Infrastructure-as-Code for Data-Intensive Architectures: A Model-Driven Development Approach

Matej Artač[2], Tadej Borovšak[2], Elisabetta Di Nitto[1], Michele Guerriero[1], Diego Perez Palacin[1], Damian Andrew Tamburri[1]

[1]DEEPSE group - DEIB - Politecnico di Milano, Milano, Italy
[2]XLAB - Ljubljana, Slovenia
matej.artac@xlab.si, tadej.borovsak@xlab.si, elisabetta.dinitto@polimi.it, michele.guerriero@polimi.it, diego.perez@polimi.it, damianandrew.tamburri@polimi.it

***Abstract*—As part of the DevOps tactics, Infrastructure-as-Code (IaC) provides the ability to create, configure, and manage infrastructures by means of executable code. Writing IaC, however, is not an easy task, since it requires blending different infrastructure programming languages and abstractions, each specialized on a particular aspect of infrastructure creation, configuration, and management. Moreover, the more the architectures become large and complex (e.g., think of Data-Intensive or Microservice-based architectures), the more dire the need of IaC becomes. The goal of this paper is to exploit Model-Driven Engineering (MDE) to create language-agnostic models that are then automatically transformed into IaC. We focus on the domain of Data-Intensive Applications as these typically exploit complex infrastructures which demand sophisticated and fine-grained (re-)configuration — we show that, through our approach (called DICER for anonymization purposes) it is possible to create complex IaC with significant amounts of time savings, both in IaC design as well as (re-)deployment times.**

***Keywords*-DevOps; Big Data; Model-Driven Engineering; Cloud; Infrastructure-as-Code; TOSCA;**

I. Introduction

The current IT market is increasingly dominated by the "need for speed". This need is reflected in the emerging use of agile and lean techniques which shorten the software development cycle and also intermix software development activities with IT operations. This trend of using software engineering tactics that reduce the space, time, and efforts between software development and operations, as well as the technical and organizational distance between these two types of software teams is known as DevOps [1]. As part of the DevOps menu, many practices entail re-using standard tools from software development (e.g., code-versioning, code-revision management, etc.) to manage what is known as the Infrastructure-As-Code (IaC) approach [2]. Following this approach, software code is developed to define and manage the whole technological stack used to run complex software architectures [2]. IaC is, in principle, very useful as it enables the automation of several deployment, configuration, and management tasks that otherwise would have to be performed manually by a specialised -Ops practitioner.

Unfortunately, despite the great potential of the available approaches, writing infrastructural code implies the usage of various different languages each one specialized on a particular aspect. For example. Software-Defined Networking exploits specific languages such as OpenFlow or NETCONF. Also, software configuration can be accomplished by writing Chef recipes or Puppet code, but, still, other aspects such us configuring the monitoring infrastructure, the load balancer or the self-adaptation mechanism require the usage of other languages and associated tools.

Such diversity implies that adopting the IaC approach today requires a variety of specialised skills that, in many cases, average companies do not have and struggle to find. For this reason, the huge automation potential offered by IaC remains under-utilised.

In this context, our goal is to experiment with Model-Driven Engineering to see if its power of abstraction and its automation potential can help simplifying IaC development.

In the scope of this paper, we focus specifically on creating IaC for Data-Intensive Architectures (DIAs), since DIAs require particularly complex infrastructure and management tasks and therefore represent a good playground for our experiments.

The specific contributions of this paper are as follows:

- A Domain-Specific Modeling Language (DSML) for DIAs deployment expressed in terms of an open source UML profile[1];
- A Library of IaC components each one contributing to the automation of the deployment, configuration, or management of the most well-known infrastructural components for DIAs;
- An open source tool called DICER[2] that, starting from an instance model of our DSML, generates the corresponding IaC by assembling the components we have previously mentioned;
- An assessment of the approach (using case-study research

[1]Link Removed for Anonymization Purposes
[2]Link Removed for Anonymization Purposes

[3]) against two real industrial use cases, which shows DRC significantly simplifies the design of DIAs and reduces their deployment time;

The remainder of this paper is organised as follows. Section 2 provides the necessary background on IaC and DIAs, Section 3 provides an overview of the approach, Section 4 focuses on the definition of the DICER UML profile, Section 5 presents our IaC library and the IaC generation realized by DICER, Section 6 presents the evaluation. Finally, Section 7 and 8 present related work and conclude the paper discussing the future work.

II. Background

A. Ops activities and IaC

Figure 1 shows the typical Ops activities required to continuously deploy and operate Cloud applications [4]:

- **Configuration management** is the process of deploying and managing at runtime all the required services (e.g. Tomcat, MySQL, Hadoop, Cassandra). It consumes reusable *recipes* that (often declaratively) describe how to manage and configure the service across lifecycle phases.
- **Server provisioning** entails acquiring (e.g. from a public provider of *Infrastructure-as-a-Service*), configuring and running the required VMs or containers upon which services can run.
- **Application deployment** is the phase in which user's applications are executed on the resulting infrastructure.
- **Monitoring** all the running component (VMs, services, middleware and applications) is a basic enabler for almost any kind of runtime management activity.
- **Self-adaptation** is about applying a set of methods (e.g. VMs autoscaling, faults recovery, etc) to keep the status conforming with service-level agreements and the quality of service (QoS) variables therein (e.g., security, network safety, etc.).

All these activities require specific languages and tools. Configuration management is typically handled by well-known tools like Chef, Puppet, Ansible, while service provisioning and application deployment are responsibilities of so called *Cloud orchestrators*. In this area, TOSCA is the "Topology and Orchestration Specification for Cloud Applications" and is the reference OASIS standard for orchestration languages, supported by over 60+ big industrial players such as IBM, Huawei and more [5]. Through this language, all needed resources and components of an application are described and connected to the corresponding configuration scripts. Even performing only two of the above activities requires expertise in two different domains and associated toolsets and, obviously, the complexity increases with increasing number of Ops activities that have to be carried out.

B. DIAs and DIA Technologies

A DIA is typically composed of multiple computational components that can be abstracted in terms of *DIA jobs*, acquiring data from one or more *data sources* and producing

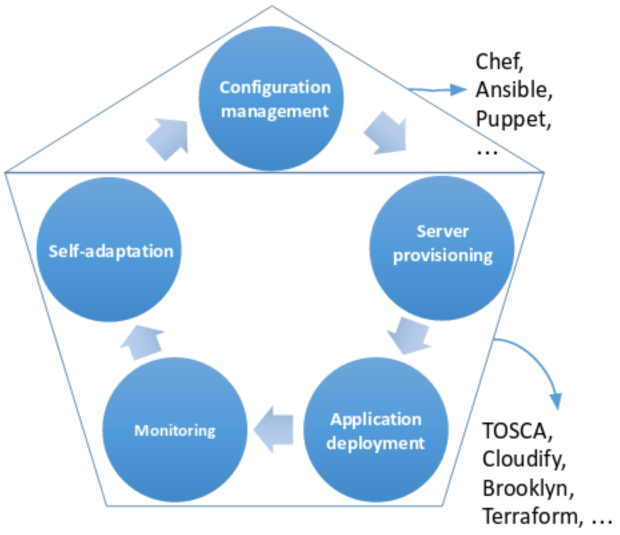


Fig. 1: Typical Ops Activities.

data to one or more *data sinks*. Data sources and data sinks can be of various kinds, for instance, GUIs, files, databases, streams, other systems. DIA jobs can work in *batch*, *stream* or *interactive mode* [6]. This has an impact on how data is acquired from sources (or produced to sinks) and on the individual component lifetime.

