Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

DICE Methodology

Deliverable 2.5

Deliverable:	D2.5
Title:	DICE Methodology
Editor(s):	Youssef RIDENE (NETF)
Contributor(s):	Joas Yannick KNOUANI, Damian A. Tamburri (PMI), Matej Artac (XLAB), Diego Perez (ZAR), Giuliano Casale (IMP), Jose-Ignacio Requeno (ZAR), Danilo Ardagna (PMI), Marcello Bersani (PMI), Marc Gil (PRO), Gabriel Iuhasz (IEAT), Pooyan Jamshidi (IMP), José Merseguer (ZAR), Darren Whigham (FLEXI), Chen Li (IMP), Ismael Torres (PRO)
Reviewers:	Chen Li (IMP), Vasilis Papanikolaou (ATC)
Type (R/DEM/DEC):	Report
Version:	1.0
Date:	31-July-2017
Status:	Final version
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2017, DICE consortium – All rights reserved

DICE partners

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

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

Executive Summary

The objective of DICE is to reduce time to market of business-critical Data-Intensive Applications (DIAs). DICE provides a bench of off-the-shelf tools that, if employed methodologically, enable users to build Big Data software efficiently. For that purpose, DICE proposes and recommends scenario-driven workflows depending on the user needs. These scenarios constitute **the DICE Methodology**.

In the following sections, we will first start by identifying typical actors found in Big Data software development projects. Next, we will overview DICE tools. Finally, we will present the DICE methodology that shows, in a scenario-driven way, how the tools can be beneficial to the actors.

1. Actors of Big Data Projects

A.1. NIST Taxonomy

The National Institute of Standards and Technology (NIST) published a valuable Big Data taxonomy depicted on Figure 1. Actor symbols represent functional roles, while component boxes denote software or hardware they create or employ. Roles are played by actors who can perform multiple roles while a role can be played by multiple actors. From left to right Fig. 1 shows the information flow chain: information is provided by *Data Providers*, digitized by *Data Producers*, processed by a *Big Data Application* and the output of the computation is presented by *Data Consumers* to *Data Viewers*. From top to bottom is pictured the service use chain: *System Orchestrators* expect some services from the *Big Data Application*, which is implemented by *Big Data Application Providers*, with the help of *Big Data Frameworks* designed by *Big Data Framework Providers*. The different activities of these five roles are encompassed by security and privacy issues. Table 1 gives examples of actors for each role.

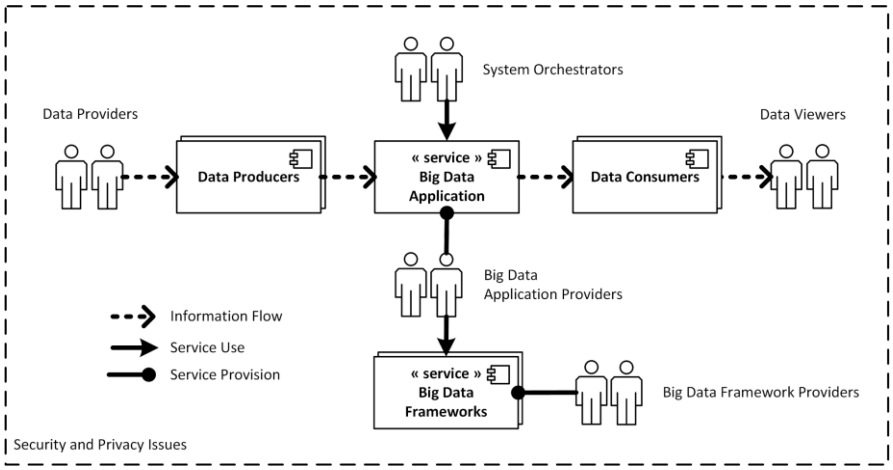


Figure 1. NIST Big Data Taxonomy

Table 1. Roles in a Big Data ecosystem according to the NIST

Role	Description	Example of actors
Data Provider	Introduces new data into the ecosystem.	Companies, public agencies, researchers, scientists, internauts.
Data Viewer	Utilizes the results of the Big Data application.	Companies, public agencies, researchers, scientists, software agents.
System Orchestrator	Specifies requirements and/or monitors their realization.	Clients, business stakeholders, project managers, consultants, requirements engineers.
Big Data Application Provider	Implements requirements.	Software engineers, network engineers, security and privacy engineers, developers.

Big Data Framework Provider	Provides infrastructures, computational resources, networks, operating systems, development platforms, and/or scalable storage or processing frameworks.	Cloud providers, companies, open source communities, system administrators, operators.

Three Big Data components interact with the Big Data application: *Data Producers*, *Big Data Frameworks* and *Data Consumers*. Table 2 displays some examples for each category.

Table 2. Big Data components

Component	**Description**	**Examples**
Data Producer	Converts information provided by Data Providers into digital data that can be processed by computers.	Sensors, cameras, Web browsers, graphical user interfaces.
Big Data Framework	A specific technology stack providing abstractions to store query and/or analyse data.	Apache Cassandra, Apache Hadoop, Apache Spark, Apache Storm.
Data Consumer	Presents data computed by the Big Data Application to Data Viewers in a form that is understandable and usable by them.	Graphical user interfaces, Web sites.

Hereafter, the roles of *Big Data Application Provider* and *Developer* are considered to be the same. The appellation "Big Data" implicates that data produced by *Data Producers* is too big, too diverse and arrives too fast to be efficiently handled by traditional non-scalable database management systems.

A.2. The DICE Methodology

The DICE Methodology mixes three fruitful and influential approaches to software development: DevOps, Service Orientation and Model-Driven Engineering.

In a DevOps process, *Developers* build and test software in an isolated, so-called, development environment, while *Operators* are in charge of the targeted, final, runtime environment. The latter comprises entities planned to interact with the program: operating systems, servers, software agents, persons and so forth. Operators are responsible for, amongst other things, preparing the runtime environment, controlling it and monitoring its behavior, especially once the application is deployed into it. For instance, operators have to ensure the presence in the runtime environment of every *Big Data Framework* necessary for the application to work. Nowadays, the trend is to wrap frameworks into services available at definite combinations of an IP address and a port number. To use a service, the application generally has to uphold a specific communication protocol (e.g., HTTP) on top of which an application programming interface (API) enables it to trigger well-specified service actions. The *Big*

Data Application can itself be implemented as a service to be included in the runtime environment of another application. Figure 2 is an adjustment of the NIST Big Data taxonomy to DevOps.

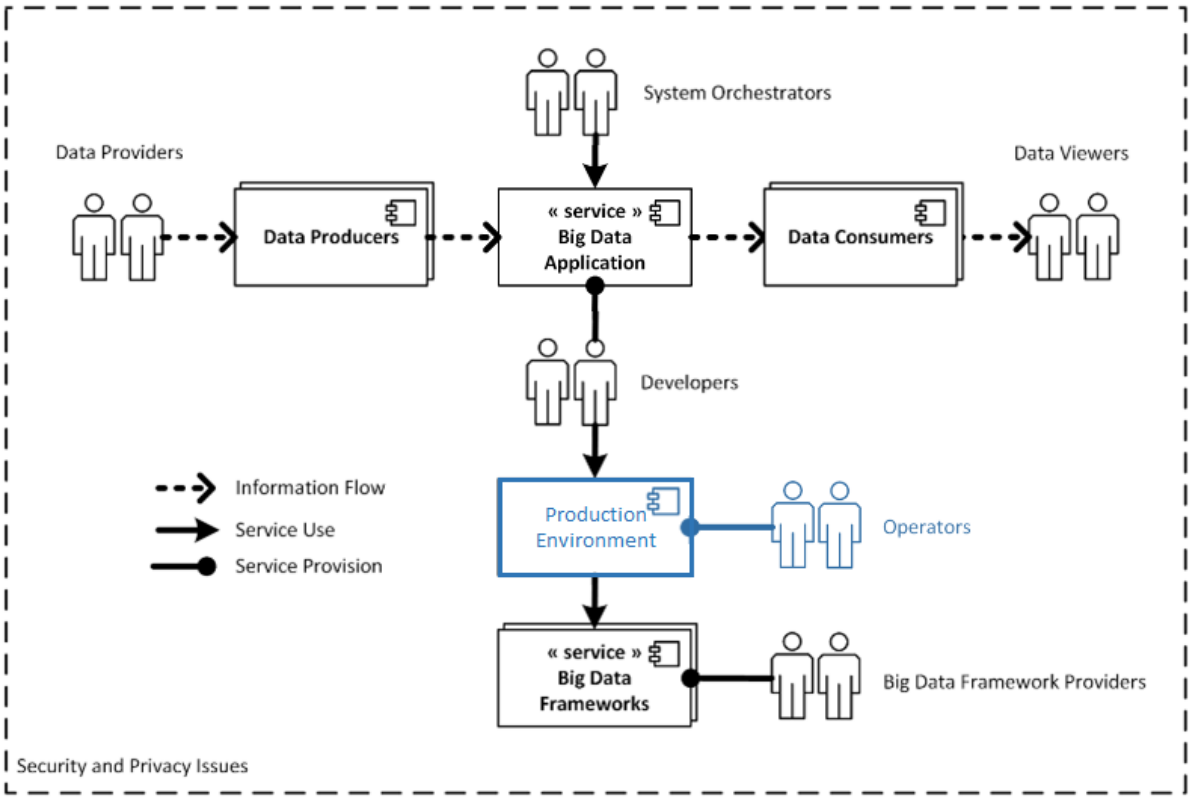


Figure 2. A DevOps Big Data Taxonomy

A contract is an artefact on which two parties agree after a comprehensive discussion. There is a contract among *System Orchestrators*, namely between clients and project managers. There is also a contract between *System Orchestrators* and *Developers*, because the latter has to implement what the former, after a careful requirement analysis, has established as software specifications. These specifications may be written in a textual document or, better, given as UML or mathematical formal models. In Model-Driven Software Engineering (MDSE), *Developers* iteratively refine these contracts/models with implementation details until they can generate a source code. The DICE Methodology proposes to extend this approach between *Developers* and *Operators*. In that context, from *Operators'* point of view, contracts/models set down the frameworks they must make available in their runtime environment to the application of *Developers*. For instance, a contract/model may ask them to install a Cassandra server. On the other hand, from *Developers*' point of view, contracts/models formulate the effects; they guarantee their application shall have on the runtime environment of *Operators*. For instance, they may ensure that their application streams to Cassandra a data flow that does not exceed a certain velocity. Contracts/models between *System Orchestrators* and *Developers*, and contracts/models between *Developers* and *Operators*, are iteratively synchronously refined until both the program and its matching runtime environment can be simultaneously generated. This workflow, where *Developers* and *Operators* collaborate and cooperate to reach an agreement on the runtime environment, complies fully with the DevOps culture.

Before detailing further the DICE Methodology, the next section overviews the DICE tools for Big Data software development.

2. Overview of DICE Tools

This section is a summary of Deliverable D1.4 (http://www.dice-h2020.eu/deliverables/) in which DICE tools are described in detail.

The DICE project is based on 14 tools: the DICE IDE, the DICE/UML profile, the DICE Rollout tool and the remaining 11 tools respectively for simulation, optimization, verification, monitoring, anomaly detection, trace checking, enhancement, quality testing, configuration optimization, fault injection, repository management and delivery.

Some of the tools are design-focused while some are runtime-oriented. Finally, some have both design and runtime aspects. Table 3 outlines the tools mapping them to this categorization scheme. All of the tools relate to the runtime environment. In other words, there is no tool that supports the model-driven development of application logic behind a Big Data job, e.g., a streaming job for Apache Storm such as the one reported in the "Big Data Applications Modelling" Section.

Table 3. DICE tools

	DICE tools
Design	• DICE/UML Profile • DICER • Simulation • Optimization • Verification
Runtime	• Monitoring • Quality Testing • Fault injection
Design-to-runtime	• Delivery
Runtime-to-design	• Configuration optimization • Anomaly detection • Trace checking • Enhancement
General	• DICE IDE

Design tools operate on models only, these either being software engineering models based on UML or quantitative models for performance/reliability assessment or verification. The DICE/UML profile is a UML-based modeling language allowing its users to create models of the data-intensive application arranged across three levels of abstraction: Platform-Independent, Technology-Specific and Deployment-Specific.

The DICE Platform-Independent Models (DPIM) specify, in a technology-agnostic way, the types of services a Big Data software depends on. For example: data sources, communication channels, processing frameworks and storage systems. Designers can add quality of service expectations such as performance goals that a service must meet in order to be useful for the application. More details can be found in http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2017/02/D2.2_Design-and-quality-abstractions-Final-version.pdf.

DICE Technology-Specific Models (DTSM) are refinements of DPIMs where every possible technological alternatives are evaluated until a specific technological design choice is made and rationalised. For instance, Apache Kafka can be selected as a communication channel, Apache Spark as a processing framework, and Apache Cassandra as both a data source and a storage system. DTSMs do not settle infrastructure and platform options. These are resolved in DICE Deployment-Specific Models (DDSM).

DDSMs elucidate deployments of technologies onto infrastructures and platforms. For instance, how Cassandra will be deployed onto any private/public Cloud.

Let us now describe the tools.

- The verification tool allows the DICE developers to automatically verify whether a temporal model of a DIA satisfies certain properties representing the desired behavior of the final deployed application. The formal model, that is obtained from the DTSM diagram, is an abstraction of the running application implemented with a specific technology. For each technology considered in DICE, there is a suitable (class of) temporal models allowing for the assessment of specific aspects of the applications which are captured by the temporal properties that the developer can verify.
- The simulation tool allows to simulate the behaviour of a data intensive application during the early stages of development, based on the DPIM specification. It relies on high-level abstractions that are not yet specific to the technology under consideration.
- Differently from the simulation tool, the optimization tool focuses on the DTSM, and relies on separate simulation capabilities to determined an optimized deployment plan in order to minimize cost subject to quality-of-service constraints (e.g., identifying the public cloud provider target for the deployment and the detailed configuration in terms of virtual machine instance type and their number).
- The DDSM model construction and its TOSCA blueprint counterpart is aided and automated by means of an additional tool called DICE Deployment Modelling. DICE Deployment Modelling in particular carries out the necessary automation to build an appropriate and well-formed TOSCA blueprint out of its DTSM modelling counterparts.

In contrast to the design tools, the runtime tools examine or modify the runtime environment directly—not its models.

- The monitoring tool collects runtime metrics about the components present in a runtime environment.
- The quality testing tool and the fault injection tool respectively inject workloads and force failures into the runtime environment; for instance, the fault injection tool shutdowns some computational resources in order to test the application resilience.

Some tools cannot be unambiguously classified as design or runtime tools because they have both design and runtime facets.

- The delivery tool is a model-to-runtime (M2R) tool that generates a runtime environment from a DDSM.
- The configuration optimization, anomaly detection, trace checking and enhancement tools are all runtime-to-model (R2M) tools that suggest revisions of models of the runtime environment according to data gathered by the monitoring tool. As opposed to the optimisation tool which is entrusted with optimising cost and resource consumption based on mathematical abstractions,

the configuration optimization tool analyses the infrastructure configuration parameters given a certain time horizon and returns optimal values for said infrastructural elements in a DDSM through experimentation on the deployed instance of the application.

