deployed for execution.

A *VMsCluster* has an *instances* property representing its replication factor, i.e., the number of VMs composing the cluster.

VMs in a cluster are all of the same size (in terms of amount of memory, number of cores, clock frequency), which can be defined by means of the *VMSize* enumeration. Alternatively the user can specify lower and upper bounds for the VMs’ characteristics (e.g. minCore/maxCore, minRam/maxRam), assuming the employed Cloud orchestrator is then able to decide the optimal Cloud offer, according to some criteria, that matches the specified bounds.

The VMsCluster stereotype is fundamental towards providing DICER users with the right level of abstraction, so that they can model the deployment of DIAs, without having to deal with the complexity exposed by the underlying distributed computing infrastructure. In fact, an user just has model her clusters of VMs as stereotyped Devices that can have nested InternalNodes representing the hosted distributed platforms.

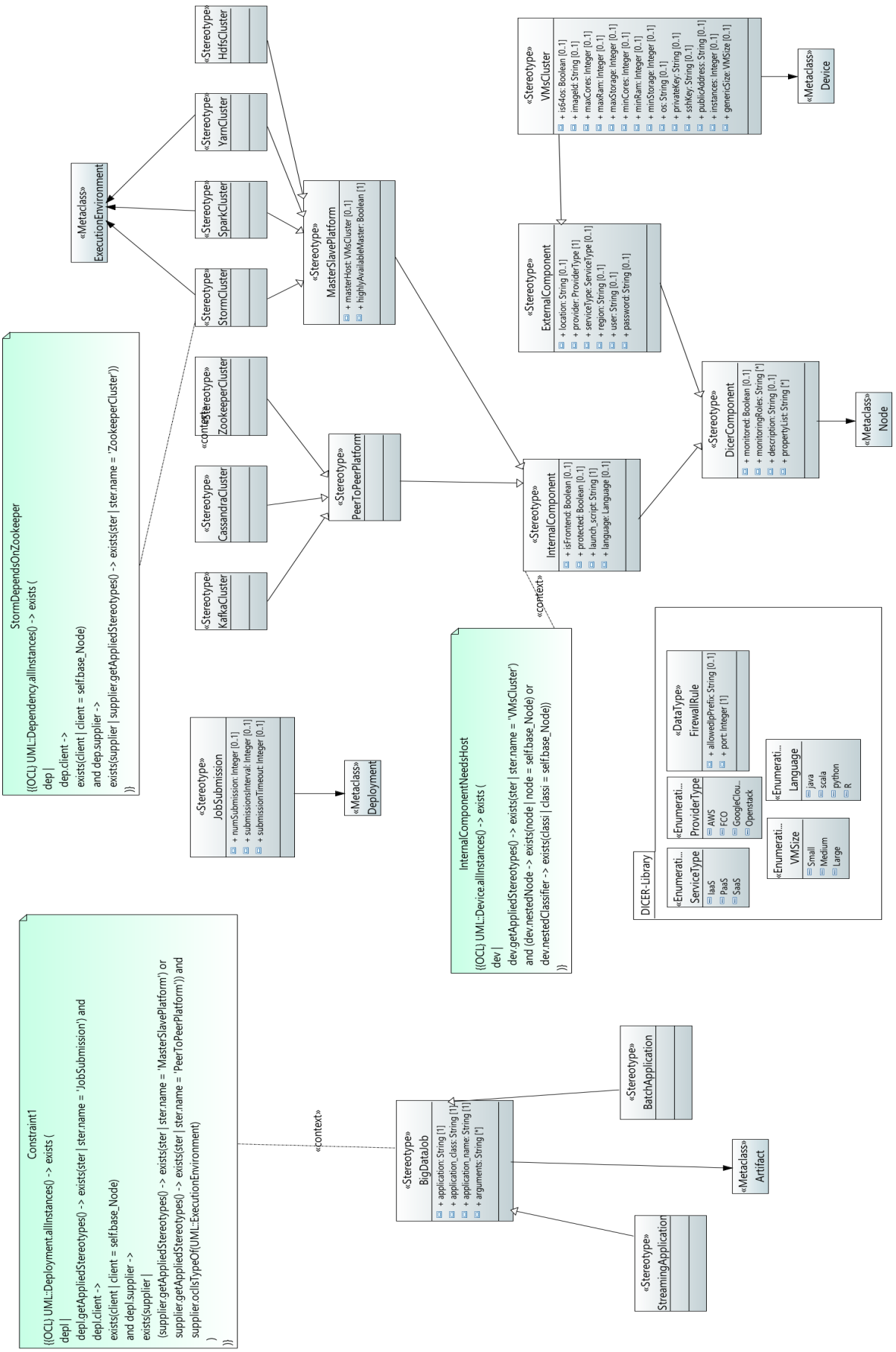


Figure 1: An extract of the DICER UML Profile for DIAs deployment.

Furthermore, a specific OCL constraint imposes that each InternalNode must be contained into a Device holding the VMsCluster stereotype, since by definition an InternalNode have to be deployed and managed by the application provider, which thus has to dispose the necessary hosting resources.

We then define DIA-specific deployment abstractions, i.e. the *PeerToPeerPlatform*, *MasterSlavePlatform* stereotypes, as further specialisations of InternalNode. These two stereotypes basically allow the modelling language to capture the key differences between the two general type of distributed architectures. For instance the MasterSlavePlatform stereotype allows to indicate a dedicated host for the master node, since it might require more computational resources.

By extending our deplyoment abstractions, we implemented a set of technology modelling elements (*StormCluster*, *CassandraCluster, etc.*), one for each technology we support. DIA execution engines (e.g. Spark or Storm) also extend UML *ExecutionEnvironment*, so to distinguish those platforms DIA jobs can be submitted to. Each technology element allows to model deployment aspects that are specific to a given technology, such as platform specific configuration parameters (that have been omitted from Figure 1 for the sake of space) or dependencies on other technologies, that are enforced by means of OCL constraints in the case they are mandatory - see the constraint on the StormCluster stereotype in Figure 1.

The *BigDataJob* stereotype represents the actual application that can be submitted for execution to any of the available execution engine. It is defined as a specialisation of UML Artefact, since it actually corresponds to the DIA executable artefact. It allows to specify job-specific information, for instance the *artifactUrl* from which the application executable can be retrieved.

The *JobSubmission* stereotype, which extends UML Deployment, is used to specify additional deployment options of a DIA. For instance, it allows to specify job scheduling options, such as how many times it has to be submitted and the time interval between two subsequent submissions. In this way the same DIA job can be deployed in multiple instances using different deployment options. An additional OCL constraint requires each BigDataJob to be connected by mean of JobSubmission to a UML ExecutionEnvironment which holds a stereotype extending one between the MasterSlavePlatform or the PeerToPeerPlatform stereotypes.

4.2 A UML Profile for DIAs' Deployment

We showcase the defined profile by applying it to model the deployment of a simple DIA that we called Wikistats, a streaming application which processes Wikimedia articles to elicit statistics on their contents and structure. The application features Apache Storm as a stream processing engine and uses Apache Cassandra as storage technology.

Wikistats is a simple example of a DIA needing multiple, heterogeneous, distributed platforms such as Storm and Cassandra, Moreover Storm depends on Apache Zookeeper. The Wikistats application itself is a Storm application (a streaming job) packaged in a deployable artefact. The resulting UML Deployment Diagram is shown in Figure 2. In this specific example scenario, all the necessary platforms are deployed within the same cluster of medium-sized VMs from an Openstack installation.

Each of the required platform elements is modelled as a Node annotated with a corresponding technology specific stereotype. In particular Storm is modelled as an ExecutionEnvironment, as it is the application engine that executes the actual Wikistats application code. At this point, fine tuning of the Cloud infrastructure and of the various platforms is the key aspect supported by DICER. The technology stereotypes allow to configure each platform in such a way to easily and quickly test different configurations over multiple deployments and enabling the continuous architecting of DIAs.

The dependency of Storm on Zookeeper is enforced via the previously discussed OCL constraints library which comes automatically installed within the DICER tool. Finally the deployment of the Wikistats application is modelled as an Artefact annotated with the BigDataJob stereotype and linked with the StormCluster element using a Deployment dependency stereotyped as a JobSubmission. Finally BigDataJob and JobSubmission can be used to elaborate details about the Wikistats job and how it is scheduled.