Several Big Data technologies exist that are used in the context of designing modern DIAs. More specifically, there are technologies supporting data storage (see, for instance, Cassandra[3], a reliable and distributed column-oriented NoSQL database, or HDFS[4], a distributed file system) or message queuing (see for instance, Kafka[5], a framework for creating data pipelines) or *execution environments* that support applications developers with specific distributed and parallel programming paradigms.

Among the other execution environments, Apache Hadoop[6] is the most known distributed implementation of the MapReduce programming model. Hadoop supports a "batch" processing mode by allowing developers to define three main types of computational elements, namely *map*, *shuffle* and *reduce*. Apache Storm[7] has been the first attempt to enable distributed real-time data processing using a "stream" processing mode (i.e., the data is analysed as soon as it is available in a continuous fashion). Besides of Hadoop and Storm, many other engines have been developed over the last few years (e.g. Apache Spark and Apache Flink) that provide more and more improvements in the way large scale data processing is done.

These technologies are typically distributed systems running on top of *clusters* of computational resources, e.g. *virtual*

[3]http://cassandra.apache.org
[4]https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
[5]https://kafka.apache.org
[6]http://hadoop.apache.org
[7]http://storm.apache.org/index.html

machines (VMs) within a Cloud infrastructure. For instance, Hadoop MapReduce applications run in an Hadoop cluster, which consists of the Hadoop Distributed File System (HDFS), which is the underlying scalable data storage, and computational resources manager such as YARN (Yet Another Resource Negotiator). HDFS follows a *master/slaves* architecture composed of a NameNode (master), which is responsible for managing namespaces metadata and file access requests from clients, plus an arbitrary number of DataNodes (slaves), whose role is to store data partitions. YARN also adopts a master/slaves architecture, in which the ResourceManager (master) keeps track of the available resources in the cluster and allocates them to running MapReduce applications, while several NodeManagers (slaves), running on different physical nodes, execute the various tasks (map, reduce) of the applications. Finally, a Hadoop deployment can optionally include Apache Zookeeper — a system which handles coordination and synchronisation of distributed systems — to achieve high availability of the single points of failure represented by the NameNode and the ResourceManager. Zookeeper is a distributed system, based on a *peer-to-peer* architecture in which each peer has the same role and responsibilities.

Given their complexity, Big Data technologies often have hundreds of configuration parameters, "knobs" that tune overall system behaviour. It is therefore necessary to try multiple, different deployments of the same DIA to find the optimal configuration. Furthermore, even a reasonably simple DIA potentially needs many Big Data technologies to fully operate. For example, "Lambda" architectures [7] are a widely adopted architecture style for DIAs and require: a) a messaging system to decouple the data-intensive jobs from the various data sources, b) a stream processing engine, for real-time processing over recent data, c) a batch processing engine for periodical batch jobs over historical data, d) at least a database for storing/processing results. The resulting (re-)deployment and (re-)configuration process can easily become very complex and time-consuming.

III. The DCR Approach

From the set of tools and technologies presented in the previous section, the reader should have got the idea that: i) IaC actually represents a large variety of approaches and tools each one specialised on a specific aspect of the Ops pipeline, ii) deploying, configuring and operating a DIA is quite complex, especially when we want to take advantage of and combine the features offered by more than one framework.

DICER aims at fulfilling the following technical requirements for DevOps teams:

- **REQ1**: *to develop IaC for DIAs in *continuity* with development of the actual software*. DICER shall provide DevOps teams with a tool that unifies the abstractions that are relevant in the IaC context with those that are relevant in classical software design.
- **REQ2**: *to enable *fast* deployment, testing, and re-deployment of data-intensive architectures through IaC*. DICER shall provide DevOps teams with means to shorten the time needed for writing and evolving IaC for DIAs, by automating all steps needed to produce deployable IaC.
- **REQ3**: *to allow the reuse of pre-existing IaC assets*, without being forced to know their implementation.
- **REQ4**: *to enable consistency checking of IaC designs*. Building infrastructural code means working at many different levels of abstraction (physical and virtual machines, network channels, software components, monitoring infrastructures). Proper consistency checking is thus very important as it can fasten the IaC development and deployment process.

The DICER approach is centered around the assumption that the IaC designer can work in continuity with the more traditional design activities, exploiting modeling features to specify the required DIA. Regarding the ops activities shown in Figure 1, we are currently focusing on of the configuration management, service provisioning and application deployment.