- Finally, the anomaly detection, trace checking and enhancement tools analyse monitoring data. The first detects anomalous changes of performance across executions of different versions of a Big Data application. The second checks that some logical properties expressed in a DTSM are maintained when the program runs. The third searches anti-patterns at all DICE abstraction levels (DPIM, DTSM or DDSM).

Application codes, models and monitoring data are saved in a sharable repository, and most tools can be invoked directly through the DICE IDE (Figure 3).

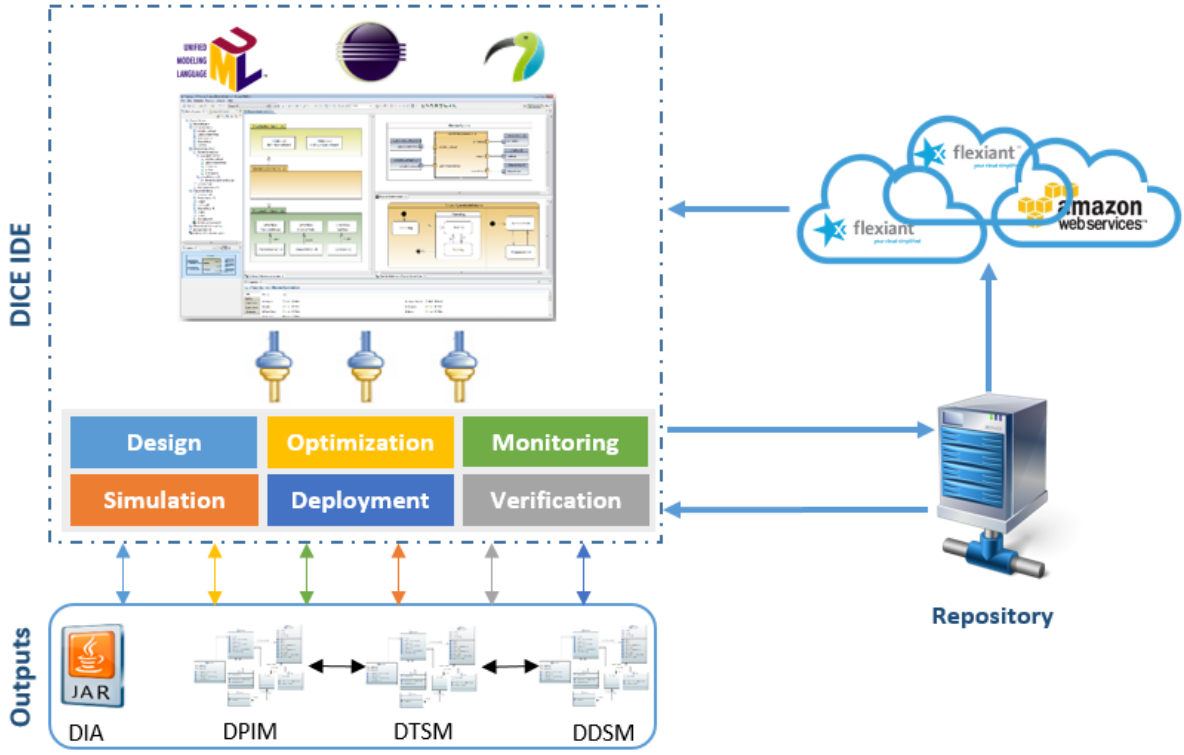


Figure 3. DICE ecosystem

Table 4 below summarises the UML/DICE diagrams each tool operates on.

Table 4. UML diagrams handled by the DICE tools.

DICE tool	Input UML diagram	Profile level
Simulation	Activity	DPIM, DTSM
	Sequence	
	Deployment	

Verification	Class	DTSM
Trace checking	Deployment	DDSM
Enhancement	Activity	DTSM, DDSM
	Deployment	
Optimization	Activity	DTSM, DDSM
	Deployment	
Monitoring	Deployment	DDSM
Deployment Modelling (DICER)	Deployment	DTSM, DDSM
Delivery tool	Deployment	DDSM
Quality testing	Deployment	DDSM
Configuration optimization	Deployment	DDSM
Anomaly detection	Deployment (indirect)	DTSM (indirect)
Fault injection	Deployment	DDSM

The DICE Knowledge Repository provides further information about each tool, including tutorials, installation guidelines, videos and getting-started documentation: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository.

3. Scenario-Driven Methodology

The DICE methodology adapts to the purpose of the user. Although DICE supports and advocates a holistic and integrated Model-Driven approach to Big Data software development, it also acknowledges that some users do not, and are not willing to practice refinement of technology-independent models into technology-specific and deployment-specific models. Particularly those looking for rapid prototyping functionalities. Therefore, in the following, we consider the use of DICE tools in four use case scenarios that illustrate alternative ways to exploit the DICE framework:

1) **Big Data Application Modeling.** The user of DICE can simply design his/her Big Data Application and s/he is willing to use an arbitrary subset of tools of the DICE framework, as long as it fulfills the needs. This modeling can be made for various purposes: (re-) documentation, architecture validation, etc.

2) **Standalone.** The user has identified a specific need which can be managed using a specific DICE tool. For example, if the user has a running Big Data application and needs to gather runtime metrics, then s/he is primarily interested to use the monitoring tool. In this scenario, the user will only have to follow a tutorial or read the documentation of this tool.

3) **Architecture Verification, Simulation and Optimization.** A development team has to implement a software that fulfills a list of requirements. Before starting the implementation, the team wants to use the performance and reliability engineering tools available in DICE to predict behaviors and cost for different implementation plans.

4) **DevOps.** A team of *Developers* has built a software and wants to automate (1) the creation of a matching runtime environment, (2) the deployment of their program into it and (3) the monitoring of its behavior in reaction to the actions performed by their application in a tight collaboration with *Operators*.

3.1. Big Data Applications Modeling

Nowadays Modelling has become a standard in software engineering. In fact, for architecture decision documentation, code generation or even simply for (re-)documentation purposes, Software Architects/Engineers build models using their favorite modeling environment and consistently with their specific industrial notations.

Beyond quality assessment, deployment automation, simulation, etc., DICE provides, first and foremost, a modeling environment for Big Data applications. These modeling capabilities are possible thanks to two different and complementary approaches: UML Profiles and a DSML (Domain Specific Modeling Language), that is, the modelling language embedded in DICE's own deployment modelling and automation tool, DICE Deployment Modelling . The following sections elaborate both approaches and highlight the characteristics of the modelling perspective, pointing the reader to further details in the knowledge repository.

3.1.1. DICE UML Modeling

3.1.1.1. Description

As aforementioned, the DICE ecosystem offers a plethora of ready-to-use tools to address a variety of quality issues (performance, reliability, correctness, privacy-by-design, etc.). In order to make profit of these tools, the user has to build specific UML diagrams enriched with stereotypes and tagged values brought by the DICE Profiles. The DICE profiles tailor the UML meta-model to the domain of DIAs. For example, the generic concept of Class can become more specific, i.e., to have more semantics, by mapping it to one or many concrete Big Data notions. Besides the consistency of the model remains guaranteed thanks to the meta-models behind the UML standard. In essence, the role of these profiles is twofold:

1. Provide a high level of abstraction of concepts specific to the Big Data domain (e.g., clusters, nodes…) and to Big Data technologies (e.g., Cassandra, Spark…);
2. Define a set of technical (low level) properties to be checked/evaluated by tools.

The methodological steps entailed by the activities above encompass at least the following activities:

a. Elaborate a component-based representation of a high-level structural architecture view of the data intensive application (i.e., a DPIM Component Diagram) - in the scope of DICE, this is done using the simple and familiar notations of a UML profile whence the user draws the stereotypes and constructs necessary to specify his/her Data-Intensive Applications nodes (source node, compute node, storage node, etc.);
b. Augment the component-based representation with the property and non-functional specifications concerning that representation;
c. Refine that very same component-based representation with technological decisions - the decisions themselves represent the choice of which technology shall realise which data-intensive application node. For example, a <<CassandraDataStore>> conceptual stereotype is associated with a <<StorageNode>> in the DPIM architecture view;
d. Associate several data-intensive technology-specific diagrams representing the technological structure and properties of each of the data-intensive nodes. These diagrams essentially "explode" the technological nodes and contain information specific to those technological nodes. For example, a <<StorageNode>> in the DPIM architecture representation can become a <<CassandraDataStore>> in its DTSM counterpart ; finally, the DTSM layer will feature yet another diagram, more specifically, a data-model for the Cassandra Cluster. These separate technology-specific "images" serve the purpose of allowing data-intensive application analysis and verification;

e. Elaborate a deployment-specific component deployment diagram where the several technology specific diagrams fall into place with respect to their infrastructure needs. This diagram belongs to the DDSM layer and contains all necessary abstractions and properties to build a deployable and analysable TOSCA blueprint. Following our <<CassandraDataStore>> example, at this level, the DTSM <<CassandraDataStore>> node (refined from the previous DPIM <<StorageNode>> construct) is finally associated with a DDSM diagram where the configuration of the cluster is fully specified (i.e., VMs type and number, allocation of software components to VMs, etc.);
f. Finally, once the data-intensive deployment-specific component diagram is available, DICE deployment modelling and connected generative technology (DICE Deployment Modelling) can be used to realise a TOSCA blueprint for that diagram.

In summary, Designers exploiting DICE UML modelling for their Data-Intensive applications will be required to produce (at least) one component diagram for their architectural structure view (DPIM) and two (or more) diagrams for their technology-specific structure and behavior view (DTSM), taking the care of producing exactly two diagrams (a structural and a behavioral view) for every technological node in their architectural structure view (DPIM) **as long as that requires analysis**. DICE UML modelling does not encourage the proliferation of many diagrams, e.g., for the purpose of re-documentation - DICE focus is on quality-aware design and analysis of Data-Intensive applications. Therefore, DICE UML modelling promotes the modelling **of all and only the technological nodes that require specific analytical attention and quality-awareness**. Finally, Designers will be required to refine their architectural structure view with deployment-specific constructs and decisions.

For example, for a simple WordCount application featuring a single Source Node, a single Compute Node and a single Storage Node, all three requiring specific analysis and quality improvement. Therefore, Designers are required to produce (in the DICE IDE) a total of 7 diagrams: (1) an architectural structure view of the general application, containing three nodes (Storage, Compute and Source) along with their properties and QoS/QoD annotations; (2) a structural and behavioral technology-specific view for every technology that requires analysis - let us assume a class diagram and an activity diagram for Storage, Compute and Source Node technologies respectively. Finally, the diagram produced in (1) is required to be refined with appropriate deployment-specific constructs, mappings and annotations.

The next section provides a realistic usage scenario of the above modelling procedure for the purpose of clarifying the DICE modelling process.

For more details (tutorials, documentation, examples, etc.), on the DICE profile and the connected technologies the reader may find additional elaborations on the DICE Knowledge Repository at:

https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#profile.

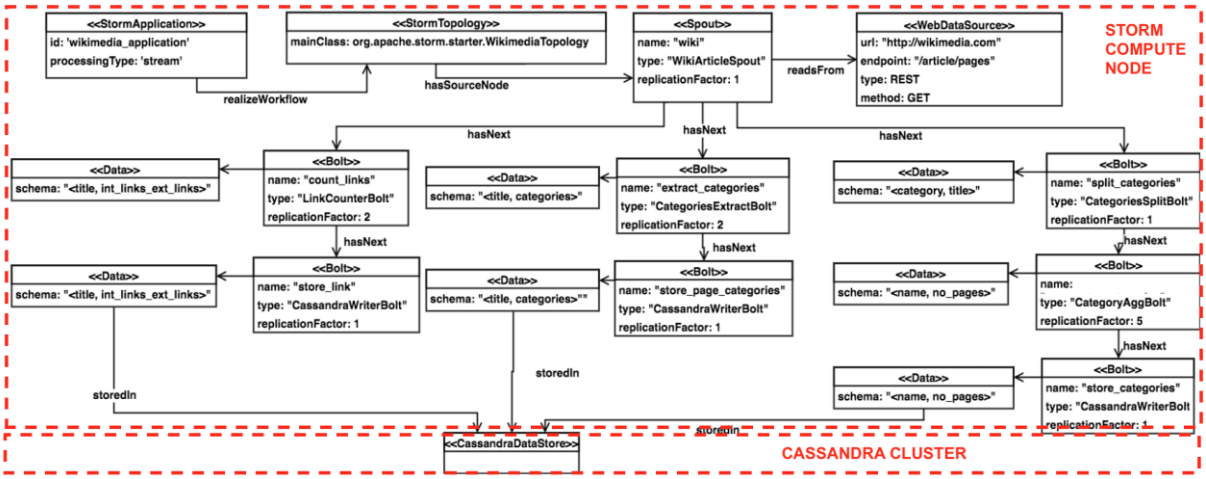
3.1.1.2. DICE UML Modelling in Action: A Sample Scenario

As a toy example, we refer to a simple Storm application of our own device called *WikiStats* which takes as input a compressed stream of 20GB web pages in XML containing snapshots of all the articles in Wikipedia. The application then processes the stream to derive article statistics. Let's assume we are interested initially in deploying our application as soon as possible rather than analyse its behavior;

Step a: DPIM Model - the DPIM model for our toy example is a component-based aggregation of two nodes: a compute node (entrusted with processing wiki pages) and a storage node (entrusted with storing and presenting results). For the sake of space we do not present this simplistic DPIM layer.

Step b and c: DPIM Model Refinement - at this point the DPIM model is used as a basis to refine DIA modelling with appropriate technological decisions; in our case, the DPIM component diagram components representing DIA nodes are stereotyped with additional technology-specific stereotypes, in our case that is the <<StormApplication>> stereotype for the only compute node in the DPIM; this signifies that the component is established to be a Storm Compute Node. Similarly, the DPIM component diagram component representing the storage node is stereotyped with an additional stereotype, that is, the <<CassandraDataStore>> stereotype; this signifies that the component is established to be a Cassandra cluster.

Step d: DTSM Model Creation - at this point, we need to "explode" the two nodes in our DPIM refined with technological decisions - all we need to do is to create a new class diagram and elaborate further on the technical-detail internals for both nodes (e.g., Storm topology details for the <<StormApplication>> and schemas for the <<CassandraDataStore>>). As a consequence, we prepare a new class diagram where a new class is created with the <<StormApplication>> stereotype and is immediately associated with bolts and spouts required in *WikiStats*; similarly, data schemas are prepared for bolts and linked to a <<CassandraDataStore>> class of which we assume no need for further internal details. The resulting diagram should look similar to the following figure.