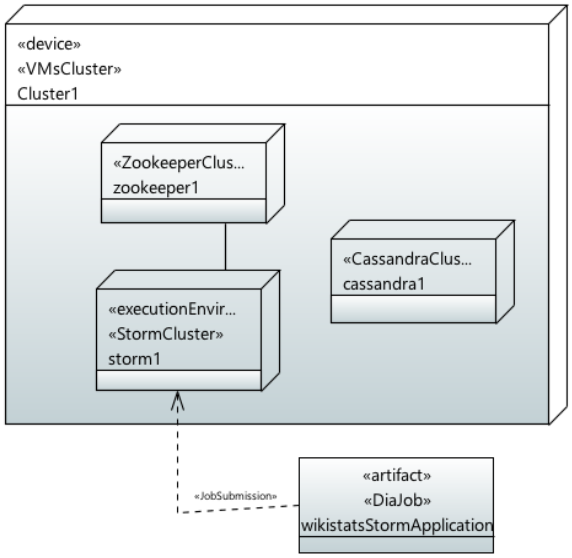


Figure 2: Deployment Diagram for the Wikistats DIA.

4.3 DICER Automations

The automations behind DICER feature model transformations as ways to make DIA models come to life in an automated fashion. We use model transformations to generate declarative code that describes the deployment of DIAs on the Cloud, fully embracing the Infrastructure-as-Code paradigm. We selected TOSCA[**toscayaml**, **toscapaper**] as target IasC specification language as it is an OASIS[4] standard and it is supported by a variety of deployment engines. Our key original contribution to TOSCA and TOSCA-enabled orchestrators is a TOSCA implementation which provides general support for Big Data platforms and DIAs. For this contribution, we designed a DIA specific *TOSCA Technology Library* (see Section 5) implemented in our own extensions for the Cloudify orchestrator engine. We then implemented a model transformation which targets such TOSCA implementation. The transformation automates the translation of DICER UML models into TOSCA-compliant equivalents that comply to out DIA-specific TOSCA implementation, i.e. the transformation instantiates the TOSCA types defined in our *TOSCA Technology Library*. Finally the transformation serializes the TOSCA model into YAML formatted IasC, according to the TOSCA YAML profile[5]. In the following we elaborate on the main rules of our transformation engine, instantiating them on our running example, i.e. the Wikistats application. In our scenario we want to deploy Wikistats on a single cluster of VMs. The VMsCluster is transformed into a TOSCA node template whose type depends on the size of the VMs, if the case the genericSize attribute of the VMsCluster stereotype is set, otherwise a *dice.hosts.Custom* node template is generated, which can be fully configured. In particular it is possible to specify the desired number of VM instances (see listing 1).

For each technology that the cluster will host that the user wants to deploy in secure mode (i.e. by enabling a pre-defined set of firewalls), the transformation generates a node template from the package *dice.firewallrules* and adds on the host nodes the *dice.relationships.ProtectedBy* relationship with the generated node template. listing 1 shows the generated TOSCA code for the VMsCluster used in the Wikistats example:

Each Node annotated with a technology stereotype is transformed into the corresponding set of TOSCA constructs. Depending on how much the employed TOSCA implementation fits the DICER modelling language, the transformation can exploit the generality of the MasterSlavePlatform and Peer-

[4] `http://oasis-open.org/`

[5] `http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/cs01/TOSCA-Simple-Profile-YAML-v1.0-cs01.pdf`

Listing 1: Sample TOSCA Properties

```
vm1:
    instances: {deploy: 1}
    type: dice.hosts.Large
    relationships:
      - {type: dice.relationships.ProtectedBy, target:
          ↪ cassandra_vm1_seed_firewall}
      - {type: dice.relationships.ProtectedBy, target: storm_nimbus_firewall}
//MISSING NODES FIREWALL
```

ToPeerPlatform stereotypes, processing all the corresponding sub-stereotypes in the same way. Our TOSCA library has been design along this direction. The TOSCA library provides specific relationships to configure connections between nodes of a platform, as well as between different platforms. For instance the Storm package of our library provides the *ConnectedToNimbus* relationship to specify the connection between slaves nodes and the master node, while the *ConnectedToZookeeperQuorum* relationship allows to specify the connection of a Storm cluster to a Zookeeper cluster. Essentially, the Zookeeper Quorum node is a logical node template, which serves in the orchestrator as a configuration coordinator (and it is also a workaround for certain Cloudify's limitations). Zookeeper needs to have each of its nodes in a cluster set up first, then all of them need to know about the addresses about all of the other ones. The Zookeeper Qorum helps gather information about all Zookeepers' dynamic addresses, then via an appropriate relationship, it also configures each one of them so that they know of each other. The transformation also properly set into the TOSCA blueprint all the technology specific configuration parameters. Finally, all the *dice.relationships.ContainedIn* relationships between technology nodes and their hosting clusters of VMs are properly instantiated. The TOSCA snippet in listing 2 is the DICER-generated blueprint for the the StormCluster and the ZookeeperCluster elements in the Wikistats deployment.

Concluding the WikiStats modelling toy example, the generated blueprint is now automatically sent to the DICER Deployment and Orchestration Service (further details are available on D5.2) for actual deployment (see Section 5). Evidence of orchestration of this very same blueprint is available online[6].

5 DICER Orchestration Service

An orchestration service is a persistent service for handling the parts of the application life-cycle which are concerned with *deploying* and *dismantling* a software system. DICER orchestration service[7] was worked out as part of the technical outputs of WP5 and consists of:

1. A *front-end service*, that exposes a RESTful interface. It receives the TOSCA blueprints by the DICER modelling tool and provides handling of commands for deployment and un-deployment of the applications. Moreover, the front-end service allows system administrators to assign parameters describing the infrastructure context (e.g., virtual machine images, Cloudify credentials) - these details are necessary for all application deployment scenarios, but should not be into DICER users' role, therefore use editable defaults.

2. A *cloud orchestrator engine*, which is the active element consuming the TOSCA blueprint. In the scope of DICER, we augmented an existing orchestrator called Cloudify[8] with DIA specific handling and monitoring features while also providing a previously inexistent Eclipse IDE integration plugin. Cloudify itself, however, comes with a set of platform plug-ins to enable provisioning of the Cloud resources such as computation, networking and storage in a platform such as OpenStack or Amazon EC2.

[6] https://github.com/dice-project/DICE-Wikistats
[7] https://github.com/dice-project/DICE-Deployment-Service
[8] http://getcloudify.org/

Listing 2: A TOSCA Blueprint for WikiStats

```
  storm_nimbus:
  properties:
  configuration: {monitorFrequency: 10, queueSize: 100000, retryInterval: 2000,
  retryTimes: 5, supervisorTimeout: 60, taskTimeout: 30}
  relationships:
  - {target: vm1, type: dice.relationships.ContainedIn}
  - {target: zookeeper_quorum, type: dice.relationships.storm.
      ↪ ConnectedToZookeeperQuorum}
  type: dice.components.storm.Nimbus
  storm_nimbus_firewall:
  type: dice.firewall_rules.storm.Nimbus
  storm_vm2_supervisor:
  properties:
  configuration: {cpuCapacity: 400, heartbeatFrequency: 5, memoryCapacity: 4096,
  workerStartTimeout: 120}
  relationships:
  - {target: vm2, type: dice.relationships.ContainedIn}
  - {target: storm_nimbus, type: dice.relationships.storm.ConnectedToNimbus}
  - {target: zookeeper_quorum, type: dice.relationships.storm.
      ↪ ConnectedToZookeeperQuorum}
  type: dice.components.storm.Worker
  zookeeper_quorum:
  relationships:
  - {target: vm2, type: dice.relationships.zookeeper.QuorumContains}
  type: dice.components.zookeeper.Quorum
  zookeeper_vm2_server:
  properties:
  configuration: {initLimit: 10, syncLimit: 5, tickTime: 1500}
  relationships:
  - {target: vm2, type: dice.relationships.ContainedIn}
  - {target: zookeeper_quorum, type: dice.relationships.zookeeper.MemberOfQuorum
      ↪ }
  type: dice.components.zookeeper.Server
  zookeeper_vm2_server_firewall:
  type: dice.firewall_rules.zookeeper.Server
```

5.1 DICE TOSCA Technology Library

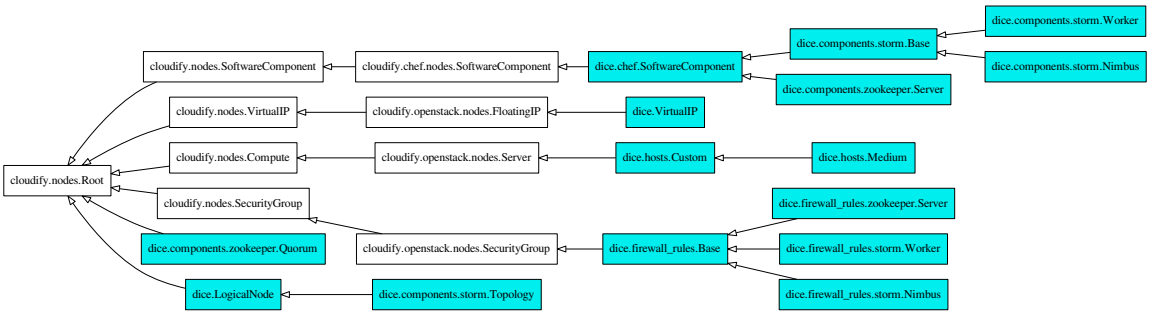
The *DICE TOSCA Technology Library*[9] is a collection of (1) node type definitions for modelling DIAs and (2) TOSCA interface implementations to bring up the modelled DIAs.