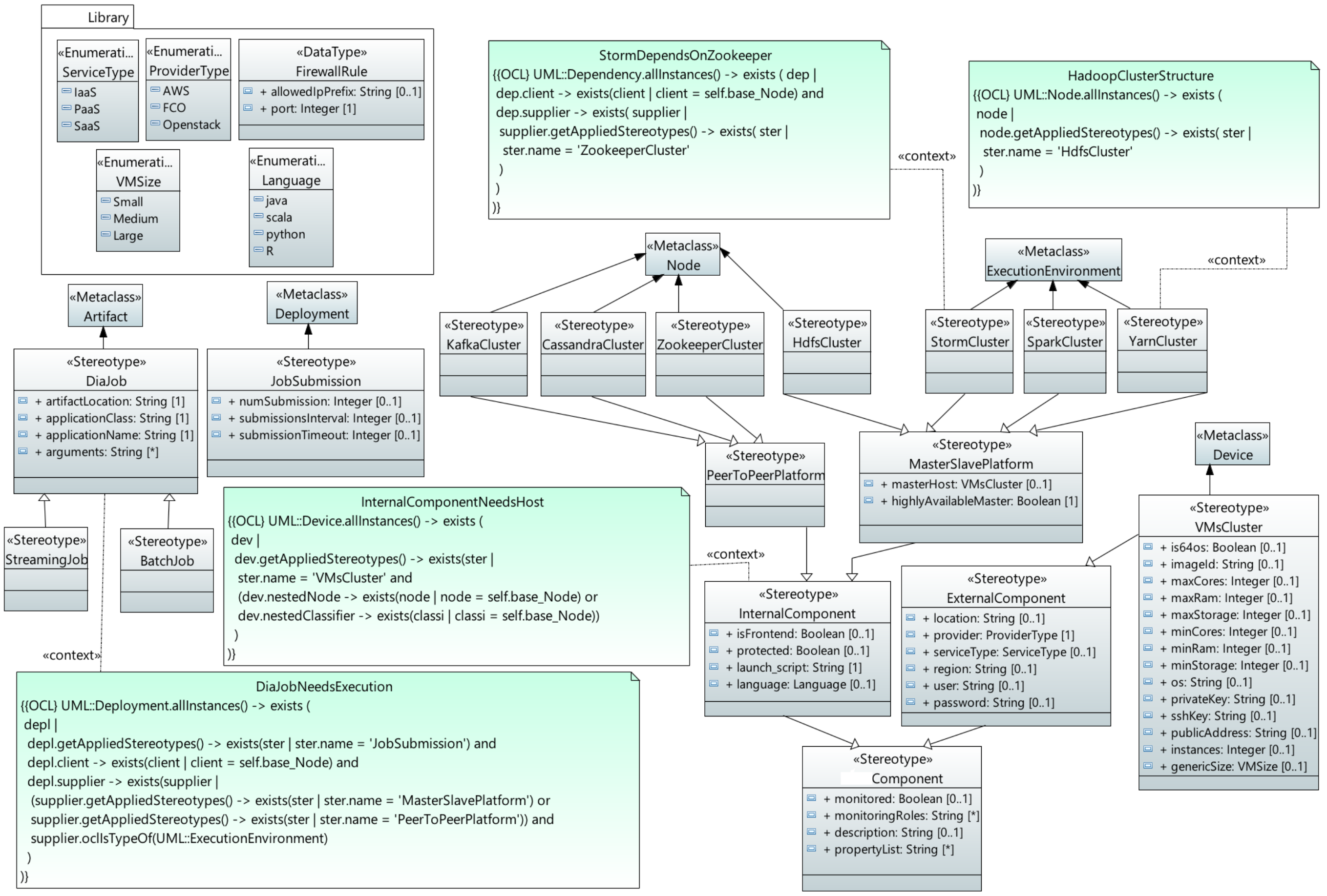
Figures 3, 4 and 5 show, using stereotyped UML deployment diagrams, different configurations of the Big Data frameworks exploited by a toy example DIA called WikiStats[8]. As evident, each model includes information relevant to create clusters of virtual machines running the needed software components and provides, at a glance, a clear view on the organisation of such machines and components. To be actually executed, the models have to be transformed into executable code (see Listing 2 for an excerpt). Implemented as an extension to the Eclipse IDE[9], DICER is in charge of generating that code from the high level models, thus giving to IaC designers the possibility to work with more intuitive DIA and IaC abstractions directly in a classical IDE. The generated code can be then immediately deployed using a suitable Cloud orchestrator.

IV. UML Profile for DIA infrastructure deployment

The process of defining the DICER UML profile begun with an in depth study of the Big Data technological landscape in order to understand the main characteristics and commonalities of DIAs. As the foundation of our profile, we relied on CloudML [8], a modeling language for multi-Cloud applications, that we extended adding DIA-specific modeling concepts, such as those defined in Section II. We defined the DICER UML profile (implemented in Eclipse UML Papyrus[10]) to be applied to UML deployment diagrams[11], which are the natural choice when modeling the deployment of software architectures. A relevant part of DICER profile is shown in Figure 2, which also reports various constraints, written using the Object Constraint Language (OCL), that are to automatically validate the consistency of models.

First, a *Component* can be *InternalComponent*, representing any piece of software that is managed and deployed by DICER

[8]The details of WikiStats models will be clarified further on in the paper.
[9]www.eclipse.org
[10]https://www.eclipse.org/papyrus/
[11]http://www.uml-diagrams.org/deployment-diagrams.html


Library
«Enumerati... ServiceType
IaaS
PaaS
SaaS
«Enumerati... ProviderType
AWS
FCO
Openstack
«DataType» FirewallRule
+ allowedIpPrefix: String [0..1]
+ port: Integer [1]
«Enumerati... VMSize
Small
Medium
Large
«Enumerati... Language
java
scala
python
R
«Metaclass» Artifact
«Metaclass» Deployment
«Stereotype» DiaJob
+ artifactLocation: String [1]
+ applicationClass: String [1]
+ applicationName: String [1]
+ arguments: String [*]
«Stereotype» JobSubmission
+ numSubmission: Integer [0..1]
+ submissionsInterval: Integer [0..1]
+ submissionTimeout: Integer [0..1]
«Stereotype» StreamingJob
«Stereotype» BatchJob
«context»
«Metaclass» Node
«Stereotype» KafkaCluster
«Stereotype» CassandraCluster
«Stereotype» ZookeeperCluster
«Stereotype» HdfsCluster
«Metaclass» ExecutionEnvironment
«Stereotype» StormCluster
«Stereotype» SparkCluster
«Stereotype» YarnCluster
«context»
«context»
«Stereotype» PeerToPeerPlatform
«Stereotype» MasterSlavePlatform
+ masterHost: VMsCluster [0..1]
+ highlyAvailableMaster: Boolean [1]
«Metaclass» Device
«Stereotype» VMsCluster
+ is64os: Boolean [0..1]
+ imageId: String [0..1]
+ maxCores: Integer [0..1]
+ maxRam: Integer [0..1]
+ maxStorage: Integer [0..1]
+ minCores: Integer [0..1]
+ minRam: Integer [0..1]
+ minStorage: Integer [0..1]
+ os: String [0..1]
+ privateKey: String [0..1]
+ sshKey: String [0..1]
+ publicAddress: String [0..1]
+ instances: Integer [0..1]
+ genericSize: VMSize [0..1]
«context»
«Stereotype» InternalComponent
+ isFrontend: Boolean [0..1]
+ protected: Boolean [0..1]
+ launch_script: String [1]
+ language: Language [0..1]
«Stereotype» ExternalComponent
+ location: String [0..1]
+ provider: ProviderType [1]
+ serviceType: ServiceType [0..1]
+ region: String [0..1]
+ user: String [0..1]
+ password: String [0..1]
«Stereotype» Component
+ monitored: Boolean [0..1]
+ monitoringRoles: String [*]
+ description: String [0..1]
+ propertyList: String [*]

```
StormDependsOnZookeeper
{{OCL} UML::Dependency.allInstances() -> exists ( dep |
 dep.client -> exists(client | client = self.base_Node) and
 dep.supplier -> exists( supplier |
  supplier.getAppliedStereotypes() -> exists( ster |
   ster.name = 'ZookeeperCluster'
  )
 )
)}
```

```
HadoopClusterStructure
{{OCL} UML::Node.allInstances() -> exists (
 node |
  node.getAppliedStereotypes() -> exists( ster |
   ster.name = 'HdfsCluster'
  )
)}
```

```
InternalComponentNeedsHost
{{OCL} UML::Device.allInstances() -> exists (
 dev |
  dev.getAppliedStereotypes() -> exists(ster |
   ster.name = 'VMsCluster' and
   (dev.nestedNode -> exists(node | node = self.base_Node) or
   dev.nestedClassifier -> exists(classi | classi = self.base_Node))
  )
)}
```

```
DiaJobNeedsExecution
{{OCL} UML::Deployment.allInstances() -> exists (
 depl |
  depl.getAppliedStereotypes() -> exists(ster | ster.name = 'JobSubmission') and
  depl.client -> exists(client | client = self.base_Node) and
  depl.supplier -> exists(supplier |
   (supplier.getAppliedStereotypes() -> exists(ster | ster.name = 'MasterSlavePlatform') or
   supplier.getAppliedStereotypes() -> exists(ster | ster.name = 'PeerToPeerPlatform')) and
   supplier.oclIsTypeOf(UML::ExecutionEnvironment)
  )
)}
```

Fig. 2: An extract of the DICER UML Profile for DIAs deployment.

users, and *ExternalComponent*, any hw/sw resource owned and managed by a third-party provider (see the *providerType* property of *ExternalComponent* stereotype).