Step e: DDSM Model Creation - at this point, the technologies used in the DTSM are mapped to physical resources and automated rollout is applied to obtain a deployable TOSCA blueprint (see DICE Delivery Service for additional deployment features). DDSM creation at this step involves creating or refining a UML Deployment Diagram with DDSM Profile Stereotypes. Continuous OCL-assisted modelling can be used to refine the UML Deployment diagram in a semi-automated fashion. In a typical scenario, the DICE user randomly selects a technology from the DTSM diagram and instantiates a Deployment Node to apply that technology stereotype on it. Subsequently, the DICE user can check the diagram for satisfaction of DICE-DDSM OCL constraints, addressing any missing dependencies for that technology as well as any missing deployment specifications (e.g., additional nodes, firewalls, missing characteristics and attributes, etc.). The same process shall be replicated by the DICE user until all the technologies in the DTSM are modelled at the DDSM level as well. Finally, a deployment artifact representing the DIA runnable instance itself shall conclude the modelling at the DICE DDSM layer.

The subsequent section, elaborates on how to prepare a deployment model independently and regardless of the DICE DPIM and DTSM diagrams if the DICE user is so inclined or required. DICE Deployment modelling after this point relies on a domain-specific language (DSL) specifically designed for independent DDSM modelling.

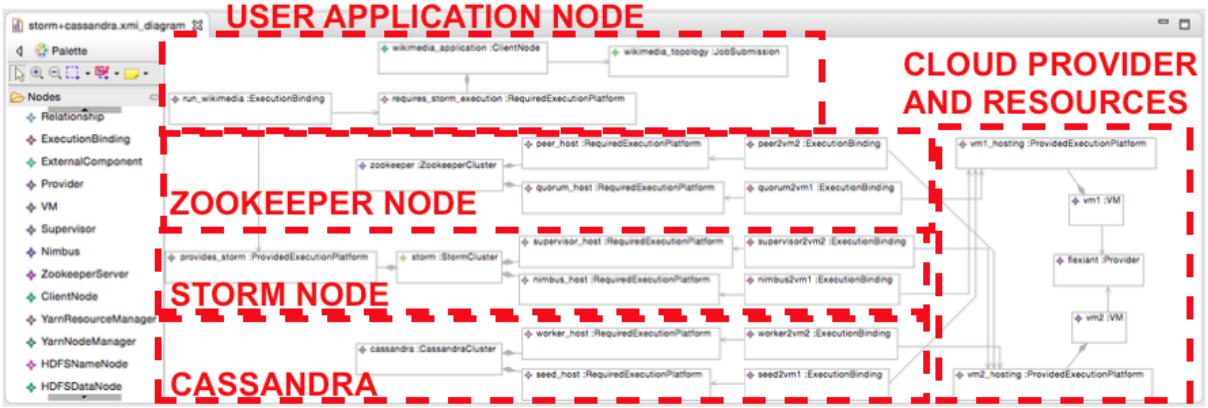
3.1.2. DICE Deployment Modelling

DICE Deployment Modelling is a complete environment (editor, palette, properties view, etc.) built upon an extensible set of Big Data specific-modeling languages (Storm, Hadoop, Cassandra, Zookeeper and Spark). Each such language captures the necessary deployment software nodes, concepts and restrictions that every big data technology addressed in DICE is composed of, along with their configuration characteristics (e.g., dependencies) and parameters (e.g., required and provided properties) as well as any dependencies on other nodes and technologies. This package structure is adopted to achieve modularity and ease DICE Deployment Modelling extension with new technologies.

The main benefits of using the DICE Deployment Modelling are related to the rapid design of an execution environment using concrete concepts. In a user-friendly approach, the users can employ the Eclipse Ecore modelling tool to create a deployment model (DDSM), taking advantage of the DICE profile.

For example, in the scope of re-documentation scenarios, software Architects or infrastructure Engineers may need to focus on re-documenting their architecture views - similarly, Architects may want to use DICE Deployment Modelling to elaborate on those architecture views using concepts and relations from well-known and established big data technologies (or many other concepts typical in infrastructure design for that matter, such as virtual machines, execution bindings, etc.).

DICE Deployment Modelling Modelling takes up at **step d** of the procedure highlighted above and allows to design a DDSM in a completely reserved environment specific for deployment details which can easily be extended to desired deployment-specific technologies and packages. DICE Deployment Modelling DDSM models are equivalent to DICE DDSM UML Profiles. For example, a diagram for the *WikiStats* application would look something like the following figure.



In turn, this DICE Deployment Modelling DDSM model can immediately be produced into a fully deployable TOSCA blueprint at the click of a button and sent for deployment, at the same time, using the built-in deployment service and delivery tool part of DICE.

DICE Deployment Modelling usage is encouraged when stakeholders and roles require quick and painless deployment of their own DIA (e.g., to evaluate initial performance figures and/or execution traces for further analysis). This notwithstanding, DICE Deployment Modelling may also be used in combination with the aforementioned modelling procedure, where step d is executed directly within the DICE Deployment Modelling instead of the DICE UML profile.

3.2. Standalone

The standalone usage mode is straightforward and is closely related to the built-in tools of the DICE IDE. For such scenario, the user is guided through some steps using Eclipse dialog windows. Since many DICE tools use the same input models, the user may run on the same model more than one tool and analysis. For example, the Configuration Optimization tool may be used automatically with the same input that the Deployment tool uses.

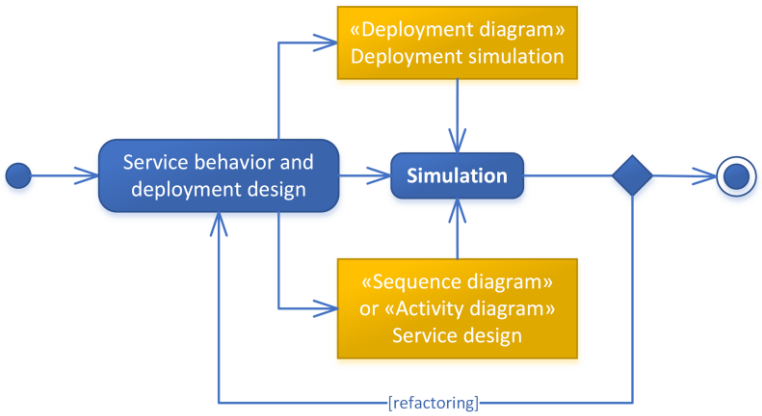
Some tools can be used outside the DICE IDE without using the modeling features (e.g. using command line). For such usage, the user may go through the guides available at: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository.

The rest of this section is devoted to explaining the necessary details to run the DICE analysis tools in standalone mode. The use of tools, which are intrinsically and inherently linked to the use of the DICE IDE (i.e., the design tools such as the UML profile, or the DICE Deployment Modelling tool), do not appear as standalone tools since their use as standalone in the scope of DICE only makes sense in continuity with the use of some other analysis tools - however, for further details on these the reader may refer to the DICE knowledge repository in the link above.

- **Simulation**

The Simulation Tool is able to simulate the behavior of a DIA to assess its performance and reliability. This tool uses Petri net models for prediction. The DIA is defined with behavioral UML diagrams, in particular Sequence or Activity diagrams that are complemented with the Deployment diagram. These diagrams are enhanced within the DICE profile. The DIA can be defined both at DPIM level or at DTSM level using a particular technology (e.g., Storm, Spark). The internal utilization of Petri net models is transparent to the DICE user, thus releasing his/her from any knowledge of the formal model.

The output of the simulation is the evaluation of a set of performance and reliability metrics for early-stage quality assessment. For example, the users can obtain, as performance results, the expected mean response time or throughput of the DIA, or the utilization of the resources assigned to the application. As reliability results, users can obtain failure probabilities for the application execution, or the mean time to failure of DIAs that execute continuously along time.

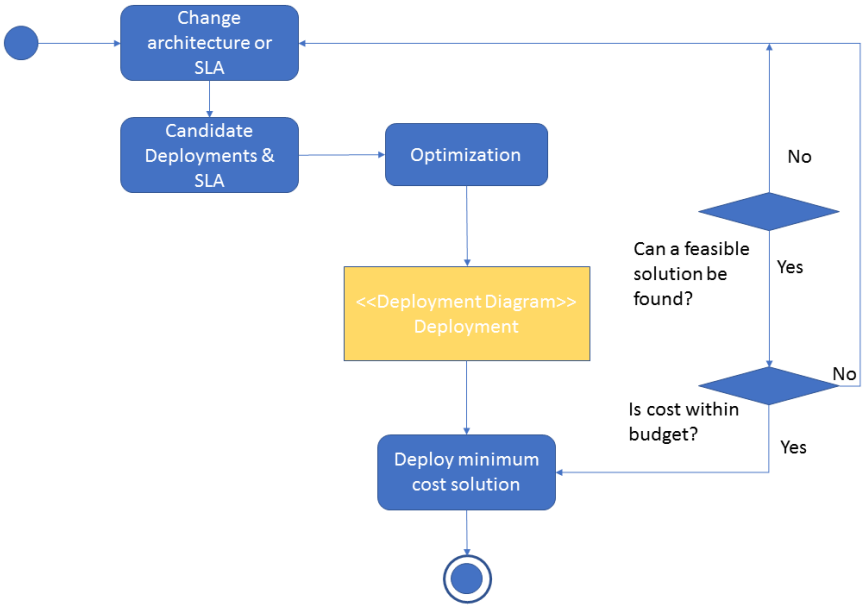


A simulation is performed considering a performance scenario. A performance scenario can be modelled by either a sequence (SD) or an activity diagram (AD). While the SD focuses on the message exchange between components (lifelines), the AD focuses on the actions performed by the components (partitions). The deployment (DD) is used to specify both the availability of resources in the system, e.g., number of cores, and how the interacting components (lifelines in the SD, partitions in the AD) are mapped onto physical nodes.

More details are available at : https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#simulation.

- **Optimization**

The optimization tool allows the DICE Architect to assess the performance and minimize the deployment cost of data-intensive applications against user-defined properties, in particular meeting of service-level agreements (SLAs). The input is: (i) a set of DICE DTSM models (one for every candidate target environment, i.e., virtual machine type at a given of provider), (ii) a partially specified DICE DDSM Deployment model and the SLAs to be achieved . The optimization consists in finding the less expensive cluster configuration able to guarantee the application jobs to be executed before a user defined deadline (for MapReduce or Spark applications) or such that the cluster utilization is below a given threshold (for Storm). The Architect can analyze the application behavior under different conditions. For example, he/she can study the pros and the cons of public clouds versus private cloud in terms of execution costs. The output of the optimization tool is a DICE Deployment model that corresponds to the optimal solution found and which can then automatically deployed by the DICE Deployment tool. The usage flow of the tool is reported in the figure below. Given the set of candidate solutions and SLAs, if a feasible solution cannot be found or the cost is not within a budget constraint, the architecture or SLAs need to be revised. Vice versa, the the DIA can be deployed to the target execution environment specified by the minimum cost deployment model identified by the optimization tool.



For more details, please visit :

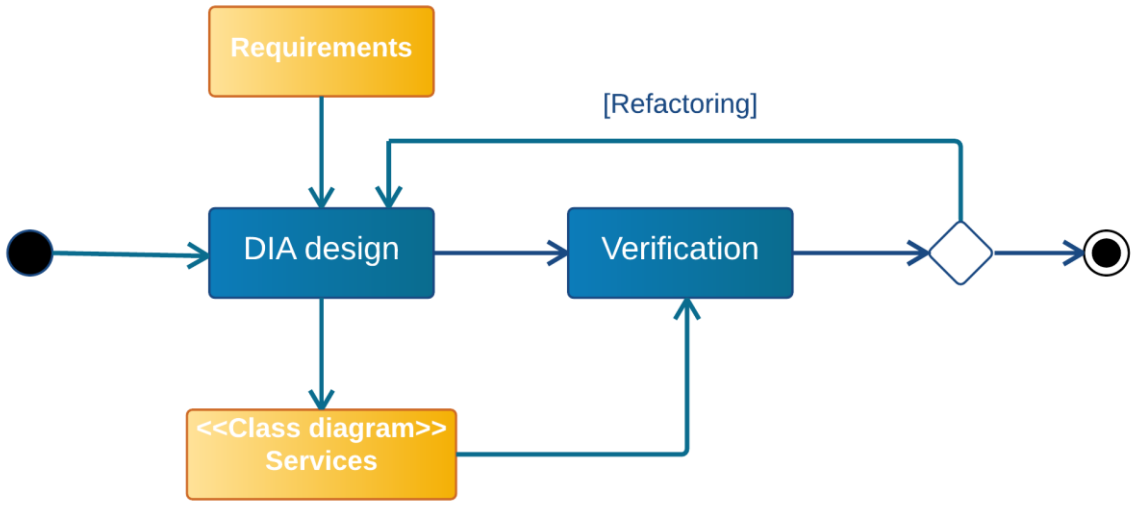
https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#optimization-tool.

- **Verification**

DICE Verification Tool allows Designers to evaluate their design against user-defined properties, in particular safety ones, such as reachability of undesired configurations of the system which might be caused by the effect of node failures or by the incorrect design of timing constraints. The verification process allows the DIA Designer to perform verification tasks using a lightweight approach. DICE Verification Tool fosters an approach whereby formal verification is launched through interfaces that hide the complexity of the underlying models and engines. These interfaces allow the user to easily produce the formal model to be verified and the properties to be checked without the need of high technical expertise. To promote verification, the user annotates the DPIM elements undergoing verification with the (class of) property that must be validated. For example, if the property is “queue boundedness” then the user annotates with a label “safety-queue boundedness” all the computational

nodes that require the validation of that property. In the DTSM, the generic annotations stated at the DPIM, requiring the verification of a property, can then be further enriched with more specific annotations that are related to the class of property to assess and to the technology employed to implement the node. Those specific annotations provide a value to all the necessary parameters that are needed to carry out the verification (for instance, the time required by tasks to process a message).

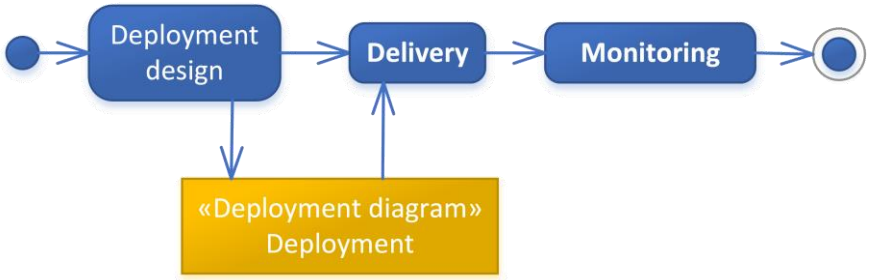
If the verification task fails then the Designer can refactor the design of the DIA. The outcome of the verification phase is a counterexample, i.e., an execution violating the property in analysis, that can help the designer in identifying the cause that originates the undesired behavior.



A more detailed description is available at the following link: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#verification.

- **Monitoring**

DICE monitoring platform (DMon) collects, stores, indexes and visualizes monitoring data in real-time from applications running on Big Data frameworks. DMon is able to monitor both the infrastructure (memory, CPU, disk, network etc.) and multiple Big Data frameworks currently supported being Apache HDFS, YARN, Spark, Storm and MongoDB. The core components of the platform (Elasticsearch, Logstash, Kibana) and the node components running on the monitored cluster are easily controlled thanks to a Web-based user interface that backs up the DMon controller RESTful service. Visualization of collected data is fully customizable and can be structured in multiple dashboards based on the user needs, or tailored to specific roles in your organization, such as Administrator, Quality Assurance Engineer or Software Architect.