The node type definitions of all the technologies that we support serve as an abstraction for the concepts that are understandable to the application orchestration engine. A presentation of the DIA using TOSCA and our technology library is equivalent to the one using the DICER metamodel. The two presentations therefore contain the same level of intent and granularity. As a result, the DICER model transformation can produce TOSCA blueprints agnostic to orchestrators.

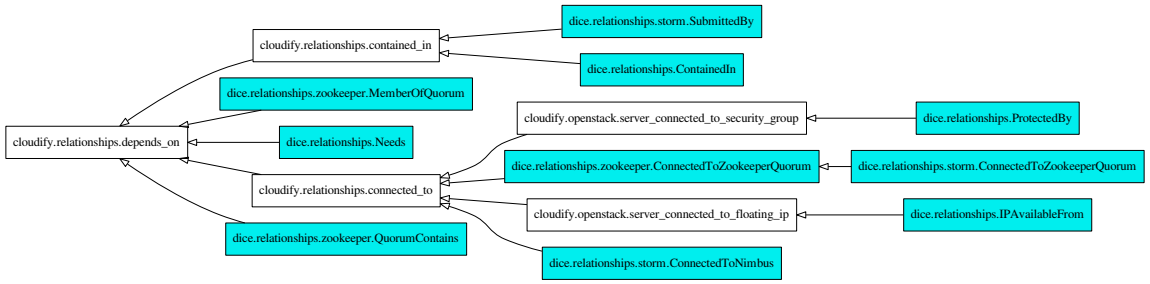
The type definitions consist of all the concepts that are needed in modelling a DIA, e.g., virtual machines or hosts, specific services, their relationships, etc. For example, Listing 2 shows a blueprint for a Storm application, which employs inheritance hierarchy as shown on Fig. 3: subfigure (a) shows a hierarchy of the node types, and subfigure (b) shows inheritance for relationships. Boxes in blue (right-hand side) highlight types and relationships from the library.

A major limitation behind our library is that our TOSCA definitions inherit Cloudify's own TOSCA definitions for the platform concepts which slightly deviate from the Standard but is consistent with TOSCA notations used in major TOSCA-enabled orchestrators such as ARIA TOSCA[10], Indigo[11],

[9] https://github.com/dice-project/DICE-Deployment-Cloudify
[10] http://ariatosca.org/
[11] https://www.indigo-datacloud.eu/



(a)



(b)

Figure 3: the DICER TOSCA Technology Library, a sample for Apache Storm featuring node types (a) and relationship types (b);

Apache Brooklyn[12] or ECoWare [**ecoware**, **ecoware1**]. Cloudify offers a solid basis for DIAs orchestration, including a complete implementation of the cloud orchestration and working plug-ins for popular cloud platforms. Moreover, Cloudify was (and still is) the most advanced stable solution compatible with TOSCA at the time of our project's initiation.

The `interface` section of the node types and relationships declare what steps and logic need to be applied by orchestration engine during execution of orchestration workflows. This consists of (1) definition of intermediate steps that need to occur during the deployment to gather dynamic parameters (e.g., host names and addresses that might be different with each deployment) of the related nodes; and (2) configuration of the individual hosts (virtual machines). Implementation of (1) is specific to Cloudify and takes form as its plug-in. To implement (2), we followed the best practice and used an engine, which enables expressing an intent of a host-level configuration and takes the necessary steps to apply them. For our technology library, we employ the Chef configuration manager. The decision to use Chef was due to our past experiences and an existing repository of cookbooks[13]. Conversely, any other configuration management engine such as Ansible[14] could equally apply for the task. We built or refined Chef cookbooks to support all major stages in DIA orchestration (initialisation, configuration, installation, starting, stopping). Normally, the cookbooks from the Chef marketplace are built to configure each individual host in a single step. Orchestration of a cluster requires configuration to take multiple stages to address dependencies in the cluster. For example, services like Apache Zookeeper require that all peer services are installed in the first stage, then in the second stage each node is configured with a list of all peers' addresses. Only then a functioning Zookeeper cluster is bootstrapped.

From Fig. 3 we can see that our approach in defining our technology library is to make a clean abstraction for the end user from the underlying technology. Replacing Cloudify would require updating

[12] `https://brooklyn.apache.org/learnmore/`

[13] a sample of the cookbooks we elaborated is available online: `https://github.com/dice-project/DICE-Chef-Repository`

[14] `https://www.ansible.com/`

the technology library, but the DICER transformation tool would not need to change. Also, any existing blueprints would remain compatible. In our technology library's Chef recipes, there is only a minimal amount of dependency on the Cloudify implementation. With a relatively small effort, they may be ported to other orchestrators.

Finally, at the time of writing this article, from an infrastructure provisioning point of view, the DICER orchestrator supports the OpenStack and FCO[15] cloud infrastructures, soon to be extended to Amazon EC2.

5.2 WikiStats: 1-Click Deployment

Listing 2) shows the blueprint generated from the WikiStats model ready for submission. DICER launch configurations allow to access the RESTful API of the DICER deployment service directly from the Eclipse launch-configurations console. The Deployment Service receives and parses the blueprint and constructs a directed acyclic graph of the TOSCA topology. Relationships between nodes define the order in which the individual nodes get provisioned, deployed and configured. Topology construction typically starts from building network firewalls (labelled `zookeeper_vm2_server_firewall`, `cassandra_vm1_seed_firewall`, `cassandra_vm2_worker_firewall` and `storm_nimbus_firewall` in the Listing) and virtual machines (`vm1` and `vm2`). These steps happen concurrently Next, the virtual node `zookeeper_quorum` is instantiated, being in charge of proper propagation of interdependent services in the cluster. The main cluster services follow, first the management services `zookeeper_vm2_server`, `storm_nimbus` and `cassandra_vm1_seed`, then the `storm_vm2_supervisor` and `cassandra_vm2_worker` services. The last to run is `wikistats_application`, the user's application being submitted to the running Storm cluster; this step wraps the DICER process, returning a working application as modelled. According to our experiments (as showcased in a recent Webinar hosted by the GigaSpaces Foundation[16]) application deployment times fluctuate around 20—45 min depending on an application complexity between 5 and 10 DIAs (see more in Section 6.2.2).

[15] `https://www.flexiant.com/flexiant-cloud-orchestrator/`

[16] `http://getcloudify.org/webinars/devops-data-intensive-app-preso.html`

6 DDSM and DICER in Action: Evaluation

DICER evaluation features a 1-year action-research initiative involving a total of 12 practitioners in 3 medium-sized companies involved in the DICE project, namely, Prodevelop[17], NetFective[18] and ATC[19].

6.1 Evaluation Setting and Industrial Objectives

First, Prodevelop's (PRO) challenge is to quickly redeploy their trademark geo-fencing and maritime operations control software called *PosidoniaOperations* in multiple sea-port authorities. The software in question was already existing and comprises a total of 9 DIA components for which current deployment times are in the range of 1-3 days involving up to 2 technicians working full-time - Prodevelop expected to improve its deployment productivity times by at least 50% (see page 8 of D6.1 online: `http://tinyurl.com/jx33rbo`). In other words, Prodevelop expects to reduce its deployment times from 21 to 10 working hours at most (i.e., 600 minutes).