The *VMsCluster* stereotype is defined as a specialization of *ExternalComponent*, since renting computational resources such as virtual machines is one of the main services offered by Cloud providers. *VMsCluster* is applied to the *Device* UML meta-class. A cluster of VMs logically represents a single computational resource with processing capabilities, upon which applications and services may be deployed for execution. It 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). There are offered two possible manners to define the size of VMs. The first is to use the *VMSize* enumeration, while the second is to specify lower and upper bounds for the VMs characteristics (e.g. minCore/maxCore, minRam/maxRam). If the user chooses the second manner for the VMs size definition, it is assumed that the employed Cloud orchestrator is then able to identify 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 inherent complexity of deploying distributed systems. The OCL constraint *InternalComponentNeedsHost* imposes that each *InternalComponent* must be contained into a *Device* holding the *VMsCluster* stereotype, since by definition an *InternalComponent* has to be deployed and managed by the application provider, who makes available the necessary hosting resources.

We then define two DIA-specific deployment abstractions, i.e. the *PeerToPeerPlatform* and *MasterSlavePlatform* abstract stereotypes, as further specialisations of *InternalComponent*. These two stereotypes allow the modeling language to capture the key differences between the two types of distributed architectures introduced in Section II-B. For instance, the *MasterSlavePlatform* stereotype allows indicating a dedicated host for the master node, which might require more resources.

By extending the *PeerToPeerPlatform* and *MasterSlavePlatform* stereotypes we implemented a set of technology-specific *concrete* stereotypes (*StormCluster*, *CassandraCluster*, etc.), one for each technology we support. DIA execution engines (e.g. Spark or Storm) extend UML *ExecutionEnvironment*, so to identify those platforms which DIA jobs can be submitted to. All the other technology-specific stereotypes, e.g., those relevant to represent databases, extend UML *Node*. Each of these stereotypes allows to model deployment aspects that are specific to a given technology, such as platform specific configuration parameters (omitted for the sake of space) and dependencies on other technologies, that are enforced by means of OCL constraints in the case they are mandatory. For instance the *StormDependsOnZookeeper* constraint on the *StormCluster* stereotype in Figure 2 requires a *StormCluster* to be connected with a *ZookeeperCluster* by means of a UML Dependency.

The *DiaJob* stereotype represents the actual application that can be submitted for execution to any of the available execution engines. It is defined as a specialisation of UML *Artifact*, since it corresponds to the DIA executable artifact (e.g. a Hadoop MapResuce or a Storm application). It allows to specify job-specific information, for instance the *artifactUrl* from which the application executable is retrieved. It is specialized by two concrete stereotypes, namely *StreamingJob* and *BatchJob*. The *JobSubmission* stereotype, which extends UML *Deployment*, is used to specify additional deployment options for a DIA job. For instance, it allows to specify job's 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. Additionally, an OCL constraint requires each *DiaJob* to be connected by means of *JobSubmission* to an execution engine, i.e. an *ExecutionEnvironment* holding a sub-stereotype of *MasterSlavePlatform* or *PeerToPeerPlatform*.

A. Modeling the Wikistats Running Example

In order to show the DICER profile at work, we present Wikistats, a simple example of a DIA that processes Wikimedia[12] articles to elicit statistics on their contents and structure. Wikistats exploits Apache Storm as a stream processing engine, having Wikimedia as a source, and uses Apache Cassandra for storage. A first possible deployment model for Wikistats is shown in Figure 3. In this specific scenario, all the necessary platforms are deployed within the same cluster of 3 large-sized VMs from an Openstack installation (*cluster1*). *stormWikistats* is the Storm application itself modeled as a specific *StreamingJob*. Both Cassandra and Zookeeper, which is needed by Storm, are modeled as Nodes (*cassandra1* and *zookeeper1*) annotated with a corresponding technology-specific stereotype, while Storm is modeled as an ExecutionEnvironment (*storm1*), since it is the engine that executes the Wikistats application code. The Dependency element between *storm1* and *zookeeper1* is enforced via the previously discussed OCL constraint. The Wikistats job is linked to the StormCluster element using a *Deployment* dependency stereotyped as a *JobSubmission*. *DiaJob* and *JobSubmission* can be used to elaborate details about the Wikistats job and how it is scheduled. Within the cluster, Storm will be deployed using 3 slave nodes, while for Cassandra and Zookeeper there will be 3 worker nodes each. The annotations on the diagram explicitly highlight the most important deployment and configuration parameters that we have set for this example.

[12]https://www.wikimedia.org/

At this point, fine tuning of the Cloud infrastructure and of the various platforms is the key feature supported by DICER. The technology stereotypes allow to configure each platform (see the annotations in Figure 3) in such a way to easily and quickly generate and then test different configurations over multiple deployments and enabling continuous architecting of DIAs. Let us assume the previous model does not satisfy certain architecture's requirements, for instance because by co-locating all the required Big Data platforms within the same cluster this becomes overloaded (e.g. the average CPU consumption of the cluster is over a desired threshold). One of the possible solutions can be to host the Cassandra cluster on a dedicated set of VMs. The resulting model is shown in Figure 4. The DICER user can easily derive it from the original one and generate the corresponding blueprint. Furthermore, let us consider the case that, during the development process, one may want to compare different implementations of the same application, using different execution engines. For instance, an alternative implementation of WikiStats could use the Spark framework. In this case, we could extend the deployment model into the one in Figure 5 where we have added a new VMCluster and a new ExecutionEnvironment element with the SparkCluster stereotype (*spark1*) as well as a Spark version of WikiStats (*sparkWikistats*) modeled identically to the Storm one.