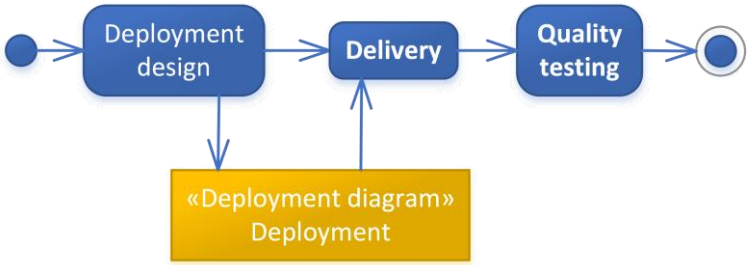


For more details, please visit : https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#monitoring-tool.

- **Quality Testing**

This is a suite of tools to help stress testing Data-Intensive Applications based on technologies such as Storm, Kafka and Spark. The tool allows Developer to run basic load tests on the application throughout the development cycle in order to support the activities of configuration optimization and anomaly detection across software versions.

The Quality Testing tool takes as an input initial dataset provided by the user and test scenario characteristics (e.g. load injection rate, volume and duration) from which it generates the application load and injects it into the application. The workload generation is handled by a module called QT-GEN, whereas the injection of the workload is enacted by a module called QT-LIB. The tool output is a measure of the application behaviour (e.g., throughput) that is visualized by the Continuous Integration tool. These dependencies are presented as diagrams (and traces) that can be obtained and visualised in DICE Monitoring tool.

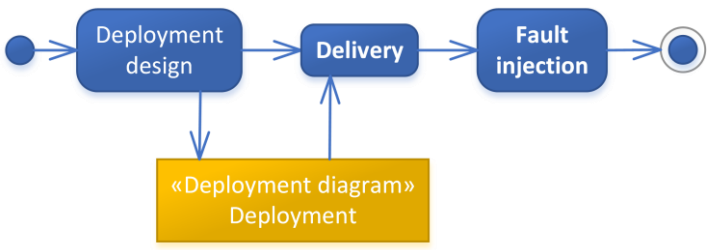


The tool requires a running DIA, which in a standalone scenario needs to be set up using the Delivery tools. This, in turn, uses DDSM as an input. The QT-LIB code is added to the application itself since it is provided as a Java library part of the DICE IDE. The input trace needs to be packaged within the DIA jar and it is assumed to be in JSON format. When using a realistic trace is not relevant for the test, an alternatively instantiation of QT-LIB consists in requiring the tool to generate random data of the appropriate size, which does not require to package within the DIA jar the data to be played. QT-LIB offers example templates to automatically halt the test experiment based on monitoring data obtained by the DICE Monitoring Platform.

For more details, please visit: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#quality-testing-tool.

- **Fault Injection**

The DICE Fault Injection Tool (FIT) has been developed to generate faults within Virtual Machines. The FIT provides the ability for a user to generate faults at the VM level. The purpose of the FIT is to provide VM owners with a means to test the resiliency of an application target. With this approach, the Designers can use robust testing, showing where to harden the application before it reaches a commercial environment and allows a user/application owner to test and understand their application design/ deployment in the event of a cloud failure or outage. Thus allowing for the mitigation of risk in advance of a cloud based application deployment. This tool will assist Developers and Cloud Operators in offering their best services to all customers and simplify testing within the DevOps paradigm.



1. User selects fault by GUI/command line;
2. Fault is started on VM;
3. Required tools installed and configured;
4. Fault Starts and is logged;
5. Fault Completes and status of run is recorded in log.

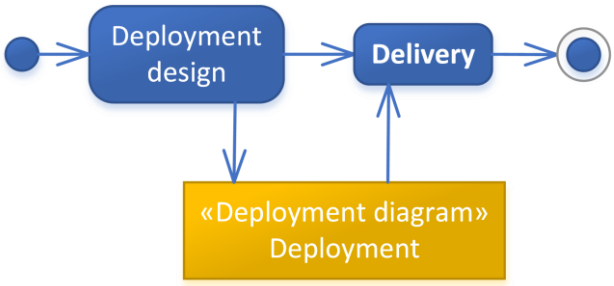
For more details, please visit: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#fault-injection-tool.

- **Delivery**

DICE delivery tools enable a simple way to continuously deploy Data-Intensive Applications on a testbed. Starting up Big Data clusters for running applications such as Storm topologies or Spark jobs is a matter of executing a TOSCA YAML document. The tools consist of a deployment modelling tool and a deployment service tool in cascade to the TOSCA-based deployment modelling tool. The deployment service is complete with a web user interface and command line interface. As a part of the delivery tools we also provide the DICE technology library that contains the configuration and deployment recipes for the supported Big Data services.

In their standalone usage mode, the tools can be used for experimenting with various setups of Big Data technologies without the need to spend effort on manually installing and configuring the cluster. The users can create throw-away clusters for fast prototyping, or persist the ones that prove useful in the form of a DDSM or its equivalent, the TOSCA blueprint, which works as a versionable description of the configuration. We have also designed the tools to work well in a Continuous Integration workflow.

The DICE delivery tools have a setup and configuration phase, when *Administrator* first boot-straps the DICE Deployment Service (along with Cloudify) as the support service for the test-bed. This phase also includes assigning input parameters that are specific to the platform hosting the test-bed. The configuration phase is a one-time (or at worst a very rare) operation.



A more frequent operation is creation of the deployment model of the DIAs. The users use the IDE to create a DDSM either using Eclipse Ecore modelling tool to create UML profiled-Deployment Diagram via the DICE Profile, or a specific TOSCA infrastructure diagram created directly within DICE Deployment Modelling. Subsequently, the DICE Deployment Modelling tool transforms DDSM into OASIS TOSCA blueprints represented as YAML documents.

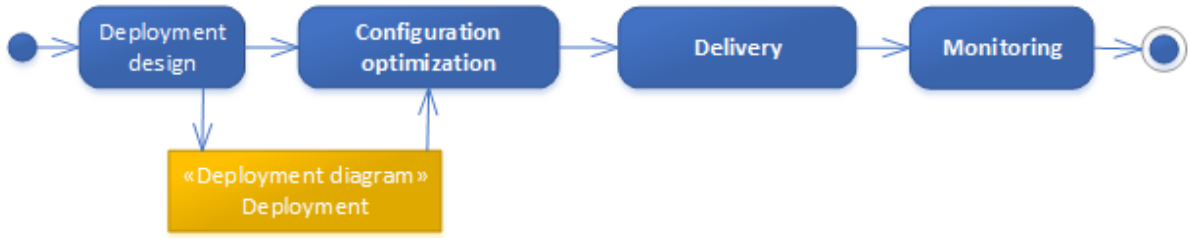
The main and certainly the most frequent interaction with the DICE delivery tool is deploying the DIAs according to their deployment diagrams. The users first choose or create a virtual deployment container as a destination of a deployment at the DICE Deployment Service. In the DICE IDE, they select the virtual deployment container and the YAML blueprint, and then submit the deployment. In the IDE, they can then monitor the status of the deployment (e.g., preparing to install, installing, and error).

An expected outcome is a new runtime of the DIA in the test bed. Depending on the blueprint, the dynamic (output) parameters for the DIA then become available (e.g., URLs of the deployed service). The users can then proceed to using, testing or experimenting with the DIA runtime.

For more details, please visit: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#delivery-tool.

- **Configuration Optimization**

The Configuration optimization (CO) tool automatically tunes the configuration parameters of Data-Intensive Applications. These are developed with several technologies (e.g., Apache Storm, Hadoop, Spark, Cassandra) each of which has typically dozens of configurable parameters that should be carefully tuned in order to perform optimally. CO tool enables end-users of such application to auto-tune their application in order to get the best performance. CO is integrated with DICE delivery tools (including deployment service and continuous integration) as well as DICE monitoring platform.

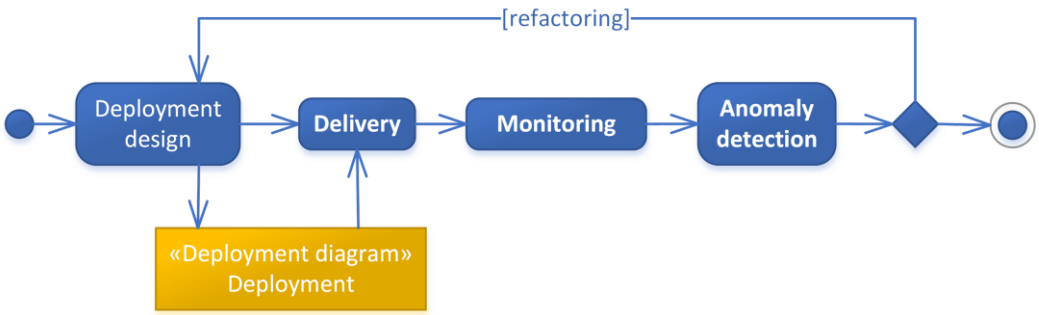


The deployment diagram (DDSM) for CO is a source of the initial configuration values, which are the starting point for the optimization. In other respects, the CO passes the deployment diagram (actually its transformation, the TOSCA blueprint) to the Delivery tool. The outcome is a new set of configuration values (as a collection of "parameter name: value" pairs or an updated TOSCA blueprint), which could be used in an enhanced DDSM.

For more details, please visit: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#configuration-optimization-tool.

- **Anomaly Detection**

The anomaly detection (AD) tool reasons on the base of black box and machine learning models constructed from the monitoring data. In order for models to be able to detect not only point anomalies but also contextual anomalies, the tool will select a subset of data features from the Monitoring Platform to train and validate a predictive model, which is later stored in Monitoring Platform itself. The predictive models are then used to detect contextual anomalies in real-time monitoring data streams. A second use case supported by the anomaly detection tool is the analysis of monitoring data based on two different versions of DICE application, thus detecting anomalies introduced by latest code changes.



In essence, during supervised anomaly detection, the user has to define a training set with labelled data. This means that the query issued by the AD tool to the monitoring platform will automatically generate

the training and validation set. In the case of unsupervised methods, this is not required. No user interaction is required.

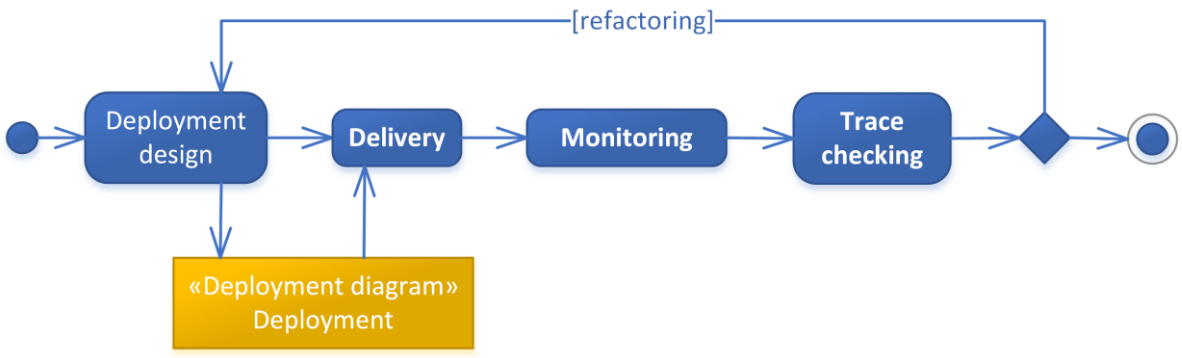
For more details, please visit: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#anomaly-detection-tool.

- **Trace Checking**

Trace checking is an approach for the analysis of system executions that are recorded as sequences of timestamped events, to establish whether a certain property (usually expressed in temporal logic) is satisfied.

Trace checking tool is employed to extract information that is not available from standard monitoring platforms whereas this information is specific of the modeling used for verification in DICE. The collected data is used to assess the validity of the results obtained at design time with the verification tool. Trace checking is carried out first by selecting the names of the nodes whose logs are going to be analyzed with trace checking. The property to assess is specified by the user who chooses it, in the plugin user interface, from a list of predefined properties. The properties are specific for the technology adopted in the DTSM and might also depend on the class of verification properties specified through suitable annotations on nodes in the DPIM. The user then specifies the time duration of the portion of the logs to be extracted from the monitoring platform and, finally, runs the trace checking. In case of negative response, i.e., the property is violated by the logs, the user might redo a trace checking analysis with a longer log trace or run the verification again by considering an updated model of the application where the parameter values are determined from the collected logs.

Privacy constraints on the use of resources can be assessed through trace checking at DPIM level. The user specifies the accessing relations among the components of the system through SecureUML profile and annotates the nodes that undergo specific accessing restriction with a time window, composed of two time bounds expressing the beginning and the end of the period of time under analysis. Throughout the time window, the accessing constraints that must be enforced by the system, can be verified via trace-checking analysis. The designer selects the privacy property for the previously annotated components in the trace checking user interface and then runs the tool. Based on the technology used to implement the node, that the user specifies in the DDSM, the log retrieval can be carried out automatically and the trace-checking executed on the collected execution trace.

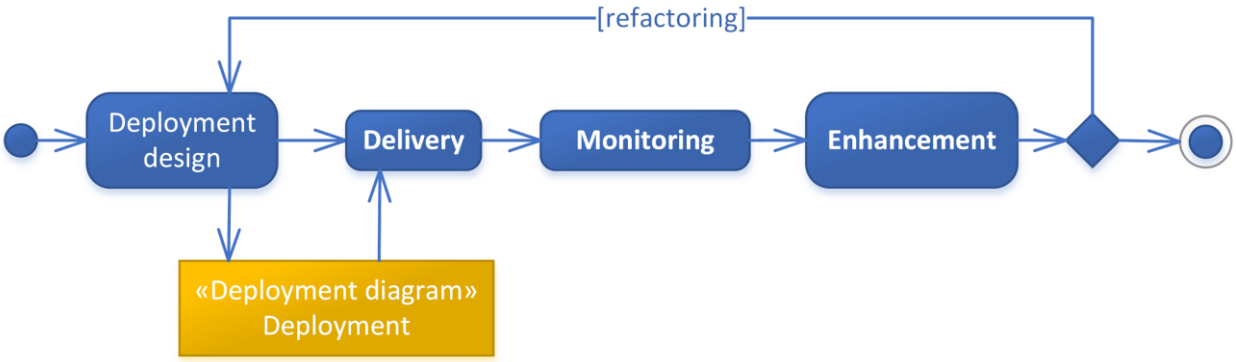


More details are vailable at: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#trace-checking-tool.