Second, NetFective (NETF) technology is completely new to the Big Data domain - their interest is in developing and deploying *BigBlu*, a tax-fraud detection eGovernment software that processes huge amounts of privacy-sensitive financial transactions; their challenge is that their application (which needs to involve 10 DIA components by design) needs to be prototyped (without any prior learning curve) and re-configured/experimented upon as quickly as possible, such that privacy, security and performance requirements are all met at once. When we started to experiment with them, NETF managed to deploy a preliminary prototype in a canary infrastructure environment investing around 5 days with a junior engineer, with later (re-)deployment times settling around to 400 minutes with ad-hoc scripts - NETF expected to improve its technological learning curve and deployment productivity by at least 50% (see page 8 of D6.1 online`http://tinyurl.com/jx33rbo`). NETF expects to reduce deployment times from 400 to 200 minutes at most.

Finally, the Athens Technology Center (ATC) owns *News-Asset* a trademark information discovery application, processing big data from several sources online, including a big data social sensing DIA. The application counts 5 DIA components in total and ATC's main challenge is that this application needs to be experimented upon iteratively and re-configured for a specific performance requirement every time the application is changed/refined. ATC currently experiments with their application using ad-hoc scripts that take up to 180 mins and expects to improve their DIA experimentation times by at least 50%[20].

6.2 Evaluation Methods

Our evaluation conjecture was that DICER helps all three industrial practitioners by speeding up their deployment times.

To address the above evaluation conjecture, our action research initiative followed a "Living-Lab" approach [**HigginsK14**] whereby we directly involved our industrial practitioners who played the part of active co-innovators in designing and refining the DICER tool by directly trying out over 12 versions of the DICER tool; after each trial the tool was incrementally refined following bi-monthly tool-improvement reports. Industrial DICER trials required practitioners to model, deploy and change/improve their own application model after a sample deployment, reporting their observations and experiences in free-form through weekly open-ended interviews.

After the above action research, a final technology acceptance testing and evaluation assessment for DICER was gathered by means of: (a) chronometric assessment ethnographic self-reports [4] - this was done to understand the amount of time saved by using the last version of the DICER tool in action, against original deployment and operations tasks and consequent improvement expectations (see Sec. 6.2.1); (b) self-assessment questionnaires structured according to design science principles [**hevner2004design**] - this was done to understand the end-status of the DICER tool, along with venues for further research

[17] `https://www.prodevelop.es/`

[18] `https://www.bluage.com/company/netfective-technology-group`

[19] `http://www.atc.gr/default.aspx?page=home`

[20] More details on these industries, their case-study, and their objectives is available online: `http://tinyurl.com/jx33rbo`

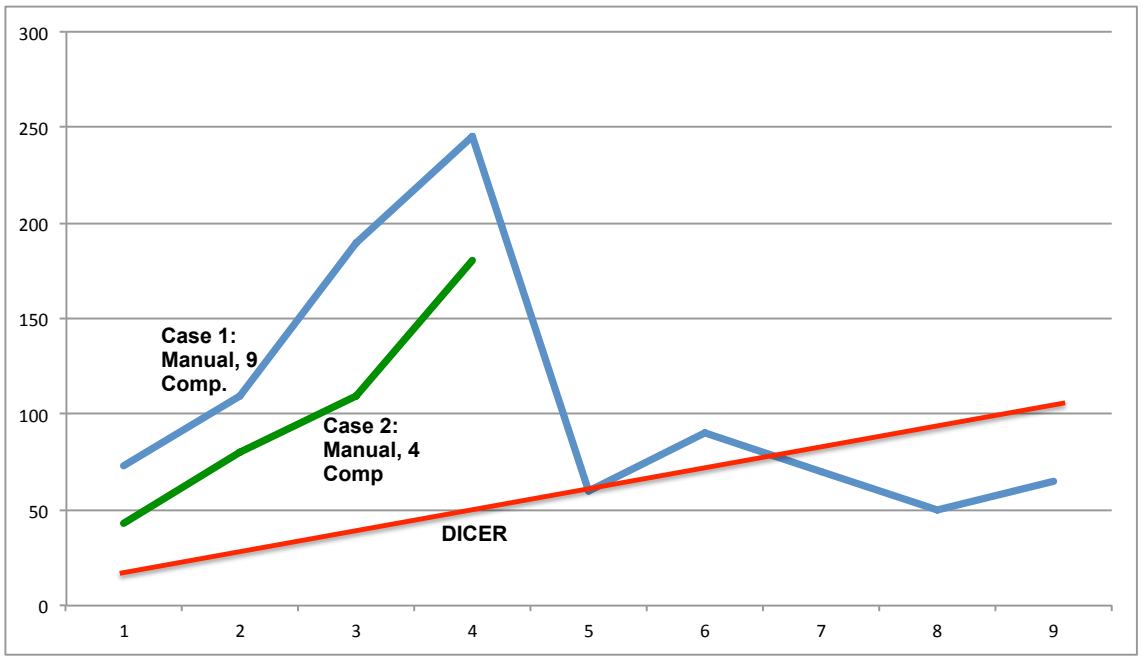


Figure 4: Evaluating DICER in industry, manual design and deployment times (in min, Y-Axis) per number of DIA component (X-Axis); Case 1 and Case 2 reflect industrial cases without DICER and are compared to DICER itself (red line) being used for modelling and deployment in the same scenarios.

and refinement (see Sec. 6.2.3). A complete sample of the reports are available online[21] to encourage verifiability.

6.2.1 Chronometric Assessment

Ethnography is the qualitative empirical evaluation of personal direct experience concerning the elaboration of a treatment [4]. In our case, the final DICER tool (i.e., the treatment) was deployed indoor to all 3 industrial partners for them to evaluate by means of junior developers without prior knowledge or experience with DIAs and DIA middleware. As an evaluation sample to encourage verifiability of our ethnographic reports, we offer online[22] the timesheet reports generated in 2 of our industrial partner premises, ATC and NETF - the reports show that DICER automation reaches as high as 74%, averaging out around 72% - a considerable gain against expectations from all partners[23]. Moreover, Fig. 4 plots the increase in infrastructure design and operations script development times as the number of DIA components involved increases, as observed in 2 of our experiments. Both case 1 (NETF, 10 DIA components) and case 2 (ATC, 4 DIA components) show a super-linear nature which our practitioners explained as being connected to the hand-coding, trial-and-error infrastructure design and deployment exercises. Conversely, Fig. 4 also outlines the modelling times at the increase of DIA components using DICER: linearity of the DICER curve is reportedly connected to its well-structured, model-driven feature allowing reiteration of the same modelling procedure briefly outlined in the previous sections[24]. In conclusion, these times indicate strongly that DICER shows promise in bridging the gap between DIAs (re-)design, their (re-)configuration, and operations using DevOps.

6.2.2 Deployment Time Evaluation

An important validation criterion for the DICER is that the deployments need to succeed in a reasonable time. The automated nature of the DICE's deployment service allowed for measuring the times that

[21] `http://tinyurl.com/gtprpwp`
[22] `http://tinyurl.com/gtprpwp`
[23] Chrono assessment for PosidoniaOps is omitted at our partners' request.
[24] more details on the DICER modelling procedure are available online: `http://tinyurl.com/zatsf6t`

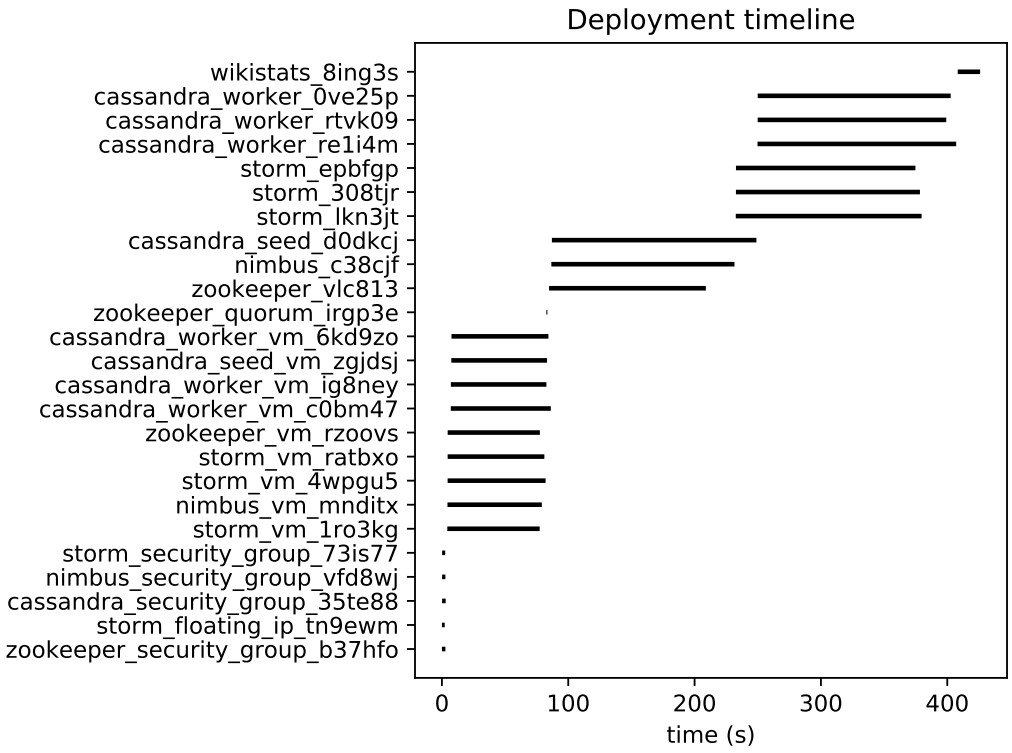


Figure 5: Wikistats application deployment timeline.