V. From Models to Infrastructural Code

One of the key feature offered by DICER is the automatic translation of models written using the DICER profile into runnable IaC. As IaC languages we have selected TOSCA

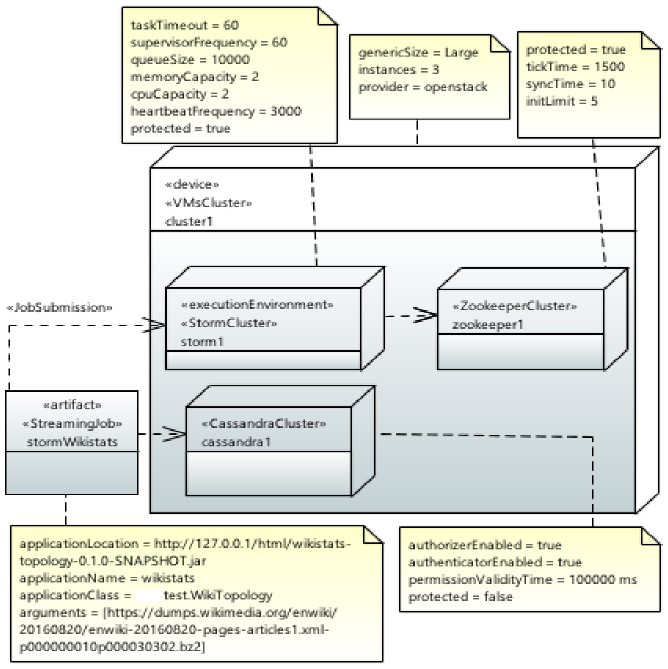


Fig. 3: Initial deployment diagram for the WikiStats DIA.

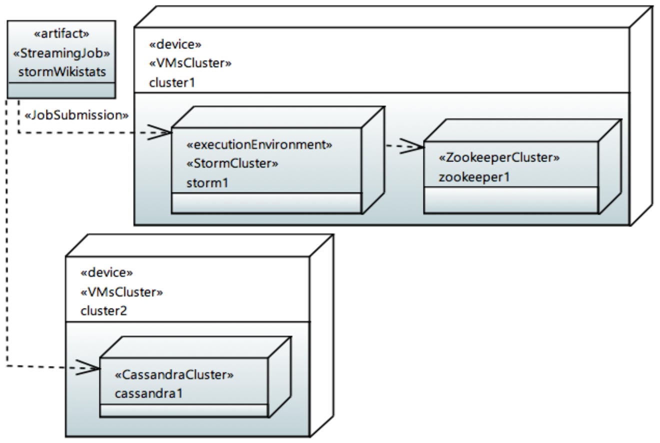


Fig. 4: First refinement of the deployment diagram for the WikiStats DIA: moving Cassandra to a dedicated cluster.

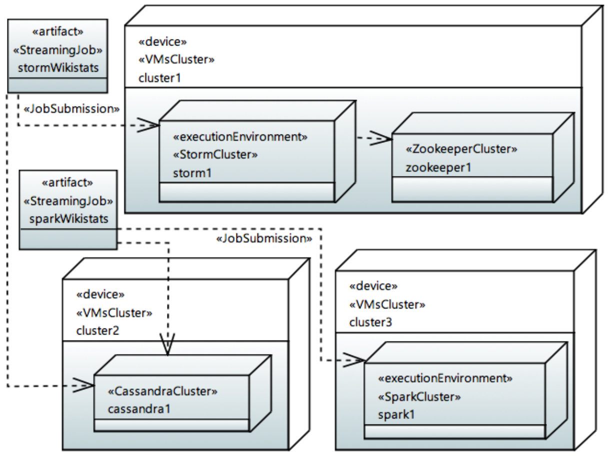


Fig. 5: Second refinement of the deployment diagram for the WikiStats DIA: deploying a Spark version of WikiStats.

and Chef, to support configuration management, service provisioning and application deployment.

TOSCA provides a flexible and highly extensible DSL for modeling resources and software components. TOSCA *blueprints* are executable IaC composed of *node templates* and *relationships*, defining the *topology* of a hardware/software systems. Node templates and relationships are instances of *node types* and *relationship types*, that are either normative (i.e., defined in the standard), provided by the specific engine that executes a blueprint (the orchestrator), or an extension of one of the above, such as in our case, with DIA-specific node and relationship types. Node types are essentially used to describe hardware or virtual resources (machines or VMs) and software components. Relationship types predicate on the association between node types. For instance, a TOSCA node type representing an executable component A must be associated to a node type representing a VM through the relationship *hosted_On*.

Each node type and relationship type also enables specifying *interfaces*, which are composed of operations that have to be carried out at specific stages of the deployment orchestration. Typical examples of interface operations include installing, configuring or starting of components, and may take various forms, for instance, Python/bash scripts, or pointers to Chef recipes. We have chosen to use these last ones because of reusability reasons. Chef, in fact, offers a rich set of *cookbooks* that include recipes for configuring a very large number of useful components.

To enable the automatic generation of infrastructural code from the modeling level, we have created a library of reusable infrastructural code that includes:

- The definition of TOSCA node and relationship types for representing all main components of data-intensive architectures.
- The defined or reused Chef recipes that perform the operations install, start, configure, stop, etc.
- Some Phyton code fragments that manage configuration parameter passing between components. For instance, each Storm Worker node has to own a reference to its corresponding Master node. However, such reference is only known when the Master node is launched. A code fragment, therefore, has to interact with the orchestrator executing the infrastructural code, acquire such information and pass it to the starting Worker node.

This library at the moment supports the deployment and complete configuration of many of the most widely used Big Data frameworks, namely Apache Hadoop, Apache Kafka, Apache Cassandra, Apache Spark, Apache Storm, MongoDB and Apache Zookeeper. It includes 92 TOSCA node and relationship types and 44 Chef recipes.

When a IaC designer completes the deployment diagram associated to his/her DIA, she lauches the generation of the infrastructural code. As a result, the TOSCA node and relationship types relevant to the input model are instantiated.

Listing 1 reports in pseudo-code an excerpt of the transformation workflow.

Listing 1: DICER transformation pseudo-code.

```
for(UML::Device dev in UML::Model) {
  if (dev.hasStereotype('VMsCluster')) {
    generateHostNodeTemplate(dev);
    for(UML::Node node in dev.nestedNodes) {
      stereotypes := node.getAppliedStereotypes();
      for (UML::Stereotype ster in stereotypes) {
        if(ster.isSubStereotypeOf('InternalComponent')
            ↪ and ster.getProperty('protected')) {
        generateFirewall(dev, node);
      }}}
```

This transforms each *VMsCluster* into a TOSCA node template as the one shown Listing 2, which is generated from the initial WikiStats model in Figure 3. The type of such template (Listing 2, line 2) depends on the *genericSize* property of the VMsCluster stereotype, if this is set. The number of VM instances (line 3) and the selected Cloud provider (line 7) correspond to those in Figure 3. For each *protected* component internal to the VMsCluster (see the *protected* property of the *InternalComponent* stereotype), the transformation generates a node template from the package *firewallrules.** (Listing 2, lines 7-10) and binds this with the generated *host.** node template using a *relationships.ProtectedBy* relation (Listing 2, lines 5 and 6). Similarly, the TOSCA library provides specific relationships to configure dependencies among Big Data technologies. For instance the Storm package provides the *component.storm.ConnectedToZookeeperQuorum* relationship, which allows to specify the connection of a Storm cluster with a Zookeeper cluster, on which Storm relies at runtime.