- **Enhancement**

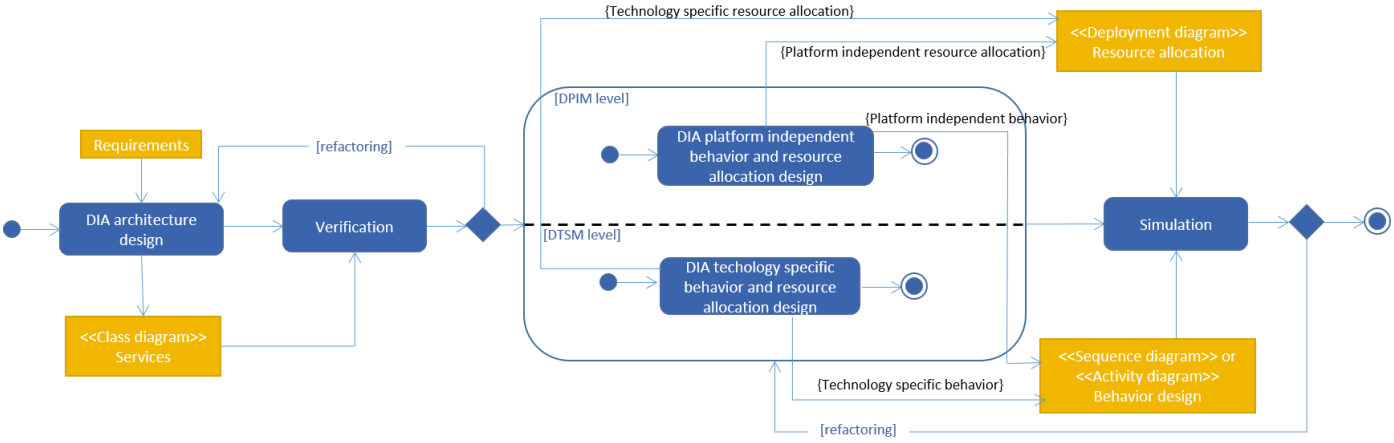
The Enhancement tools are the core DICE solution to close the gap between runtime (monitoring data) and design time (models and tools). They feature the correlation between monitoring data to find the

existence of different abstractions between design concepts and runtime measurements, that is, feeding results back into the design models to provide guidance to the developer on the quality offered by the DIA at runtime. They also provide the initial parameters for the Simulation tools and Optimization tools from monitoring data and detect the anti-patterns in the application design.

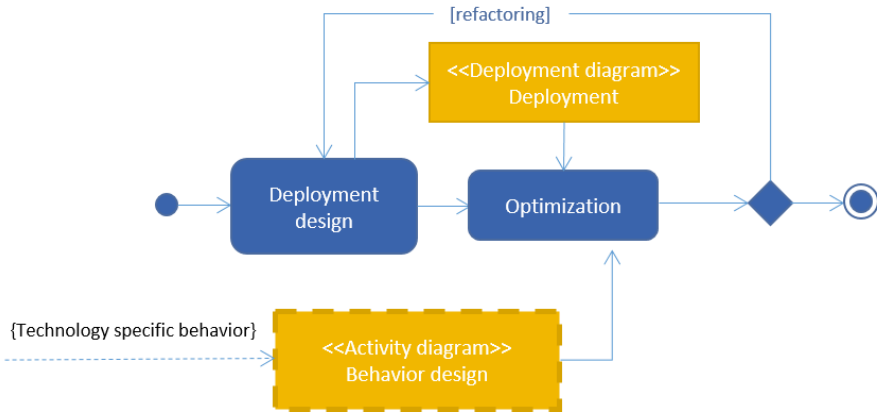


An extended description is available at: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#enhancement-tool.

3.3. Architecture Verification, Simulation and Optimization



Workflow for quality analysis at DPIM and DTSM



Workflow for quality analysis at DDSM

In the architecture simulation and verification scenario, *Developers* get from *System Orchestrators* a list of requirements about the system to construct, and want to study models of possible runtime environments in order to predict behaviors and performance for different implementation strategies. This work is similar to the one done by engineering or consultancy offices for civil or aeronautical projects, such as the construction of a building or the manufacturing of an aircraft. However, in these disciplines, the characteristics of the environment are imposed by natural physical conditions; whereas in software development, the characteristics of the runtime environment are negotiated with *Operators*. This use case is enabled thanks to the DICE/UML profile, the verification tool, the simulation tool and the optimization tool. Here are the stages to follow:

3.3.1. Step 1: Architecture Design

The first step of The DICE Methodology for the architecture verification and simulation scenario—the **DIA architecture design** (Figure 4)—consists in modeling a solution in terms of software components (or software nodes) serving and/or being served by other software components. For this activity, the modeling language endorsed by DICE is the UML Object diagram. The design can be accomplished

both in a platform-independent and technology-specific manner by using respectively the DPIM and DTSM profiles. At the DPIM level, components are stereotyped as source nodes, storage nodes, computation nodes or channel nodes; underlying technologies and communication protocols are not yet determined. It is only at the DTSM level that *Developers* can state, for instance, that a given storage node is actually a Cassandra cluster.

Once an architecture is envisioned, it is indispensable to classify the services thereof as internal or external. An external service must be provisioned in the runtime environment by *Operators*, while an internal service must be programmed by *Developers* and deployed into it afterwards. An external service can be, for example, a software open-sourced on the Internet or subcontracted to a company. This partitioning of services takes place during exhaustive deliberations between *Developers* and *Operators*. In other words, the DICE methodology innovates by integrating *Operators* into the development lifecycle from the beginning to the end of the project.

3.3.2. Step 2: Verification

Developers can invoke the verification tool to check that the services satisfy some invariant properties (Figure 4, **verification**) related to the temporal evolution of the application. Two kind of verification analysis are available in DICE that depend on the technological framework adopted by the developers to implement the application. Apache Storm and Apache Spark, the latter in its batch version, are the baseline technologies considered for verification in DICE. Storm applications are generally implemented to compute operations on streams, i.e., infinite sequences of tuples produced by a source of data. The Storm developer can verify the presence of bottleneck nodes in the deployment that saturate their memory. This analysis is relevant, since the presence of nodes that cannot process the input workload through a timely computation might cause a performance drop of the entire application. The verification is performed at the DTSM layer by checking the existence of an execution of the topology which leads to an unbounded growth of some bolts' queue. The inputs for the Storm verifier are:

1. A DTSM diagram;
2. The parameter values of all the spouts and bolts in the topology modeled by the DTSM (e.g., the throughput of the spouts and the time to process a tuple for bolts);
3. A positive discrete constant measuring the number of discrete time positions that define the length of the executions analyzed by the verifier (the user can consider that for each discrete time position an event in the topology can occur).

More details and all the required parameters are available in deliverable in D3.5 - Verification tool - Initial version and the implemention of the DICE Verification Tool (D-VerT) at https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#verification.

Verification of Spark application is designed to allow the DICE Developer for verifying temporal properties of batch applications implemented in Spark. Batch applications inherently produce finite executions, which yield an outcome given a certain (finite) input by iteratively applying complex transformations on data (that are stored into the so called RDDs). The Spark Developer can verify whether a finite execution of the application can be realized at runtime in a timely manner, i.e., within a given deadline defined at design time. The execution of Spark applications can be represented on the basis of the operational workflow of the data (Spark RDDs) that the user designs and implements. In Spark applications, an execution instance is represented through a Directed Acyclic Graph (DAG) that the Spark engine computes before the execution of the application. The Spark engine first decomposes the execution flow into various independent execution entities, called stages, that are composed of a set of tasks on the dataset. Each stage constitutes a node in the application DAG and the edges of the graph determine the precedence relation among the stages and, therefore, among the operations on data. The verification is performed by checking the existence of an execution of the Spark application which meets the deadline. The negative response entails that all the executions of the application cannot be completed

within the estimated time; otherwise, there is an appropriate scheduling of the tasks for the available computational resources that allow a timely execution of the application.

The inputs for the verifier are:

1. The operational workflow of RDDs (activity diagram);
2. The parameter values of the infrastructure running the Spark application (e.g., estimated time to complete a task with a certain machine);
3. A deadline which bounds the time span of the application.

Further details can be found in Deliverable 3.6 - Verification Tool - Intermediate version.

3.3.3. Step 3: Design of the DIA Behavior

In order to proceed with further evaluations of the quality of the DIA, in particular the evaluation of the performance and the reliability of the DIA, it is necessary that the user provides a design of the system behavior. In fact, the scenario-based performance evaluation is the way to traditionally carry the computation of performance metrics.

The behavioral design of the DIA is represented through UML behavioral and structural diagrams. In particular, Activity or Sequence Diagrams for the former and UML Deployment diagrams for the latter. The behavioural design is enhanced with stereotypes from DICE DPIM or DTSM profiles.

In the Activity and Sequence diagrams, the user specifies the operations performed by the DIA, their relative ordering, and concepts that are typical in the description of workflows, such as the execution in parallel of a set of operations, conditional execution of operations, iterations over a set of operations, etc. The utilization of DICE profiles allows to include information that is relevant for the performance and reliability with which the DIA will execute. Some examples of pieces of relevant information are: probabilities of execution failures, execution times of single operations, configuration characteristics of the Big Data technologies used, quantity of jobs submitted to these technologies, etc.

In turn, the structural part of the DIA design is also stereotyped using DICE DPIM or DTSM profiles. These diagrams model the assignment of the behavioural software artifacts previously defined into a simulated environment. Here, the DICE profiles allow the user to define properties of the resources used by the DIA, such as their quantity or the typical time to failure of the machines where the DIA executes.

With this information, the simulation tool can proceed with its execution (see Figure 4, **service behavior design**).

3.3.4. Step 4: Simulation

The simulation tool is able to produce performance and reliability results for a given DIA, in fact starting from the behavioural design of such DIA (Figure 4, **simulation**). In particular, the simulation tool can compute specific metrics for performance and reliability, both at the DPIM and DTSM levels. When the tool is launched, the user graphically interacts to specify the kind of analysis to perform, i.e., performance or reliability.

For performance analysis, the metrics provided refer to the service time, throughput and utilization. Then, the user can define using the DICE profile annotations that s/he is interested in assessing the expected response time of executions of the DIA, its throughput, or the utilization of resources used by the DIA. With this information, the user decides whether the current design satisfies the DIA performance requirements. The performance result regarding the utilization of resources provides the user with information to infer the location of the bottleneck of the DIA executions, or whether there is an under-utilization of resources and therefore DIA may waste them.

In case that the user had left some information in the profiled UML models as variables, this is the moment to specify their actual values. Variables in the UML models are useful for *what-if* performance analysis of the design, i.e., the user wants to know the system performance under different configurations. Some examples of these variables are: the number of computational resources that the DIA can use -ranging for instance from 2 to 5-, the quantity jobs submitted to the DIA -ranging for instance from 5 to 20-.

After these phases of specifying the kind of analysis and configuring the actual values for the simulation, the user can press the *Run* button in the graphical interface and execute the Petri net based DIA simulation. When the execution finishes, the user receives the performance results of the DIA simulation. The result produced by the evaluation is processed to generate a tool-independent report with the assessment of performance metrics. He can then proceed to the next step being informed of the expected performance, or modify the DIA design if the results do not satisfy the DIA performance requirements and/or expectations.

For reliability analysis, the process is analogous: first the user specifies the kind of reliability metrics -such as probability of failure of a concrete execution or the mean time to failure of the DIA- and configures the actual values of the information that was possibly left as variable in the design. Then press *Run*, the execution starts and the reliability results of the simulation are processed provided as a tool-independent report. Then, the user decides whether to proceed with the next step or to modify the design in case the results do not satisfy the DIA reliability expectations.

More details are available at: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#simulation-tool.

3.3.5. Step 5: Optimization

The step “architecture optimization” (Figure 5) consists identifying an minimum cost deployment providing a priori performance guarantees through the Optimization tool. The Optimization tool takes a set of DTSMs and reorganizes the infrastructure and the technologies therein to find an optimal architecture. The Optimization tool supports Storm, Hadoop and Spark applications (Spark limited to batch jobs).

The tool complements the Simulation tool and supports an automatic exploration of the solution space based on a wizard. This tool is invoked as follows:

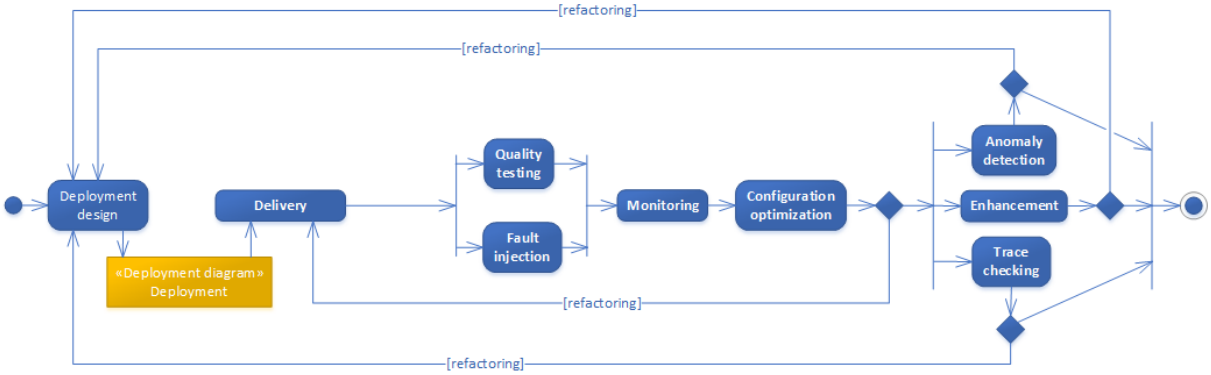
1. The Designer selects a set of DTSMs (including a DICE activity diagram) one for each target deployment annotated with the service demand of the DIA for the candidate configuration;
2. The Designer selects a DICE Deployment Diagram that will be updated with the final solution found;
3. The Designer specifies QoS constraints (cluster maximum utilization for Storm, deadline for job executions for Spark and Hadoop);
4. At this stage, the Optimization tool runs the Model-to-Model transformation tool (part of the simulation tool) to derive the underlying performance models;
5. Next, the Optimization tool explores the design space and runs QN/PN simulation to evaluate performance metrics of the candidate solutions;
6. Finally, the Optimization tool outputs the DICE Deployment Diagram of minimum cost, specifying the type and number of VMs for target deployment. Such model can then be deployed to the runtime environment by relying on the DICE deployment tool (see the next Section).

More details are available at: https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#optimization and in Deliverable 3.9 - Optimization Tool - Final version.

3.4. DevOps Delivery Lifecycle

Let us now consider the situation where a development team wants to automatically without the support of *Operators* deploy a *Big Data Application* and receive feedback on the application's behaviour in the pre-production environment. This scenario assumes the application is ready: the *Developers* already know which technologies must be present in the runtime environment.

Several of the steps in the lifecycle (Delivery, Quality testing, Fault injection, Configuration optimization) involve manipulating the test-bed or the pre-production environment into setting up the *Developers'* DIA and performing various operations on top of the DIA. These operations can all be done manually through the tools' front-end. However, to make the lifecycle truly DevOps, we describe the scenario assuming that a Continuous Integration service is in place to automate all the steps that should be performed unattended.