the service (and the orchestrator) need to carry out a required operation.

To quantify the time needed to deploy a blueprint, we ran repeatedly (ten times) our WikiStats blueprint deployment and timed its duration from submission until the DICE deployment service declared that the deployment has succeeded. We measured the time required for complete teardown of a blueprint in similar manner: we took a fully deployed blueprint, started teardown procedure and measured the time until the DICER deployment service reported termination. We ran the experiments on XLAB's internal OpenStack Mitaka infrastructure.

The results show that the deployment takes approximately 7.4 minutes, with the slowest time being less than a minute slower from the fastest one. The teardown takes around 1.2 minutes.

Figure 5 shows a typical sequence of node types being deployed for the WikiStats application. The chart also shows that this application runs on top of a stack with up to 3 tiers of dependence, and the WikiStats application itself makes the 4th tier. The node templates named `*securitygroup*`, `*ip*` and `zookeeperquorum*` are all virtual resource instantiations, such as floating IPs and firewalls, which take a minimal time to perform. Then the `*vm*` nodes follow, bringing up the virtual machines. Then the service configuration can follow, according to the derived dependency graph, and WikiStats can run.

6.2.3 Final Acceptance Evaluation

To further confirm the validity of our results, we applied design science principles [**hevner2004design**] and distilled a technology-acceptance questionnaire for our industrial partners[25]. The goal of our evaluation here was to understand the industrial impact of the final DICER prototype along with its automations in terms of continuous architecting. For the purpose of this evaluation, the models and blueprints produced in the scope of our ethnomethodological evaluation (see Sec. 6.2.1) of DICER were discussed in 3 rounds of interviews and 7 focus-groups structured according to a clear-cut design science questionnaire. The results of this exercise can be summarised in 2 key evaluation points:

- **a.** DICER does in fact speed-up (re-)deployment for savings around 70% of the time, even more than our original estimates [8]. However, for scenarios where interoperation with previous deploy-

[25] questionnaire and sample responses are available online: `http://tinyurl.com/zocnogu`

ment technologies is critical, DICER slows down considerably the deployment process due to the difficult contiguity with previously existing deployment technologies. For example, Prodevelop often faces the need of blending pre-existing infrastructures (e.g., Drools configuration scripts) and legacy assets - using DICER regardless of previously existing formats and notations may dilute deployment times. Further research should be invested in procuring more fine-grained identification of improvement “hotspots" on produced TOSCA blueprints — this is reportedly needed for several reasons but especially since migrating from a previous deployment solution to DICER requires comparing the benefits of DICER blueprints with previous scripts — this comparison needs more automation.

- **b.** DICER offers speed at the expense of heavy modelling which may be expensive in small-medium enterprises (SMEs). For example NETF declared that their tax-fraud detection application is supposed to be installed on client premises and never actually migrated. As a consequence, part of the effort of modelling using more elaborate notations in DICER may go wasted due to lack of further incremental and iterative use of the models themselves beyond a single first deployment. This consideration led us to refine DICER with the possibility to exploit configuration defaults for adopted technological packages - i.e., simplify the modelling as much as possible. Further iteration with the company is planned to verify that our extensions fulfil their expectations.

research is needed to simplify the notation and should be invested in studying and refining model-to-model transformations that increase the automation and convenience behind using DICER, e.g., to increase interoperation between previously existing artefacts and other modelling technologies. This effort could try to address seamless and possibly effortless migration or modelling information interchange with DICER.

7 Discussion and Observations

First of all, DICER was evaluated by means of qualitative, quasi-quantitative (i.e., by means of content analysis and quantifications stemming from qualitative data), and quantitative evaluation — evaluation confirms DICER usefulness and technological design acceptance following rigorous design science procedures. While the quantitative and quasi-quantitative evaluations are valid, the qualitative ones cannot be generalised to any possible DICER application scenario for several reasons, for example: (a) they may involve the biases intrinsic to human intervention and the usage of human subjects alone; (b) they do not rely on accurate machine-based time-estimations but on time-sheets and connected chronometric observations; (c) they use interviews and opinion-based observations by industry stakeholders. More specific industrial validation, perhaps involving larger companies and larger product lines based on big data frameworks should be invested to confirm and generalise the validity of DICER beyond the three cases involved in this study. To strengthen this threat, we triangulated the final acceptance of the DICER tool in industry re-iterating the technology acceptance interview with two experienced researchers from Academia[26]. The outcomes confirm the validity of our solution but also point out to several venues for improvement, discussed further in the conclusions. Moreover, the times plotted in Fig. 4 assume an up-front fixed-cost investment for the DICER learning curve which we were recently invited to showcase by Gigaspaces Corp[27]. We observed that an up-front investment of 1-3 hours is sufficient to familiarise with DICER[28].

Furthermore, both the technological acceptance rounds in industry and accademia highlighted that DICER is limited by its own technological stack and the assumptions that entail its design. Even our own evaluation reports on its limitations in biasing big data solutions towards the frameworks and technologies currently supported. We plan to extend the DICER technological library with Apache TEZ, HBase and Flink. Finally, although DICER is set within a DevOps way of working, it is currently still limited in its ability to provide direct operations feedback models. For example, it is currently not possible for DICER to modify directly TOSCA blueprints with feedback from operations or monitoring infrastructures - closing this feedback loop needs to be done manually by designers, developers and operators. Nevertheless, within the DICE H2020 project where DICER was born, we are currently working to refine said direct DevOps continuous feedback, assisting DevOps in a more actionable way.

[26] Interview Guides for this side are available online:`http://tinyurl.com/jjsvvg7`

[27] Gigaspaces is the owner of the Cloudify product and the community around it - `http://www.gigaspaces.com/`

[28] our invited Gigaspaces Cloudify Webinar on DICER is available online:`http://getcloudify.org/webinars/devops-data-intensive-applications-webinar.html`

8 Conclusions

8.1 Summary

Summarising, the main achievements of this deliverable in relation to its initial objectives:

Objective	Achievement
DDSM Meta-Models	We have achieved a final version of the DICE Deployment Modelling abstractions (DDSM) by combining an edited and updated version of the MODACloudsML language grammar (called MODAClouds4DICE) with the TOSCA standard updated to the latest release-draft of the YAML Simple Profile v 1.2[29] accessible only to OASIS TOSCA voting members such as ourselves. Thus distilled, the DDSM core model has been extended to support 100% of the technologies originally committed in DICE; as a result, our DDSM abstractions now contain the necessary concepts required for DICE deployment modelling.
DICER Tool	We have achieved a fully-integrated implementation of (a) Model-To-Model transformations that transmute models from a DTSM specification stereotyped with DDSM constructs into a TOSCA intermediate and editable format (e.g., for experienced and ad-hoc fine-tuning) as well as (b) a Model-2-Text transformation to produce actionable TOSCA blueprints. We conclude that DICE WP2 Task T2.3 is successful and results are outstanding.