Listing 2: The TOSCA code generated for a VMsCluster stereotype.

```
1  cluster1:
2    type: hosts.Large
3    instances: {deploy: 3}
4    relationships:
5    - {type: relationships.ProtectedBy, target:
         ↪ storm_1_firewall}
6    - {type: relationships.ProtectedBy, target:
         ↪ zookeeper_1_firewall}
7    properties: {platform: openstack}
8  storm1_firewall:
9    type: firewall_rules.storm.Worker
10 zookeeper1_firewall:
11   type: firewall_rules.zookeeper.Server
12 ...
```

The transformation implemented in DICER ensures that all the elements of a model are properly instantiated into the corresponding IaC. In the case of the WikiStats example, the simplest deployment leads to the generation of 782 lines of infrastructural code, counting also the used Chef recipes.

When the infrastructural code is generated, it can then be executed by a proper Cloud orchestrator. We selected Cloudify[13] as TOSCA orchestrator as it is one of the most well-known and aligned to the TOSCA standard (other possible ones are ARIA TOSCA[14], Indigo[15], Apache Brooklyn[16] or ECoWare [9], [10]) and we have relied on its integration with the Chef framework and with a Python interpreter to execute the other parts of our code.

As the reader may have noticed, supporting the transition from a modeled DIA deployment into the corresponding IaC has required us to develop or reuse code written in three different languages (i.e. TOSCA, Chef and Python) with different programming models. Thanks to DICER, users can perform this transition automatically. The limits of the current implementation concern the extendibility of the approach to include frameworks that at the moment we did not consider. As discussed in Section VIII, this is certainly an important and complex issue that we plan to tackle in the forthcoming months.

[13]http://getcloudify.org

[14]http://ariatosca.org/

[15]https://www.indigo-datacloud.eu/

[16]https://brooklyn.apache.org/learnmore/

VI. Evaluation

This section evaluates the DICER approach. The aim of the evaluation is the following:

- Check that the modeling approach allows us to define executable deployment models using and combining the Big Data frameworks we have coded in DICER. By executable deployment models we mean models that we are able to translate into deployable IaC.
- Check that the time needed for the transformation into IaC and for its subsequent deployment is reasonable, to enable DICER users to smoothly follow the deployment, testing, and redeployment cycle.
- Check that DICER can effectively be used in an industrial context.

Section VI-A is focusing on the first two aspects while Section VI-B reports on the last one.

A. Modeling Approach Effectiveness and Transformation Overhead

Blueprint	Used Technology	Max dep.	Num. Elems	Avg generation time [s]	Gen. LOC	Avg deploy. time [s]	Avg teardown time [s]
Pi Calculus	Spark	3	3	1.18	294	442	55
WordCount	Zookeeper Storm	3	4	1.26	393	445	70
Hadoop cluster	HDFS YARN	4	5	1.31	646	550	65
WikiStats	Storm Zookeeper Cassandra	4	7	1.69	782	480	85
WikiStats (with trace-cheking service)	Storm Zookeeper Cassandra Spark HDFS	5	10	1.93	1175	675	92

TABLE I: Summary of transformations and deployments

To verify that DICER works correctly without introducing a significant overhead, we have experimented with various DIAs that exploit different combinations of the technologies we currently support. Table I shows a summary of some of these experiments. In all the cases, we developed the deployment model for the corresponding DIA and we generated and deployed the resulting blueprint on our internal OpenStack Mitaka infrastructure[17]. In all considered cases, the process led to a correct deployment fulfilling our expectations. The first column in Table I includes the names we have assigned to the considered DIAs and the second lists the used technologies. From this column the reader can see that we have experimented with an increasing combination of technologies to achieve deployments with a different level of complexity. The *Num. Elems* column reports the number of UML modeling elements used to model each DIA. The *Max dep.* column represents the length of the longest path in a blueprint's dependency graph, not counting virtual resources that are created almost instantly. For example, level 1 represents an architecture with services that are all peers and where none of

[17]The executable blueprints for all experiments are available online, the URL is not provided for the sake of anonymisation.

the services depends on any other service. Level 2 represents a two-tier architecture, where a service (e.g., a database) needs to be deployed before another one (e.g., an application) can be deployed and configured. The larger is the value of *Max dep.* the larger is the complexity of the deployment procedure to be executed for the corresponding DIA as it requires a specific deployment order to be kept as well as a runtime configuration of the dependencies between the involved components. *Gen. LOC* reports on the number of Lines of Code of infrastructural software generated by DICER in each experiment. This number gives an idea on the amount of low level programming effort saved by DICER users. From the experiments we have conducted, it appears that this number depends on the number of modeling elements.

To quantify the time needed to generate and deploy blueprints, we repeated the experiments 10 times and computed the average duration for generating the blueprint (fifth column in the table), for automatically deploying it (seventh column), and for tearing it down (eighth column, this is the time from when we started the teardown procedure to when the DICER deployment service reported termination). As we expected, we can see that the blueprint generation time is significantly lower that the time needed to deploy all considered cases.

Figure 6 illustrates further details on the timings for the deployment of the WikiStats application, where the scheduling due to dependencies, is clearly visible. The node templates with names containing *security_group*, *ip* and *zookeeper_quorum* are all virtual resource instantiations which take a minimal time to perform. Then the *vm* nodes follow, bringing up the virtual machines. This bootstraps service configuration according to the derived dependency graph. When configuration is complete, WikiStats is launched automatically.

To see if DICER was able to handle also cases more complex that the ones in the table, we carried out a scalability analysis starting from a simple model with a single

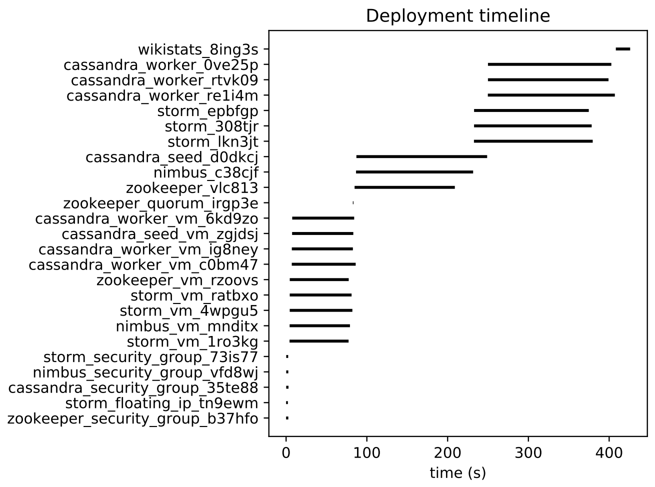


Fig. 6: Wikistats application deployment timeline.

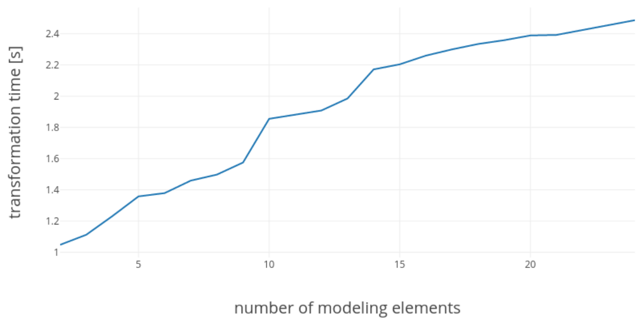


Fig. 7: TOSCA blueprint generation avg times per number of modeling elements.