DICE workflow for DevOps

3.4.1. Step 1: Deployment Design

Developers start by drawing DTSM Object Diagram describing the technological services on which the Data-Intensive Application depends, without imposing infrastructure and platform choices. This phase is the **runtime environment design** also known as **Deployment design** (see Figure 5 and modelling procedure in Sec. UML Modelling). The tool supplied for the environment design is the technology-specific part of the DICE/UML profile available in the DICE IDE. This tool is invoked as follows:

1. Create a new or open an already existing DICE project;
2. Create a new empty DTSM Object Diagram using Papyrus and DICE UML Profile;
3. Draw the model.

Once Developers are in accord on the technological point of view, they have to discuss deployment concerns. This stage, named deployment design (Figure 5), ends with the definition of DICE Deployment Diagram that should answer every deployment question, whether it be about infrastructures, platforms and so on. The DICE IDE has a deployment-specific UML Profile, which is used this way:

1. Open an already existing DICE project containing a DICE Object Diagram;
2. Create a new DICE deployment diagram. A form is displayed;
3. Select the DTSM Object Diagram to refine and fill the form;
4. Click "finish". The technologies mentioned in the DICE Object Diagram are automatically loaded into the new DICE Deployment Diagram - the DDSM;
5. Complete the model (if needed).

Although it is also possible to draw a DICE deployment diagram from scratch, it is nevertheless better to begin with a DICE Object Diagram in order to benefit from trace checking - this will also allow the

straightforward use of the DICE-Rollout tool part of the deployment service to quickly put together a deployment blueprint for that DICE DTSM diagram.

3.4.2. Step 2: Delivery

Once the optimal architecture and configuration are produced, it is time to prepare the runtime environment and deploy the application into it. *DICE Delivery Tools* enable a simple way to continuously deploy the DIA in a testbed. The Continuous Integration (CI) component makes sure that the subsequent steps in the methodology run automatically at prescribed times or whenever the developers update the DIAs. Here, we assume that an *Administrator* has created CI projects for triggering processes downstream: Quality Testing, Fault Injection, Configuration Optimization or just a deployment for manual validation of the DIA.

The DICE Deployment Service creates the DIA's runtime from a TOSCA YAML document, which is generated from the DICE Deployment Diagram. The general sequence to perform that is as follows:

1. Open an existing DICE Deployment Diagram;
2. Use DICE Rollout (the model-to-text transformer) to transform the DICE Deployment Diagram into a TOSCA blueprint;
3. Prepare any of the target-tool-specific configurations and resources (see steps below for details);
4. Commit the application, TOSCA blueprint and any additional files from the previous two steps into the VCS (e.g., Git or Subversion);
5. A Continuous Integration job will pick on the update and automatically use the Deployment service to deploy the application and set up all its execution environment (i.e., Storm, Zookeeper, etc.). It will also get automatically registered with the DICE Monitoring Service;
6. The DIA will now be running in the testbed. Any supported services will send their metrics to the DICE Monitoring Service;
7. Depending on the configuration of the CI project, the tools described in the subsections below will start;
8. Open the Continuous Integration's job to access any output information from the DIA's deployment, such as services' URL, as well as any tool outcome.

Note that the CI has to have been configured by the *Administrator* to use a specific deployer's container, i.e., a virtual unit, which can take up to one deployment of the DIA. This ensures that the deployer cleans up any previous deployments assigned to the same container.

3.4.3. Step 3: Fault Tolerance & Quality Testing

Testers want to evaluate the performance of a DIA. They also want to test its robustness when confronted with a hostile runtime environment. In the DevOps scenario, we expect that these steps run automatically, so in DICE this is ensured by the Delivery Tool's process in Step 2.

The *Quality Testing* tools subject the deployed DIA with a varying volume and velocity of the input data. The DICE Monitoring Tool measures DIA's runtime metrics, which the Quality Testing tool can use to produce a report on the application's performance parameters such as throughput and response times.

In the **Step 2**, the following tool-specific sequence needs to take place:

1. (Part of **Step 2**) Developer uses the Quality Testing tool's library to prepare a test mode version of the DIA;
2. (Part of **Step 2**) Developer prepares the load generation scenario input and configuration (data volume, rate, test duration).

When the CI executes the Quality Testing, the sequence is as follows:

1. The workload to be injected in the system is prepared using QT-Gen;
2. The QT-Lib custom spout is added to the DIA code and the workload packaged with the application JAR file;
3. The application is deployed and QT-Lib performs the load generation;
4. Quality Testing collects the monitoring data from Monitoring Tool and stores a report;
5. Delivery Tools save the performance data report with the DIA's test execution;
6. The user visits the Continuous Integration's web interface to view the history of the performance testing.The user can access individual execution run's information as numerical results.

The *Fault Injection Tools* help the users create a hostile environment by triggering faults against the DIA:

- The application is deployed via CI tools onto the target cloud;
- The Deployment information is passed to the FIT after the application has been deployed, where the component names are set to corresponding faults;
- The FIT uses this information to determine the faults to enact across the VMs;
- The faults are enacted across the VMs and details stored within the generated logs;
- This information can be picked up by monitoring and fed back to the administrators/developers before the next deployment to be used to determine any points of failure.

3.4.4. Step 4: Monitoring

DICE Monitoring Platform (DMon) collects, stores, indexes and visualizes monitoring data in real-time from applications running on Big Data frameworks. It supports DevOps professionals with iterative quality enhancements of the source code and with the optimisation of the deployment environment.

Use one tool to overview all your environments and their log entries, front to back. APM and Troubleshooting all at once and leverage new valuable data to quickly and precisely understand areas of improvement within the DIA. DMon requires a lot of information about the currently deployed Big Data frameworks and the DIAs that are running on them. Because of this, the delivery service is also responsible for deploying and configuring DMon. It is also important to note that DMon has the ability to auto-detect what services run on each monitored node. The steps required to setup DMon are:

1. Register the nodes to be monitored (node endpoint, operating system, credentials);
2. Install monitoring agents on the nodes:
    a. If the deployment service is already configured and started the monitoring agents DMon will wait for the agent to heartbeat;
    b. Agents are automatically installed and started using parallel SSH connections to all monitored nodes.
3. Add Roles to each monitored node (YARN, Storm, Spark, Cassandra etc). Some roles can be autodetected by DMon;
4. Detect location of key services:
    a. Storm -> Nimbus, UI;
    b. Spark -> History Server;
    c. YARN -> History Server.
5. Set metric polling period;
6. Configure and start DMon core services (Logstash, Kibana and Elasticsearch);
7. Generate and apply configurations to agents and start monitoring. Application metrics versioning should also happen at this step.

It is important to note that all of these steps are done automatically within the DICE solution (using the DS).

3.4.5. Step 5: Configuration Optimization

Part of DevOps is also to make sure that these services and the application itself are configured to perform efficiently. The *Configuration Optimization (CO) tool* automatically tunes the configuration parameters of data-intensive applications. The DIAs are developed with several technologies (e.g., Apache Storm, Hadoop, Spark, Cassandra), each of which has typically dozens of configurable parameters that should be carefully tuned in order to perform optimally. CO tool enables end users of such application to auto-tune their application in order to get the best performance. Considering that each CO execution can run for several hours, we delegate the scheduling and running of the CO to the Continuous Integration (CI). In **Step 2** above, we therefore need to provide CO-specific settings:

1. **(Part of Step 2)** Select YAML file (generated at the Deployment Design step) containing the configuration parameters with default values;
2. **(Part of Step 2)** Optionally, specify the parameters of interests and the possible range of values for each parameters, also the experimental details e.g., exp. budget (i.e., number of runs), also address of services CO is dependent on.

When the CI executes the CO, the sequence is as follows:

1. CO runs a number of iterative experiments until budget is finished;
2. Optimum values are set by the CO tool directly to the YAML file and the DICE Deployment Diagram is updated correspondingly.
3. The developer can obtain the updated blueprint and the optimal configuration in the CI's web interface.

3.4.6. Step 6: Trace Checking, Anomaly Detection and Enhancement

During environment design, Infrastructure Designers and Developers can take advantage of the *Trace Checking Tool* to improve the design of the previous development iteration, if any. Indeed, the *Trace Checking Tool* allows them to verify whether runtime monitored data satisfies some properties. They have to write directly in the DTSM all the properties they desire to be respected by the technologies installed in the runtime environment. They will be notified as soon as the tool terminates its analysis. Then, they may redesign the DTSM according to the boolean answers they receive. Here is the procedure to perform trace checkings with the IDE:

1. Open an already existing DICE project containing a DTSM Object Diagram;
2. Open the dialog box of the trace checking tool. A form is displayed;
3. Fill the form;
4. Click the button "run";
5. Await the notification (a yes/no result for each property) of the trace checking.

The trace checking, anomaly detection and enhancement tools are runtime-to-design tools that analyse monitoring data and suggest improvements of models. To use the trace checking tool:

1. Load the UML Deployment Diagram (DDSM model);
2. A form is displayed where configuration arguments of TC is filled;
3. TC tool is run;
4. The user is notified whether the trace is checked (Yes/No response).

To use the anomaly detection tool:

1. User selects AD option from DICE menu;
2. A form is displayed where arguments for AD tool are to be provided;
3. AD tool is run;
4. Model is stored on DMon platform and user is notified about success/failure and probable cause of anomaly.

Enhancement tool aims at filling the gap between the runtime and design time by inferring the parameters of UML models from monitoring data provided by DICE Monitoring Platform and feeding results back into the design models to provide guidance to the developer on the quality offered by the application at runtime. *Enhancement tool* mainly accesses on DICE UML model on DDSM and DTSM levels. *Enhancement tool* contains two modules: DICE-FG and DICE-APR. DICE-APR contains two sub-modules: Tulsa and APDR. Tulsa is for M2M transformation (from UML model to LQN model) and the generated LQN model can be solved by LQN solver (e.g., LINE, lqns). APDR is for anti-patterns detection and refactoring. Enhancement tool is integrated with DICE IDE and it also has standalone version. To run the Enhancement tool in DICE IDE, user needs to install Matlab Compiler Runtime (MCR) R2015a and configure the runtime environment. User also needs to download the sample configuration files from Github page[1] or create them in current project. Here is procedure to perform Enhancement tool in DICE IDE:

1. Run DICE-FG:
    a. User sets input parameters of the *DICE-FG-Configuration.xml* for DICE-FG;
    b. Select the UML model and right click to invoke DICE-FG from a popup menu (Enhancement Tool - > FG);
    c. DICE FG parametrizes the target UML model by using statistical estimation and fitting algorithms.

2. DICE-APR:
    a. User sets input parameters of *DICE-APR-Configuration.xml* for anti-patterns detection and refactoring, e.g., Anti-Patterns boundaries.
    b. Select the UML model and right click to invoke DICE-APR from popup menu (Enhancement Tool - > APR);
    c. DICE-APR invokes the Tulsa to load the target DICE UML model (including Activity diagram and Deployment Diagram) and perform the M2M transformation to generate LQN model;
    d. DICE-APR invokes APDR to solve the LQN model for APs detection and shows the refactoring suggestions in console if AP is found.

[1] https://github.com/dice-project/DICE-Enhancement-APR/tree/master/Plugin/doc/Configuration%20Files

4. The DICE Methodology in the IDE

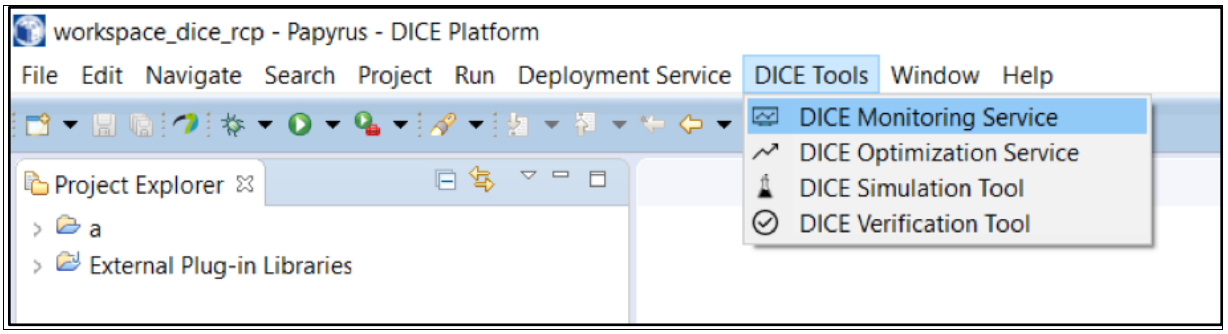
The DICE IDE offers the ability to specify DIAs through UML models. From these models, the toolchain guides the Developer through the different phases of quality analysis, formal verification being one of them.

The IDE acts as the front-end of the methodology and plays a pivotal role in integrating the DICE tools of the framework. The DICE IDE can be used for any of the scenarios described in the methodology. The IDE is an integrated development environment tool for MDE where a Designer can create models at different levels (DPIM, DTSM and DDSM) to describe data-intensive applications and their underpinning technology stack.

It initially offers the ability to specify the data-intensive application through UML models stereotyped with DICE profiles. From these models, the tool-chain guides the developer through the different phases of quality analysis (e.g., simulation and/or formal verification), deployment, testing and acquisition of feedback data through monitoring data collection and successive data warehousing. Based on runtime data, an iterative quality enhancements tool-chain detects quality incidents and design anti-patterns. Feedbacks are then used to guide the Developer through cycles of iterative quality enhancements.

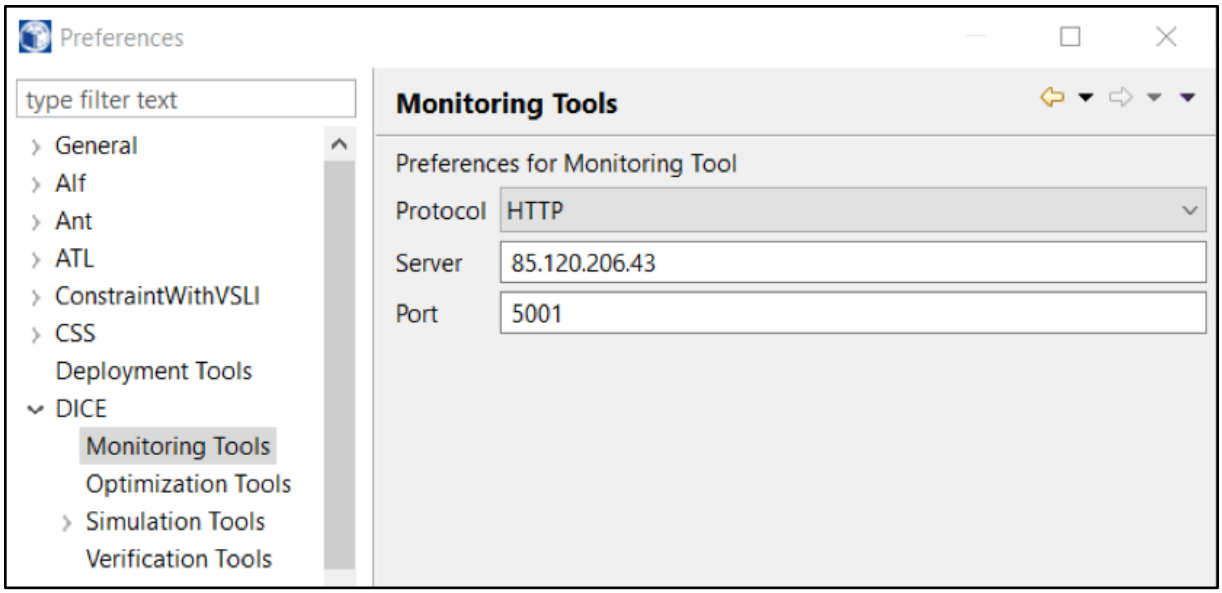
The DICE IDE is based on Eclipse, which is the de-facto standard for the creation of software engineering models based on the MDE approach. DICE customizes the Eclipse IDE with suitable plug-ins that integrate the execution of the different DICE tools in order to minimize learning curves and simplify adoption.