8.2 Mapping with DDSM-related Requirements

This section offers a full mapping between initial DICE requirements, their improvements and revisitations over the months in Y1 and Y2 of DICE (Columns 1,2), and the features in the current version of the DICER tool and connected technologies (columns 4 and following). Our requirements compliance is over 90% - requirements remaining unfulfilled are connected to: (1) design limitations of the baseline technologies we considered (e.g., the Cloudify orchestration engine); (2) “Should-have" features which were eventually deemed out of scope by the WP2+WP5 DICER task-force, such as support for deployment and rollout of Network-Function Virtualisation for DIAs; (3) privacy and security concerns to be supported in DICE which were never previously foreseen as being in the scope of DICE and are currently object of further research, development, and evaluation.

8.3 Further work beyond DICE

This deliverable concludes our efforts over providing abstractions and automations to support the straightforward design and rollout of data-intensive applications for the cloud in a model-driven fashion. In continuation of our efforts well beyond the scope of the DICE EU H2020 project, in the future we are planning to: (a) refine the technical possibilities behind the DICER tool, possibly strengthening its technological support; (b) further refine the automations behind DICER extension, possibly by means of additional model-driven facilities; (c) enhancing DICER integration and user-experience within a more complete deployment analysis and design toolkit (i.e., the DICE IDE); (d) extend and refine DICER to support privacy-by-design within the scopes and intents and purposes possible at this point in the DICE timeline.

Table 1: DICE DDSM-related Requirements, an overview with resolution.

Req. ID	***Description***	***Source***	***Satisf.***	***Rationale***
PR2.0	Following the basic approaches to formal languages design, the DICE profile will necessarily require a meta-modelling notation to cover for the basic structure and semantics of the language intended behind the DICE profile. Also, the DICE profile will need the implementation of said basic structure and semantics following a commonly usable format as best fit with respect to DICE goals and tenets.	D1.2	YES	DDSM is structured to be modularised into: (a) a TOSCA-specific level which allows straightforward rollout; (b) a complementary generic cloud infrastructure modelling level which allows for generalisability; (c) a concrete UML profile that allows for straigthforward concrete modelling and design;
PR2.1	The DICE profile MUST follow the default abstraction layers known and supported in Model-Driven Engineering, namely, Platform-Independent Model, Platform-Specific Model and add an additional layer specific to supporting the modelling of Deployment-ready implementations, i.e., a Deployment-Specific Model.	D1.2	YES	DDSM responds to this requirement providing its specification in the form a deployment-specific meta-modelling notation in the form of a stand-alone DSL as well as a concrete UML syntax and its associated OCL constraints.
PR2.11	The DICE Profile shall use packages to separately tackle the description of targeted technologies in the respective profile abstraction layers (e.g., DTSM and DDSM). Said packages shall be maintained consistent.	D1.2	YES	the DDSM layer addresses separation of concerns by isolating the concerns connected to continuous deployment to a single UML deployment diagram.
PR2.12b	The DICE Profile shall focus on DIA-specific privacy and/or security restrictions.	D1.2	NO	the DDSM layer is connected to security-by-design addressed in DICE but the support to secure deployment is, at this stage of its completion, still under investigation for final inclusion in the D2.5 deliverable concerning the DICE methodology overview.
PR2.18	The DICE IDE needs to be provided with a fully automated transformation that is capable of constructing an ad-hoc TOSCA blueprint stemming from the deployment information that can be made available in a DTSM and DDSM model. The usage of deployment knowledge for each technology in the DTSM shall be used by such transformation as a means to determine the deployment structure. Subsequently, a DDSM model proposal shall be built from this automated understanding. Finally, a TOSCA blueprint shall be constructed from such DDSM model using an appropriate mirroring between the DDSM model instance and the TOSCA notation.	D1.2	YES	the DICER tool allows to fully prepare and deploy in an automated fashion a UML deployment diagram stereotyped with DDSM constructs; the DICER tool includes a batch of sequential m2m transformations that allow this requirement to be fullfilled to a whole.
PR2.20	The DICE IDE needs to be rigged with a M2M transformation that provides coherent and comparable aggregates of the elements in the DICE technological library such as to allow for architecture trade-off analysis specified in PR2.19.	D1.2	YES	the DICER tool allows designers to explode their requirements into multiple UML deployment diagrams and deploy them concurrently using DICER 1-click deploy. By means of this feature, comparative deployment times as well as trade-off analysis may be carried off in a semi-automated, tool-supported fashion.
MR2.2	Every abstraction layer (namely, DPIM, DTSM and DDSM) of the DICE profile shall stem from UML, wherever possible.	D1.2	YES	the DICER tool and its automations stem and rest heavily on UML - the requriement is fully addressed by this feature.
MR2.12	The DDSM layer shall support the definition of an Actionable deployment view (TOSCA-ready): this view offers conceptual mappings between the technological layer defined in the DTSM and concepts in the TOSCA metamodeling infrastructure such that one-way transformation between the technological layer and the actionable deployment view is possible.	D1.2	YES	the UML DICER tool and the DICER technology library addressed in WP5 fully address this requriement by providing a solid and state-of-the-art TOSCA baseline for deployment with Cloudify.
MR2.13	The DDSM layer shall support the definition of framework overrides. This allows designers to provide ad-hoc tweaks to framework settings based on specific constraints or design concerns.	D1.2	YES	the DDSM layer provides features and facilities such that designers may configure in their own desired configuration every parameter - this feature set is made transparent for those who may not wish to see these configurations or wish to assume defaults. DICER rollout technology assumes default rollout any time no ad-hoc configuration is provided.
MR2.14	The DDSM layer shall support the management of VMs and similar resources as well as the necessary environmental setup connected to the application of specific frameworks (e.g., Hadoop/MapReduce).	D1.2	YES	The modelling and configuration of physical resources is addressed in DICER by means of the MODACloudsML language we extended as part of the work in DICE.
MR2.15	The DDSM layer shall allow the support for linking ad-hoc config. scripts or default config. scripts within the DIA.	D1.2	YES	We addressed this feature extending the MODACloudsML notation as well as the TOSCA library baseline to support the modelling of generic nodes with which users can model: (a) ad-hoc scripts or artefacts necessary for deployment; (b) generic user nodes such as the application node itself.
MR2.16	The Actionable Deployment View within the DDSM layer shall support DIA application bundling, e.g., using the CSAR formalism adopted by the TOSCA notation.	D1.2	NO	the CSAR file archive format is not supported yet by Cloudify Deployment technology - by inheritance, we had to constrain our DICER technology to address this limitation.
PRD2.1	The DICE Profile shall define deployment-specific construct contiguously to TOSCA-specific constructs and their relations.	D1.2	YES	the DICE profile now includes a concrete sub-area to address DDSM modelling. This has been detailed in the scope of the technical content of this deliverable.
PRD2.2	The DICE Profile shall define technology-specific properties in terms of required- and provided-properties.	D1.2	NO	the DICER tool and connected modelling technologies cannot support the TOSCA-based provided- and required-properties semantics since Cloudify orchestration does not reason in such terms. Consequently, our tool-support was designed to override these limitations with a more generic approach.
PRD2.3	The DICE Profile shall define annotations support the specification of required- and provided-execution platforms for the deployment of DIAs.	D1.2	YES	the DICER tool and its core modelling abstractions do provide the ability to model multiple execution platforms onto the same deployment diagram. However, for the scope of rollout each (micro-)service in the diagram will need to be rigged for execution by means of a single platform for consistency with Cloudify automated-deployment constraints.
PRD2.4	The DICE Profile shall provide facilities to model virtualized network-functions and their respective relations in an NFV topology.	D1.2	NO	Although an intriguing extension, we workout of DICER-based NFV facilities was judged out of scope for EU H2020 DICE.

References

Vesset, Dan et al. (2012). “Worldwide Big Data Technology and Services 2012?2015 Forecast”. In: *IDC Technical Report for EU H2020* 1, pp. 1–34.

Bersani, Marcello M. et al. (2016). “Continuous Architecting of Stream-Based Systems”. In: *Proceedings of the 25th IFIP / IEEE Working Conference on Software Architectures*. Ed. by Henry Muccini and Eric K. Harper. IEEE Computer Society, pp. 131–142.

Schmidt, Douglas C. (2006). “Model Driven Engineering”. In: *IEEE Computer* 39.2, pp. 25–31.

Hammersley, Martin and Paul Atkinson (2003). *Ethnography*. London: Routledge.

Rajbhoj, A., V. Kulkarni, and N. Bellarykar (2014). “Early Experience with Model-Driven Development of MapReduce Based Big Data Application”. In: *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*. Vol. 1, pp. 94–97. DOI: 10.1109/APSEC.2014.23.