VMsCluster including one technology element and by gradually increasing the complexity up to a case that included, 8 VMsClusters, 8 distinct technology-specific elements (e.g. StormCluster, CassandraCluster, etc.) and 8 distinct DiaJobs. Such models were randomly generated and the amount of generated infrastructural code ranged from 150 lines, for the simplest model, to 2092 lines for the more complex one. We ran each single experiment of blueprint generation 30 times, averaging the resulting generation time. In all cases DICER behaved correctly and generated the corresponding executable bluprint. Figure 7 plots the time needed for such generation that remains always within acceptable boundaries.

B. DICER Evaluation in Industry

The research questions we wanted to answer in the scope of our industrial evaluation were the following:

- **RQ1**: does the DICER tool cover the needs of the considered industrial use cases?
- **RQ2**: are the DICER UML profile and its consistency checking feature useful and effective for modeling data-intensive architectures?
- **RQ3**: can DICER be used to effectively deploy and re-deploy industrial use cases?
- **RQ4**: does the DICER approach provide some time saving?

Positive answers to these questions would give an evidence that the requirements we gave in Section III have been satisfied. Indeed, **RQ1**, **RQ2** and **RQ3** are related to the satisfaction of **REQ1**; **RQ2** and **RQ4** relate to **REQ2**; **RQ2** relates to **REQ4**; **RQ3** relates to **REQ3**.

In order to answer these questions, we applied DICER in two medium-sized companies that, for anonymization purposes we call NT and AC. Both of them were already familiar with UML modeling. This allowed as to focus on the actual value provided by DICER as a UML-based model-driven tool. In the next future we plan to extend the analysis also to companies without previous knowledge about UML to assess the advantages and disadvantages also in this case.

Industrial DICER trials required practitioners to model, deploy and change/improve their own architecture with and without DICER, reporting their observations and experiences in a survey[18].

In the following, we first introduce our two case studies (Sections VI-B1 and VI-B2), then we present what they reported about experiencing with DICER (Section VI-B3), finally we discuss and draw conclusions on our research questions (Section VI-B4).

1) Blu: NT is a company working in the e-government domain. NT is completely new to the Big Data domain and is starting to develop Blu[19], a tax fraud detection system that shall analyse Big Data in order to send alerts whenever a suspicious tax declaration enters the information system. Blu is implemented as a set of Spark applications that periodically process tax declations and store the results into Cassandra for subsequent analysis and for notifying alerts. NT's main challange is that they want to shorten as much as possible Blu's time to market and, to this aim, they need to highly integrate the development process with the process of continuously prototyping their DIA in a production-like environment.

2) NewsRepository: AC is a company working in the news and media domain. AC owns NewsRepository[20], a management solution for handling large volumes of news. AC is adding to NewsRepository a new module called X for anomization purposes, an innovative service for assessing the trustworthiness of information found in Social Media. By processing Big Data from several sources online, including social networks, X allows its users to verify the trustworthiness of a piece of information. The core of X is implemented as two distinct Storm applications that perform stream processing on the content coming from the various data sources, namely Twitter APIs, a MongoDB instance and a Kafka cluster, and store result again in the MongoDB instance. AC's main challenge when we started working with them was that X needed to be experimented upon iteratively and re-configured and re-deployed multiple times in order to meet a specific performance requirement. AC currently experiments with their application using ad-hoc scripts. Also some of the deployment steps currently require human/manual intervention on the nodes (VMs) which is usually an error prone process.

3) Introducing DICER: In order to address all the aforementioned problems, NT and AC introduced the DICER tool in their development processes. However, before starting using DICER, NT measured the time needed for a complete manual deployment and configuration of their DIA. Such process was conducted by a software engineer highly skilled in Linux and in Java and took exactly 63 working hours. In this timeframe the engineer went from zero knowledge of the Big Data frameworks to be used to a complete and working deployment. According to NT, this time was mainly due to the lack of tools for tasks automation. NT highlighted that, a mobile application developer can write the code, compile, deploy and simulate his/her application by simply pushing the "Run" button. A similar situation occurs in the case of web applications, while this is currently not possible for Big Data application developers. NT also argued that for complex applications like Big Data ones, developers have to manually deal with the whole development cycle, as there are several aspects to fine tune and keep under control to ensure the fulfillment of the critical QoS requirements.

AC declared previous experience in deployment of DIA frameworks and declared that a complete re-deployment of their DIA currently takes around 4 hours of work of an expert engineer.

The main benefits experienced by both our case studies using the DICER approach were related to 1) the simplified design of an execution environment, which was also useful for the purpose of easily identifying and documenting their architecture (**RQ2**), 2) the automated deployment of such design (**RQ3**) and 3) the nice integration between Dev practices (UML-based design) with Ops pratices (rapid deployment in pruduction environments). Using DICER, NT was able to design the deployment model and to deploy Blu in around 6 hours, while AC estimated a time saving of around the 80%, since with DICER they were able to define a valid deployment model and to deploy their DIA in around 40 minutes (**RQ4**). In both cases, further re-deployment took even less, considering that the model had only to be re-configured but not created from scratch (**RQ3** and **RQ4**).

A limitation AC found with respect to the X case study was concerning the currently supported set of data-intensive technologies (**RQ3**) that do not include Redis and Apache Solr. AC highlighted the need for improving the extensibility mechanisms offered by DICER to make sure that new technologies can easily be added when needed.

4) Discussion: Overall, both our case studies, although with different needs, were satisfied with their experience using DICER and observed significant advantages from introducing DICER in their development process (**RQ1**). The experiences with our case studies provide us all positive answers also to **RQ2**, **RQ3** and **RQ4**. The main observed advantage resides in the integration DICER provides between modeling activities and automated deployment, which is the core aspect of DICER DevOps flavor. In fact, by using simple models to describe the deployment of complex infrastructures and by automating the deployment starting from such models, our partners experienced considerable gains both in terms of reasoning on and documenting their architectures, as well as in terms of time saved when deploying and re-deploying them. The answer to **RQ2** can be considered particularly positive, since they gained a lot from having a modeling approach for automating but also describing their deployment and the design process was made even faster thanks to the use of UML and of the model validation feature provided by the OCL constraints. Also the answer to RQ4 is remarkably positive, given the estimated time saved thanks to DICER and the quite complex architecture of both the case studies. The less positive answer concerns

[18]Reference to full case-study descriptions is omitted for the sake of anonymization

[19]This is a name of fantasy used for anonymization purposes.

[20]This is a name of fantasy used for anonymization purposes.

RQ3. According to AC's experience, this is mainly due to the lack of support for some of the required technologies. Indeed, this is symptom of a relevant issue connected to the Big Data phenomenon, i.e. the explosion of the number of (often overlapping) data-intensive technologies. Although DICER tries to support many of the most popular ones, it certainly does not cover all the needs. Providing simple extension mechanisms to integrate new Big Data technologies is currently part of future work.