The DICE IDE has been customized with suitable plug-ins that integrate the execution of the different DICE tools in order to minimize learning curves and simplify adoption. Not all tools are integrated in the same way. Several integration patterns, focusing on the Eclipse plugin architecture, have been defined. They allow the implementation and incorporation of application features very quickly. DICE Tools are accessible through the DICE Tools menu.



4.1. DICE Tools Menu

The IDE also provides an abstract Eclipse Preferences page that allows the user to modify these properties. In this way, the external web server tool integration can be modified dynamically by the user if needed.

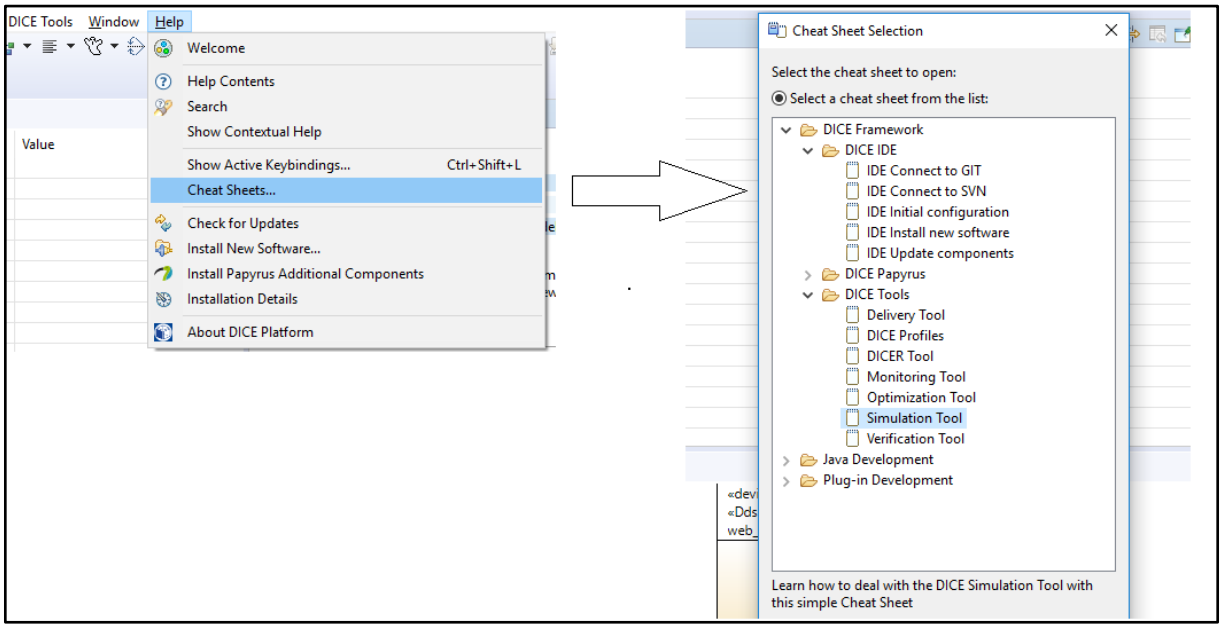


Example of Monitoring Tool external web service configuration.

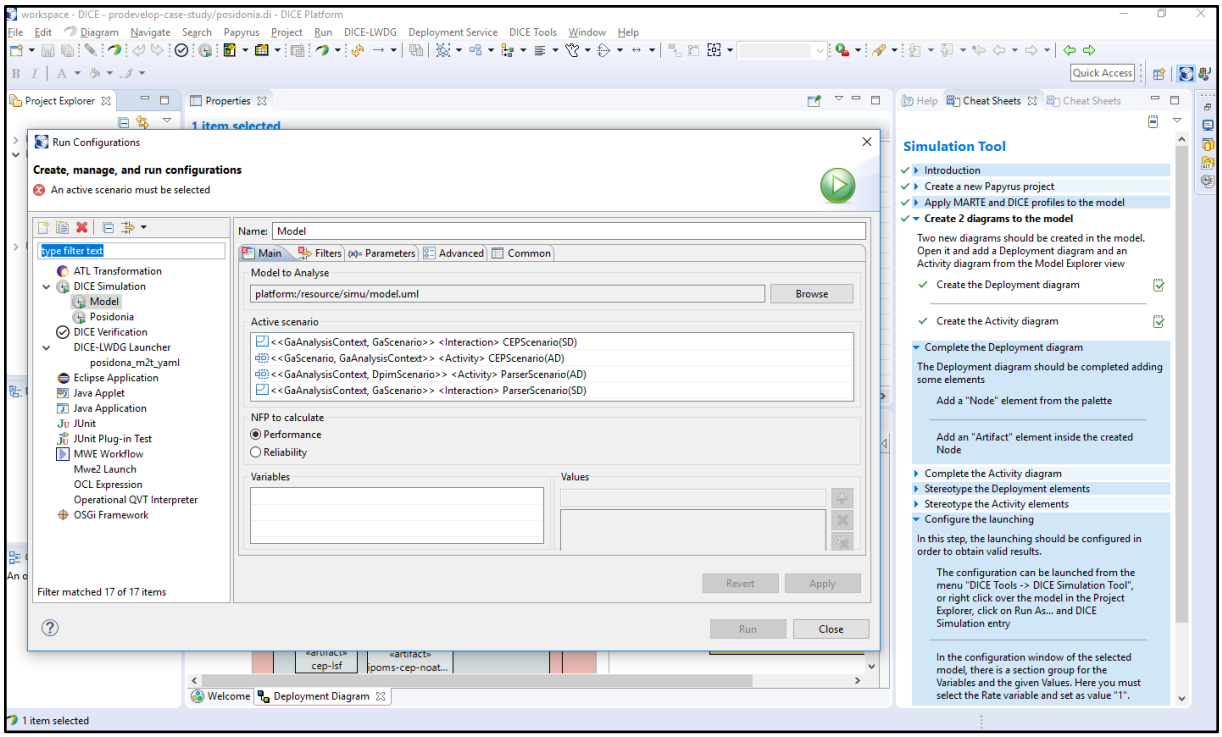
4.2. Cheat Sheets

The IDE guides the developer through the methodology, based on tools Cheat Sheets.Cheat sheets are a great way to guide users of your Eclipse plug-ins or Eclipse-based products through the steps they must follow to use your software. Cheat sheets let Eclipse users view interactive tutorials from within the Eclipse Workbench to learn how to perform complex Eclipse tasks. The DICE IDE provides cheat sheets (https://www.eclipse.org/pde/pde-ui/articles/cheat_sheet_dev_workflow/) to guide users in using DICE tools.

You can access to the Cheats Sheet through the Help menu, there are some generic cheat sheets for the DICE IDE and DICE Papyrus, and specific cheat sheet for the DICE Tools.



In the figure below you can see the simulation Tool Cheat sheet as an example. The cheat sheets contain a list of steps to follow in order to use a specific tool.



5. Annex 1 - Privacy-By-Design Sub-Methodology

The DICE consortium advocates and supports the massive use of data-intensive technology to process data of any source, type or structure. While, on the one hand, the DICE project did not foresee any explicit action towards addressing the privacy-sensitive nature of considerable subsets of that data, we are, on the other hand, aware and confident of the existence of several technologies, reference documents and methodologies that can avail the study and support of data-intensive, private-by-design technology. Quoting from The European Union Agency for Network and Information Security (ENISA) "Privacy and Data Protection by Design – from policy to engineering", privacy by design is defined as follows, in the context of DICE:

"The explicit design decision of taking privacy into account throughout the entire engineering process from earliest design stages to the operation and vice versa."

For the sake of this methodological specification for privacy in DICE, we chose to focus on:

1. Supporting the design of privacy concepts and solutions at data-intensive design levels (DPIM and DTSM, in the DICE parlance), inheriting from the state of the art;
2. Supporting the operation of privacy-by-design technology, extending the following DICE technology:

    a. *DESIGN-to-DEPLOYMENT: Extending the DICE Profile and DICER tools,* to Assist privacy-policy deployment in a semi-automated fashion;

    b. *DEPLOYMENT-to-OPERATION: Extending the DICE DtracT and Delivery Service tools,* to Support the monitoring of said policies;

    c. *OPERATION-to-RUNTIME-VERIFICATION: Extending the DICE DtracT tool,* to Verify that the clauses of said policies have not been violated at runtime, and releasing results **if and only if** this condition holds;

    d. *CLOSING-THE-LOOP: Extending the DICER and Delivery Service tools,* to Reporting privacy monitoring and verification feedback at design-time, for continuous architecting of privacy-by-design;

In the following, we define and exemplify how the above extensions work together as a coherent whole, and as part of the original research and practical contributions to the state of the art in privacy-by-design.

5.1. Outline

Our approach consists of 3 continuous architecting steps:

1. allowing a developer to set ABAC access control policies on the DICE architectural models (e.g. a UML component diagram) of a DIA;
2. propagating such policies using the DICER model-driven pipeline;
3. monitoring their validity at run-time leveraging on DICE trace-checking techniques.

Although several approaches to model-driven role-based access control exist (e.g., SecureUML, MTL [DtracT]) we learned that a much more granular way of modeling access policies is required in the context of Big Data. Moreover, existing approaches need to be combined with state-of-the-art trace-checking technology (see the DICE DtracT in the previous methodology sections) to enable quick and constant monitoring for the satisfiability of temporal-based access policies over very large system logs.

In our initial experimentations we observed that, although with several limitations and strong assumptions, our working prototype shows promise in supporting the iterative and continuous architecting process of offering privacy guarantees, right from the earliest phases of the software lifecycle. Finally, discussing prototype assumptions, limitations and initial results, we conclude that the proposed solution can serve as a starting point for future research and development. However much more effort is still required. On these premises, we define a tentative research roadmap, by identifying some of the key challenges to be addressed in the future.

5.2. Research Solution

Figure 1 outlines our solution architecture. The architecture comprises two main building blocks, a modeling environment and a runtime environment.

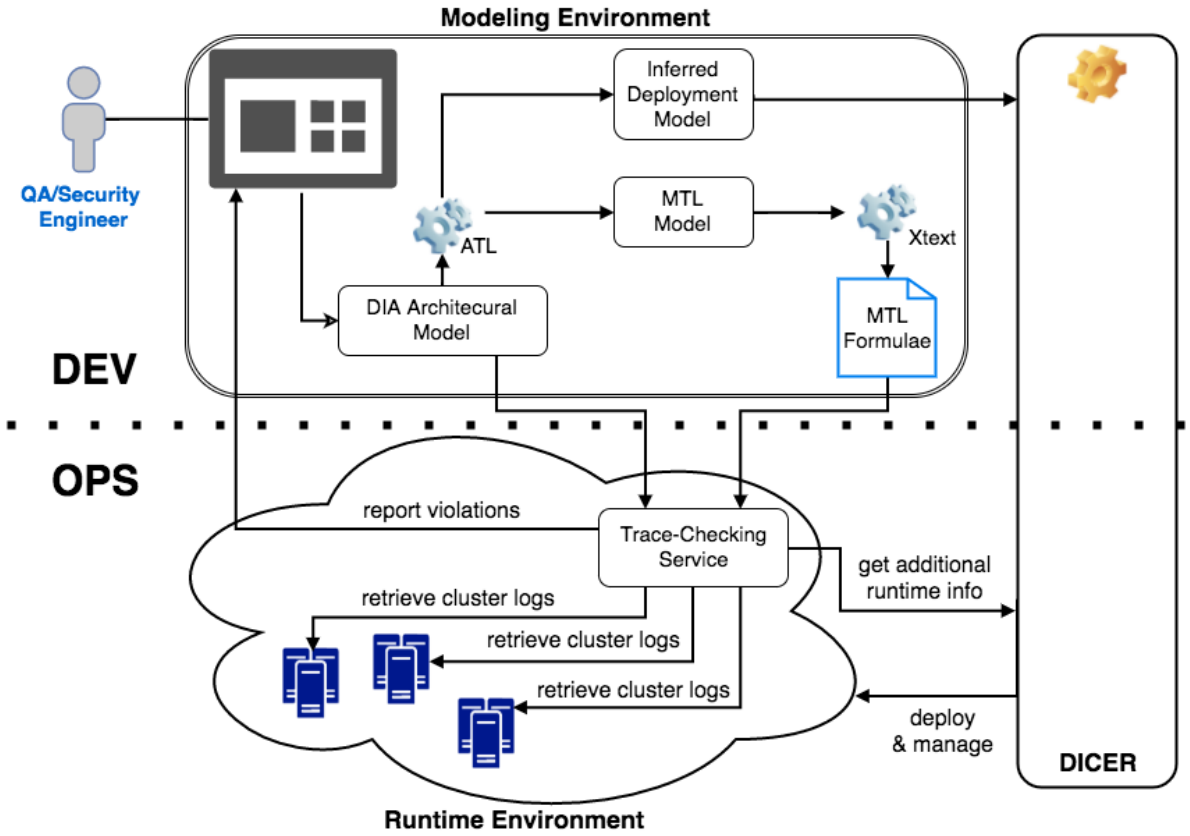


Figure 1. Solution Architecture

On one hand, at design-time a user (e.g. a QA or Security engineer) defines the architectural model of a DIA en- riched with access control policies. The automated model- driven pipeline generates from the defined model a set of Metric Temporal Logic (MTL) formulae expressing the desired policies in terms of temporal constraints over system events. MTL is a temporal logic with the ability to express metric, i.e. quantitative, timing requirements.

On the other hand, the generated MTL formulae are then deployed on Trace-Checking Service, which periodically checks over traces of events their validity, to ultimately monitor and report violations of the defined policies. To address these periodic checks in an automated fashion, in our research solution we leverage DICER, our DevOps tool introduced in our previous work, enabling the model-driven continuous deployment of DIAs on the Cloud. A default DICER-specific deployment model is inferred from the DIA architectural model using an ATL (ATLAS Transformation Language) model-to-model transformation, to automatically deploy our runtime environment, which consists of the Trace-Checking Service, along with all the Big Data platforms required to operate the DIA. Once the deployment is completed, the Trace-Checking Service is initialized with the DIA architectural model along with the generated set of MTL formulae.

Stemming from the architectural model, the Trace-Checking Service is able to identify and to locate all the information needed for building suitable traces, so that the installed MTL formulae can be checked over them using an appropriate trace-checker. The latter is an efficient large-scale trace-checker of MTL formulae introduced in the DICE Trace-Checking service [DtracT], which allows us to quickly analyze very large traces and thus to quickly monitor and report violations of granular access policies.