Santurkar, S., A. Arora, and K. Chandrasekaran (2014). “Stormgen - A Domain specific Language to create ad-hoc Storm Topologies”. In: *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, pp. 1621–1628. DOI: 10.15439/2014F278.

Ferry, Nicolas et al. (2014). “CloudMF: Applying MDE to Tame the Complexity of Managing Multi-Cloud Applications”. In: *Proceedings of IEEE/ACM UCC 2014*.

Guerriero, Michele et al. (2016). “"Towards A Model-Driven Design Tool for Big Data Architectures"”. In:

Appendix A.1 DDSM Metamodels

A.1 The `DiceDomainModel::DDSM` metamodel

Table 2: `ddsm` data types

Name	Kind	Values or Description
`VMSize`	Enumeration	`Small`, `Medium`, `Large`
`ProviderType`	DataType	`fco`, Allows to set provider-specific information, such as the access token or the provided ssh key.
`LanguageType`	Enumeration	`bash`, `python`, `R` `java`, `scala`
`Scheduling`	Enumeration	`capacity`, `fair`, `fifo`
`FirewallRule`	DataType	Allows to configure firewall rules of a DdsmInternalComponent

Table 3: The `ddsm` package

DICE `ddsm` Metamodel Element	Description	Attributes	DDSM Profile Mapping
`CloudElement`	Abstract class, inherit from MODAClaudsML which capture common concepts shared by most of the classes specified in metamodel. For example a class extending CloudElement can have Properties and Resouces associated.	1. Attributes: • `cloudElementId`: `String` 2. Compositions: • `resource`: `Resource` • `property`: `Property`	No mapping needed as the standard UML modeling infrastructure provides enough abstractions to cover the role of the CloudElement class.
`Property`	Represents a generic property, specified by a pair of <id,value> and owned by a CloudElement.	1. Attributes: • `value`: `String` • `propertyId`: `String`	`UML::Property`
`Resource`	Represents a resource associated to an element which might be used to support the deployment and configuration of the such element. For instance a Resource may detail the deployment script of a CloudElement (e.g. an InternalComponent or an ExecutionBinding).	1. Attributes: • `name`: `String` • `resourceId`: `String` • `value`: `String`	`UML::Artifact`

`Component` inherits from: `ddsm::CloudElement`	Inherit from MODACloudsML, it represents a reusable type of cloud component (e.g. a virtual machine or an application).	1. Compositions: • `providedport`: `ProvidedPort` • `providedexecutionplatform`: `ProvidedExecutionPlatform`	`DdsmComponent` streotype; extends UML::Node metaclass.
`InternalComponent` inherits from: `ddsm::CloudElement`, `ddsm::Component`	Inherit from MODACloudsML, this represents a component that is managed by the application provider, or the developer (e.g. a Big Data application).	1. Compositions: • `requiredport`: `RequiredPort` • `internalcomponent`: `InternalComponent` • `requiredexecutionplatform`: `RequiredExecutionPlatform`	`DdsmInternalComponent` stereotype; inherits from: `DdsmComponent`
`ExecutionPlatform` inherits from: `ddsm::CloudElement`	Inherited from MODACloudsML, it represents an generic hosting interface of a Component.		See the ProvidedExecutionPlatform and RequiredExecutionPlatform subclasses.
`Port` inherits from: `ddsm::CloudElement`	Represents an interface (provided or required) of a Component. It is tipically used to link components in order to enable communication.		See the ProvidedPort and RequiredPort subclasses.
`RequiredPort` inherits from: `ddsm::CloudElement`, `ddsm::Port`	A specific type of Port which specify that a Component requires to communicate and consume a features (e.g.access to a database) provided by another Component.		Squashed into the mapping of the Relationship class
`ProvidedPort` inherits from: `ddsm::CloudElement`, `ddsm::Port`	A specific type of Port which specify that a Component provides features (e.g.access to a database) which can be accessed by another Component.		Squashed into the mapping of the Relationship class
`RequiredExecutionPlatform` inherits from: `ddsm::CloudElement`, `ddsm::ExecutionPlatform`	A specific type of ExecutionPlatform providing hosting facilities (e.g. an execution environment, like a VM or a web server) to a Component.		Squashed into the mapping of the ExecutionPlatform class
`ProvidedExecutionPlatform` inherits from: `ddsm::CloudElement`, `ddsm::ExecutionPlatform`	A specific type of ExecutionPlatform which requires hosting (e.g. a Big Data application requires a Big Data platform) from a Component.		Squashed into the mapping of the ExecutionPlatform class

Relationship inherits from: ddsm::CloudElement	Represents a generic relationship that binds a RequiredPort of a Component with a ProvidedPort of another Component. Specializations, contextual properties or constraints are supposed to provide specific semantics to a Relationship.	1. Attributes: • name: String • relationshipId: String 2. Associations: • providedport: ProvidedPort • requiredport: RequiredPort	Depending on the specific semantics Relationship is mapped to various types of UML::Association, such as UML::Dependency, UML::Deployment or UML::ComunicationPath.
ExecutionBinding inherits from: ddsm::CloudElement	Represents a binding between a RequiredExecutionPlatform and a ProvidedExecutionPlatform of two components, meaning that the first component will be hosted on the second one according to the specified binding.	1. Attributes: • name: String • bindingId: String 2. Associations: • requiredexecutionplatform: RequiredExecutionPlatform • providedexecutionplatform: ProvidedExecutionPlatform	Mapped to the nestedNode association on UML::ExecutionEnvironment.
ExternalComponent inherits from: ddsm::CloudElement, ddsm::Component	Inherit from MODACloudsML, this represents a component that is managed by an external provider, for instance a AWS EC2 virtual machine.	1. Associations: • provider: Provider	**DdsmExternalComponent** stereotype; inherits from: DdsmComponent
Provider inherits from: ddsm::CloudElement	Represents a generic provider of Clouds services.	1. Attributes: • credentialsPath: String	Mapped to the ProviderType enumeration to be set on DdsmExternalComponent.

`VM` inherits from: `ddsm::CloudElement`, `ddsm::Component`, `ddsm::ExternalComponent`	It is an specific ExternalComponent representing the well know concept of virtual machine. It is possible to the size of the VM in terms of RAM and CPU and Storage size ranges, the preferred operating system, the enabled ports, the desired public address and the number of executing instances, or the replication factor. It as been customized in the context of DICE to be able to specify DICE specific type of VM.	1. Attributes: • `is64os`: `String` • `imageId`: `String` • `maxCores`: `String` • `maxRam`: `String` • `maxStorage`: `String` • `minCores`: `String` • `minRam`: `String` • `minStorage`: `String` • `os`: `String` • `privateKey`: `String` • `providerSpecificTypeName`: `String` • `securityGroup`: `String` • `sshKey`: `String` • `publicAddress`: `String` • `instances`: `String` • `genericSize`: `VMSize` • `location`: `String`	Mapped to the VMsCluster stereotype; inheriths from: DdsmExternalComponent, DdsmComponent; extends UML::Device metaclass.
`DDSM`	Represents the root of a DDSM model.	1. Attributes: • `name`: `String` • `modelId`: `String` 2. Compositions: • `cloudelement`: `CloudElement`	Mapped to UML::Model with applied the DICE::DDSM profile.
`MasterSlavePlatform` inherits from: `ddsm::CloudElement`, `ddsm::Component`, `ddsm::InternalComponent`	Abstract class representing a generic master-slave architecture which is employed by many distributed platforms for data-intensive computing.	1. Attributes: • `name`: `String` • `modelId`: `String` 2. Compositions: • `cloudelement`: `CloudElement`	Mapped to the DdsmMasterSlavePlatform stereotype; inherits from DdsmInternalComponent.
`PeerToPeerPlatform` inherits from: `ddsm::CloudElement`, `ddsm::Component`, `ddsm::InternalComponent`	Abstract class representing a generic peer-to-peer architecture which is employed by many distributed platforms for data-intensive computing.	1. Attributes: • `name`: `String` • `modelId`: `String` 2. Compositions: • `cloudelement`: `CloudElement`	Mapped to the DdsmPeerToPeerPlatform stereotype; inherits from DdsmInternalComponent.

StormCluster inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Represents a Storm cluster, composed of a master node (nimbus) and multiple worker nodes (supervisor). Allows to define and edit the structure and the configuration of the cluster.	1. Attributes: • taskTimeout: Integer • supervisorFrequency: Integer • queueSize: Integer • monitorFrequency: Integer • retryTimes: Integer • retryInterval: Integer • workerStartTimeout: Integer • minStorage: Integer • cpuCapacity: Integer • memoryCapacity: Integer • heartbeatFrequency: Integer	Mapped to the DdsmStormCluster stereotype; inherits from DdsmMasterSlavePlatform; extends the UML::ExecutionEnvironment metaclass.