VII. Related Work

DICER exploits models to support the deployment and configuration of DIAs. Other works apply models for configuration and deployment in the Cloud. Ferry et al. propose CloudML [8], a domain-specific language for application deployment on multiple Clouds. CloudML comes with its own orchestration framework. Other approaches use TOSCA as a modeling framework, e.g., [11] and [12] - these discuss seamless model-based orchestration of application topologies by transforming them into TOSCA and adopting TOSCA-enabled orchestrators. These works focus on general Cloud-based architectures, with a consequent shortcoming for TOSCA-enabled DIA solutions. In certain respects, Ubuntu Juju[21] is DICER's competitor. It is a Cloud orchestration framework complete with a graphical Web user interface for building models of Cloud applications and provides support also for multiple data-intensive technologies. However, Juju's bundles are particular to Juju's own orchestrator, while DICER uses a portable and orchestration-neutral TOSCA notation. Moreover, with respect to all these related works, DICER is based on UML and is embedded into the Eclipse IDE, thus it provides higher integration between Dev and Ops functionalities. In this respect, some initial efforts reflect on connections between UML and TOSCA [13] [14], but authors conclude with further research plans for their full interoperability and we did not see in the literature further development of these plans.

Model-Driven Engineering is nowadays a well-established discipline to support the conception, the development and the operation of software in various application domains. The variety of domains is witnessed by initiatives like AUTOSAR [15] that focuses specifically on automotive. In the Big Data context, authors of [16] focus on modeling for the purpose of application code generation in the Hadoop framework. Similar support is offered by Stormgen [17], a DSL for Storm-based *topologies*. Neither of the approach tackle the deployment aspects and they focus on a single technology, while the key challenge in Big Data is the necessity of assembling many technologies at the same time. DICER can be considered as complementar to these approaches, being specialised on the deployment and configuration aspects, rather than on the actual development. Moreover, it is technology agnostic, allowing the combination of many different pre-existing technological frameworks, and, thanks to TOSCA, generates standard IaC.

[21]https://jujucharms.com/

VIII. Conclusion

DICER is a model-driven approach and supporting tool to help users in developing IaC for DIAs that exploit and combine different Big Data frameworks, whose (re-)deployment and (re-)configuration phases are therefore complex and time-consuming. The approach supports DIAs undergoing continuous improvement cycles by enabling the automated creation of deployable IaC from properly stereotyped UML deployment diagrams. We have shown that DICER can be successfully applied to industrial use cases with significant gain in particular in terms of time saved for deploying and re-deploying DIAs. Currently DICER supports some of the most well known Big Data frameworks. Adding new frameworks is possible and we have been constantly doing this in the last year, however, it requires an in-depth knowledge of the approach and tool. As a future work, we are focusing on the definition of proper extension mechanisms that will be simpler to use by less expert users. Moreover, although we have been focusing on the deployment of data-intensive architectures, our approach is not limited to them and we plan to extend DICER towards other complex deployments, such as Microservice-based ones.

References

[1] C. A. Cois, J. Yankel, and A. Connell, "Modern devops: Optimizing software development through effective system interactions." in *IPCC*. IEEE, 2014, pp. 1–7.

[2] K. Morris, *Infrastructure As Code: Managing Servers in the Cloud*. Oreilly & Associates Incorporated, 2016.

[3] R. Yin, *Case Study Research: design and methods*. Sage Publications Inc, 2003.

[4] M. Artac, T. Borovsak, E. D. Nitto, M. Guerriero, and D. A. Tamburri, "Devops: introducing infrastructure-as-code." in *ICSE (Companion Volume)*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. ACM, 2017, pp. 497–498.

[5] P. Lipton, D. Palma, M. Rutkowski, and D. A. Tamburri, "TOSCA solves big problems in the cloud and beyond," *IEEE Cloud, To appear*, vol. 21, no. 11, pp. 31–39, 2016.

[6] N. W. Grady, M. Underwood, A. Roy, and W. L. Chang, "Big data: Challenges, practices and technologies: Nist big data public working group workshop at ieee big data 2014." in *BigData Conference*, J. J. Lin, J. Pei, X. Hu, W. Chang, R. Nambiar, C. Aggarwal, N. Cercone, V. Honavar, J. Huan, B. Mobasher, and S. Pyne, Eds. IEEE, 2014, pp. 11–15.

[7] M. Kiran, P. Murphy, I. Monga, J. Dugan, and S. S. Baveja, "Lambda architecture for cost-effective batch and speed big data processing." in *Big Data*. IEEE, 2015, pp. 2785–2792.

[8] N. Ferry, A. Rossini, F. Chauvel, B. Morin, and A. Solberg, "Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems," in *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, ser. CLOUD '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 887–894.

[9] L. Baresi and S. Guinea, "Event-based multi-level service monitoring," in *Proceedings of the 2013 IEEE 20th International Conference on Web Services*, ser. ICWS '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 83–90.

[10] L. Baresi, S. Guinea, G. Quattrocchi, and D. A. Tamburri, "Microcloud: A container-based solution for efficient resource management in the cloud," in *2016 IEEE International Conference on Smart Cloud (SmartCloud)*, Nov 2016, pp. 218–223.

[11] A. Brogi, J. Carrasco, J. Cubo, F. D'Andria, E. Di Nitto, M. Guerriero, D. Pérez, E. Pimentel, and J. Soldani, *SeaClouds: An Open Reference Architecture for Multi-cloud Governance*. Cham: Springer International Publishing, 2016, pp. 334–338.

[12] J. Wettinger, U. Breitenbcher, and F. Leymann, "Standards-based devops automation and integration using tosca," in *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, Dec 2014, pp. 59–68.

[13] A. Bergmayr, J. Troya, P. Neubauer, M. Wimmer, and G. Kappel, "Uml-based cloud application modeling with libraries, profiles, and templates," in *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud, CloudMDE@MoDELS 2014, Valencia, Spain, September 30, 2014.*, 2014, pp. 56–65.

[14] A. Bergmayr, U. Breitenbcher, O. Kopp, M. Wimmer, G. Kappel, and F. Leymann, "From architecture modeling to application provisioning for the cloud by combining uml and tosca," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science, Volume 2: CLOSER*, 2016, pp. 97–108.

[15] G. Sandmann and R. Thompson, "Development of autosar software components within model-based design," in *SAE Technical Paper*. SAE International, 04 2008.

[16] A. Rajbhoj, V. Kulkarni, and N. Bellarykar, "Early experience with model-driven development of mapreduce based big data application," in *Software Engineering Conference (APSEC), 2014 21st Asia-Pacific*, vol. 1, Dec 2014, pp. 94–97.

[17] S. Santurkar, A. Arora, and K. Chandrasekaran, "Stormgen - a domain specific language to create ad-hoc storm topologies," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, Sept 2014, pp. 1621–1628.