SparkCluster inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Represents a Spark cluster, composed of a master node and multiple worker nodes. Allows to define and edit the structure and the configuration of the cluster.	1. Attributes: • executorLogsMaxRetained: Integer • executorLogsRollingEnableCompression: Integer • executorLogsRollingMaxSize: Integer • reducerMaxSizeInFlight: Integer • reducerMaxReqsInFlight: String • shuffleCompress: Boolean • ioEncryptionEnabled: Boolean • eventLogEnabled: Boolean • uiEnabled: Boolean • uiKillEnabled: Boolean • uiPort: Integer • executorCores: Integer • defaultParallelism: Integer • executorHeartbeatInterval: Integer • defaultParallelism: Integer • filesMaxPartitionBytes: Integer • maxRequestedCores: Integer • schedulerMode: Scheduling • dynamicAllocationEnabled: Boolean • aclsEnabled: Boolean • authenticateEnabled: Boolean • sslEnabled: Boolean	Mapped to the DdsmSparkCluster stereotype; inherits from DdsmMasterSlavePlatform; extends the UML::ExecutionEnvironment metaclass.
ZookeeperCluster inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Represents a Zookeeper cluster, composed of a number of peer nodes. Allows to define and edit the structure and the configuration of the cluster.	1. Attributes: • tickTime: Integer • synchLimit: Integer • initiLimit: Integer	Mapped to the DdsmZookeeperCluster stereotype; inherits from DdsmPeerToPeerPlatform;

KafkaCluster inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Represents a Kafka cluster, composed of a number of broker nodes. Allows to define and edit the structure and the configuration of the cluster.	1. Attributes: • enableTopicDeletion: Boolean • numNetworkThreads: Integer • numIoThreads: Integer • numLogPartitionPerTopic: Integer • numRecoveryThreadsPerDataDir: Integer • numMessagesForFlushToDisk: Integer • maxMessageSittimebeforeflushms: Integer • enableTopicAutoCreation: Boolean • enableLeaderRebalancing: Boolean • logRetentionHours: Integer • queuedMaxRequests: Integer	Mapped to the KafkaCluster stereotype; inherits from DdsmPeerToPeerPlatform; extends the UML::ExecutionEnvironment metaclass.
ClientNode inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Plays the role of a client submitting a data-intensive job to the running Big Data infrastructure. It specifies properties on how to submit the job, along with details about the job itself by means of the association with JobSubmission.	1. Attributes: • skipRunningJob: String • numberOfSubmissions: String • schedule: Crontab 2. Associations: • submits: JobSumission	Mapped to the DdsmJobSubmission stereotype; extends the UML::Deployment metaclass.
JobSubmission inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Models a data-intensive job and allows to specify its deployment-specific information, such as the location of the executable artifact or required enviroment variables.	1. Attributes: • arguments: List<String> • artifactUrl: String • mainClass: String	Mapped to the BigDataJob stereotype; extends the UML::aRTIFACT metaclass.
YarnCluster inherits from: ddsm::CloudElement, ddsm::Component, ddsm::InternalComponent	Represents a Hadoop YARN cluster, composed of a master node (ResourceManager) and multiple slave nodes (NodeManager). Allows to define and edit the structure and the configuration of the cluster.	1. Attributes: • enableAcl: Boolean • schedulerType: Scheduling • schedulerMinContainerMemMb: Integer • schedulerMaxContainerMemMb: Integer • schedulerminContainerCoreNum: Integer • schedulerMaxContainerCoreNum: Integer • nodemanagerAvailableMem: Integer	Mapped to the DdsmYarnCluster stereotype; inherits from DdsmMasterSlavePlatform; extends the UML::ExecutionEnvironment metaclass.

`HdfsCluster` inherits from: `ddsm::CloudElement`, `ddsm::Component`, `ddsm::InternalComponent`	Represents a HDFS cluster, composed of a master node (NameNode) and multiple slave nodes (DataNode). Allows to define and edit the structure and the configuration of the cluster.	1. Attributes: • `dfsBlocksize`: `Integer` • `namenodeHandlerCount`: `Integer` • `datanodeHandlerCount`: `Integer` • `namenodeHeartbeatRecheckIntervalMs`: `Integer` • `permissionEnabled`: `Boolean` • `blockReplication`: `Integer` • `blocksizeBytes`: `Integer` • `blockWriteRetries`: `Integer` • `resourceNanagerRecoveryEnabled`: `Boolean`	Mapped to the DdsmHdfsCluster stereotype; inherits from DdsmMasterSlavePlatform.

Appendix A.2 TOSCA Metamodels

A.1 The `DiceDomainModel::TOSCA` metamodel

Table 4: The `tosca` package

DICE `tosca` Metamodel Element	Description	Attributes
`NodeTemplate`	A Node Template specifies the occurrence of a manageable software component as part of an application's topology model. A Node template is an instance of a specified Node Type and can provide customized properties, constraints or operations which override the defaults provided by its Node Type and its implementations. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `node_template_name: String` • `type: String` • `description: String` 2. Compositions: • `interfaces: Interface` • `properties: Property` • `attributes: Attribute` • `requirements: Requirement` • `relationships: Relationship` • `capabilities: Capability` 3. Associations: • `interfaces: Interface` • `properties: Property` • `attributes: Attribute` • `requirements: Requirement` • `relationships: Relationship` • `capabilities: Capability`
`Interface`	An interface defines a named interface that can be associated with a Node or Relationship Type. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `name: String` 2. Compositions: • `operations: Operation` • `inputs: Input`
`Relationship`	A Relationship Template specifies the occurrence of a manageable relationship between node templates as part of an application's topology model. A Relationship template is an instance of a specified Relationship Type . For the accurate description refer to the TOSCA standard document [**toscayaml**].	1. Attributes: • `type: CloudifyRelationshipType` 2. Associations: • `interfaces: Interface` • `properties: Property` • `attributes: Attribute`

`Requirement`	A Requirement describes a dependency of a TOSCA Node Type or Node template which needs to be fulfilled by a matching Capability declared by another TOSCA modelable entity. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `name`: `String` • `node`: `String` • `capabiity`: `CloudifyCapabilityType`
`Operation`	An operation defines a named function or procedure that can be bound to an implementation artifact (e.g., a script). For the accurate description refer to the TOSCA standard document [**toscayaml**].	1. Attributes: • `operation_name`: `String` • `description`: `String` • `script`: `String` • `executor`: `String` 2. Compositions: • `operation_hasInput`: `Input`
`TopologyTemplate`	A Topology Template defines the topology of a cloud application. The main ingredients of the topology template are node templates representing components of the application and relationship templates representing links between the components. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `tosca_definitions_version`: `String` 2. Compositions: • `imports`: `Import` • `outputs`: `Output` • `inputs`: `Input` • `nodeTemplates`: `NodeTemplate` • `realtionships`: `Relationship` • `groups`: `Group` • `policies`: `Policy` 3. Associations: • `imports`: `Import` • `outputs`: `Output` • `inputs`: `Input` • `nodeTemplates`: `NodeTemplate` • `realtionships`: `Relationship` • `groups`: `Group` • `policies`: `Policy`

`Import`	An import is used to locate and uniquely name another TOSCA file which has type and template definitions to be imported (included) and referenced. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `import_name`: `String` • `file`: `String` • `repository`: `String` • `namespace_uri`: `String` • `namespace_prefix`: `String`
`Group`	A group definition defines a logical grouping of node templates, typically for management purposes. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `name`: `String` • `type`: `CloudifyGroupType` • `description`: `String` • `targets`: `String` 2. Associations: • `properties`: `Property` • `interfaces`: `Interface`
`Policy`	A policy definition defines a policy that can be associated with a TOSCA topology. For the accurate description refer to the TOSCA standard document [**toscayaml**]	
`Capability`	A Capability defines a named, typed set of data that can be associated with Node Type or Node Template to describe a transparent capability or feature of the software component the node describes. For the accurate description refer to the TOSCA standard document [**toscayaml**]	1. Attributes: • `type`: `CloudifyCapabilityType` • `description`: `String` 2. Associations: • `properties`: `Property` • `attributes`: `Attribute`