5.2.1. Modeling DIAs with Granular Access Control Policies

In our scenario, a QA engineer who has to perform, among the others, privacy and security analysis of DIAs, can design, through the Modeling Environment (Top-left of Fig. 1), an architectural model of a DIA, conforming to the underlying meta-model, whose core part is shown in Figure 2.

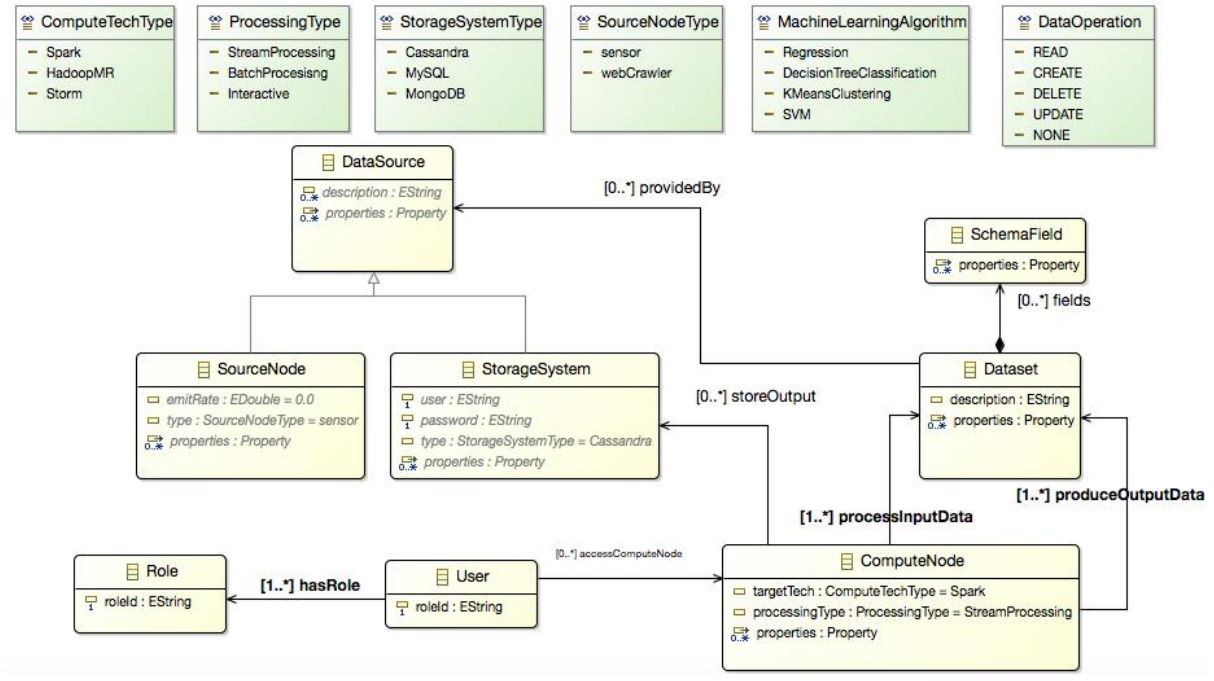


Figure 2. DIA Architecture-Level Meta-model (DPIM extract).

In this model a ComputeNode is represented as a black box containing a specific computa- tion (e.g. a machine learning algorithm, a query, etc.) which works on a set of input Datasets to produce a set of output Datasets. A Dataset is structured as a set of records with a number of SchemaFields. A ComputeNode can be implemented according to different processing types (e.g. batch, streaming) and using different Big Data middleware, e.g., Spark, Storm or Hadoop. Moreover the user can model the DataSources providing the various Datasets. These can be SourceNodes, such as sensors or web pages, or StorageSystems, such as Cassandra or MongoDB, which also provide a persistence service. A ComputeNode is owned by a User, which can plays different Roles (e.g. admin, analyst, etc) within the modeled system.

The user can then augment the obtained model with granular access control policies. All the concepts shown in Figure 2 are specializations of the generic DIAElement concept. A Permission associates its owner, which can be any DIAElement, with a set of ActionTypes, or operations that are accessible on a protected element, which in turn is a DIAElement. The freedom coming from binding pairs of generic DIAElements makes the modeling language highly flexible, since, depending on the owner and the protected element, we can specify different types of access policies.

Permission includes two attributes for specifying its validity start and end times, giving the possibility to set time-based access control policies, a key feature of our framework. In our solution we refer to the specific case in which access is granted only within a certain time interval, even though the more general MTL language can express much more elabo- rated timing requirements.

Finally by setting Properties on DIAElements, in terms of ⟨key, value⟩ pairs, we can further characterize the owner of a Permission and its protected elements, making the policy more and more specific (i.e. granular). For instance a ComputeNode could have the Property ⟨Location, Italy⟩, which can be used to restrict the access based on the location of the node.

Once the modeling activities have been finished the user can activate the model-driven transformations pipeline. First, the DICER tool can generate a default deployment model conforming to the DICER modeling language. Essentially, DICER consumes the DIA architectural model to find all the modeling elements that require certain platforms to be available at runtime. For instance, if there are one or more ComputeNodes whose target technologies is Spark, an instance of the Spark execution engine has to be available at runtime in order to execute them. The same applies for each StorageSystem containing one or more Datasets. The output is a complete deployment model, with default configurations, representing the runtime environment as illustrated in Figure 1. DICER can at this point automatically deploy such model.

Second, once the runtime environment has been deployed, the next step is to initialize the Trace-Checking Service, which takes as input the designed DIA Architectural Model and the set of generated MTL formulae. The set of MTL formulae is generated at design time so that the user is able to validate or even refine it before it is sent to the Trace- Checking Service. The generation of the MTL formulae is done using a combination of a model-to-model and a model-to-text transformations, developed in ATL and XText respectively. This second model-to-model transformation produces, from the DIA Architectural Model, an MTLModel, i.e. a set of MTLFormulae conforming to the meta-model that we derived from the MTL grammar.

Finally the user can send the final DIA Architectural Model and the generated set of formulae to the running Trace-Checking Service.

5.2.2. What Happens at Runtime?

At runtime the Trace-Checking Service is responsible for monitoring and reporting back to the Modeling Environment access control policies' violations that might happen. By looking at the DIA Architectural

Model, the Trace-Checking Service associates each permission (and its corresponding MTL formula) to a Driver process.

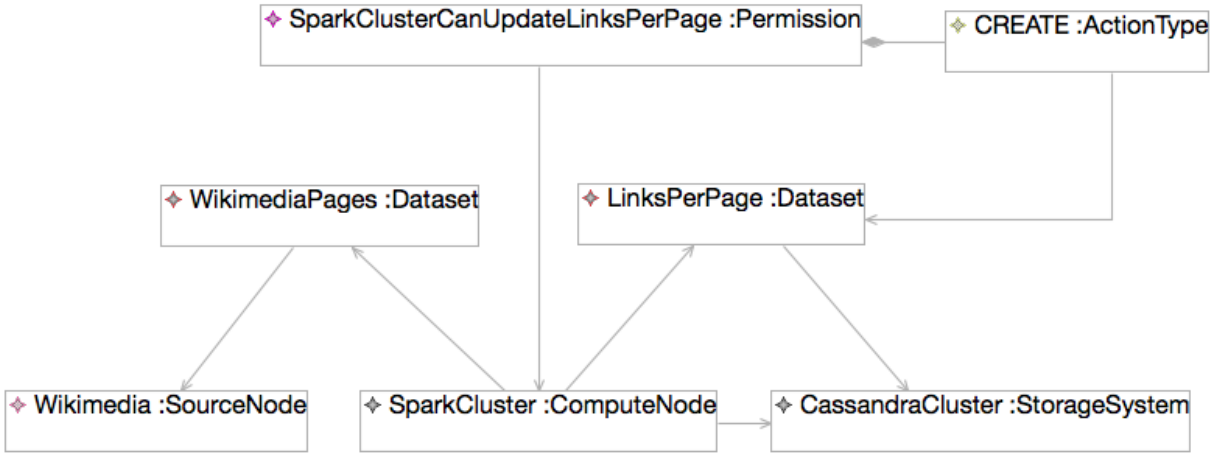


Figure 3. DIA architectural meta-model, a sample instance.

As a black box, the role of each Driver is simply to periodically update a trace, conforming to a specific format that can be read by the Trace-Checker, with necessary information for checking the satisfiability of the MTL formulae (and, in turn, the access policies) assigned to the Driver. Every time the trace is updated, the Driver also run the Trace-Checker to check if the MTL formulae are still satisfied. Finally, if there are violations, the Driver reports them to the Modeling Environment. The periodicity of the described process can be set by the user within the DIA Architectural Model. Each Driver conceptually represents the ability to check specific types of permissions. For instance a Permission granting a ComputeNode with the possibility to perform the READ action on a Dataset stored into a StorageSystem, can be checked as long as there is a Driver installed in the Trace-Checking Service dedicated to such kind of Permission (e.g. an application accessing a given dataset), which is able to retrieve all the necessary information from the available information systems (e.g. system logs, database tables, websites, etc). A different Permission could grant a User, instead of a specific application, with privileges for reading a specific SchemaField of a Dataset, instead of the Dataset as a whole. In this second case the Driver responsible for managing such policy at runtime will behave differently with respect to the first one, both in gathering all the necessary pieces of information and in building the trace. Moreover, the DIAElements involved in a given permission could be implemented using equivalent technologies (e.g. a streaming ComputeNode could be either a Spark or a Storm application), thus the responsible Driver should be realized in such a way as to be technology-agnostic and extensible.

If at some point the user wants to change a permission, she can re-traverse the model transformations pipeline and re-initialize the Trace-Checking Service. In this way our solution enables continuous architecting of DIAs in a flexible way.

5.2.3. DICE Continuous Architecting for Privacy-by-design: Example Scenario

In this section we are going to showcase our approach with a simple example, covering both the modeling aspects and the runtime behavior. As previously stated, the proposed modeling language is flexible enough to allow the definition of different kind of access policies, between different pairs of modeling elements. Each type of permission need to be managed appropriately by a specific Driver installed in the Trace-Checking Service, which also has to support the specific technologies involved in a given permission. In particular in our scenario we wanted to check permission of the type:“applications

running on the middleware M can execute the CRUD operation O on the table T contained in the datastore D only during the time interval from T1 to T2".

As previously stated, this is just one of the possible access policies that we could express. We focused on data access since this kind of access is particularly relevant in the context of guaranteeing privacy, that is the primary goal of our solution. Figure 3 shows the model obtained by instantiating the DIA architectural meta-model showed in Fig. 2. In this example a single Spark ComputeNode (e.g. Spark applications running on a Spark cluster) accesses a Dataset of web pages from a SourceNode, that is the Wikimedia website, in order to perform various data analysis and produce aggregated data. For instance, a DIA could produce a Dataset called LinksPerPage, containing for each webpage its external references, that has to be stored into a StorageSystem, for instance a Cassandra cluster. The software architect decides, by modeling an appropriate Permission, that the Spark ComputeNode is allowed to perform the CREATE operation on the LinksPerPage Dataset only during the interval ⟨T1, T2⟩. From such a model, the model-driven pipeline generates the following MTL formula:

$$Update(SparkCluster,LinksPerPage,CassandraCuster) \Rightarrow P[T1,T2]START$$

The Driver that is responsible for checking this specific type of data access permission also needs to include Cassandra among the supported datastores. In particular we found that Cassandra offers the possibility to enable a feature called Probabilistic Query Tracing, which essentially tracks the execution of each query. By enabling this feature we are able to periodically ask Cassandra for a time-ordered list of executed query, which also reports for each query the IP address of the VM from which the query was issued. We can then query the DICER service to resolve the cluster such VM belongs to, e.g., our Spark cluster rather than an- other distributed middleware available in the deployed Big Data infrastructure. By parsing the executed queries and resolving the query sender, our Driver is finally able to write traces like the one shown in Listing 1. By asking the Trace-Checker to verify the satisfiability of the MTL formula over the created trace, the Trace-Checking Service can successfully evaluate and report if the associated permission has been violated. In the provided example, assuming that (1) is instantiated with values T1 = 15 and T2 = 20, Listing 1 violates the permission as the update event with timestamp 22 happens outside the allowed time interval.

Listing 1: A piece of trace written for checking the example data access permission.

```
0  START
2  Elem(SparkCluster)
5  Elem(CassandraCluster)
7  Elem(WikimediaPages)
12 Elem(LinksPerPage)
15 Read(SparkCluster, WikimediaPages, CassandraCluster)
16 Update(SparkCluster, LinksPerPage, CassandraCluster)
22 Update(SparkCluster, LinksPerPage, CassandraCluster)
25 Create(SparkCluster, LinksPerPage,CassandraCluster)
30 Read(SparkCluster, WikimediaPages,CassandraCluster)
```

5.2.4. Conclusion and Research Roadmap

In the previous sections we illustrated our approach to the problem of enriching the design of DIAs with the definition of privacy policies, and we highlighted a first prototypical solution, enabling the a-posteriori check of such policies by means of trace-checking techniques.

This section elaborates on the main challenges and possible next steps towards the improvement and extension of the presented approach. The goal is to obtain a more complete solution to guarantee ABAC policies for DIAs.

As highlighted in the previous sections, one of the main limitations of our approach concerns the fact that policy violations can only be detected a posteriori by means of trace-checking. For this reason, a key improvement would be enabling the enforcement of ABAC policies. The problem is non trivial and needs further investigation.

Another possible improvement in that respect consists of adding some automatic mechanisms to deal with the detection of policy violations. For example, whenever a violation is reported by the Trace Checking Service, the system could "react" by adopting specific countermeasures, promoting the "continuous improvement" of the application.

One way to prevent violations is by enabling the secure deployment of DIAs. Our next step will be to support this approach at the DICER end. On one hand we will support the deployment of access policies at the data source level. On the other hand, we plan to support the deployment of secure datasets, meaning that private data are stored into protected datasets, that are accessible only by high privileged and trusted actors. The advanced version of DICER will produce deployment blueprints, that include actionable security and privacy policies.

A next major challenge is to support the whole DIA's development life-cycle, by extending the privacy analysis also to the design phase, applying, for instance, a model-based formal verification approach. Such extension would support the early detection of privacy-related design flaws and would foster the DevOps approach, by combining design-time and runtime analysis.

Furthermore, in order to at least partially overcome a ma-or limitation of our prototype, i.e. the amount of information from multiple sources that it has to locate and retrieve at runtime, we plan to adopt specialised tools, such as the DICE monitoring tool (see previous sections), to collect and index some the necessary logs and data.

In conclusion, we argue that the problem of guaranteeing data privacy in the context of Big Data becomes critical and much more difficult to be addressed than in traditional data-intensive systems, mainly because of a) the huge amount of sensitive data that are captured and b) the new and complex technological landscape. This context requires new techniques to reason about data privacy, in order improve the design of privacy-aware solutions. On one hand, it is necessary to consider privacy issues from multiple perspective simultaneously, e.g. at the database and at the application level, and to adapt traditional privacy techniques to the current technological context. On the other hand, data privacy needs to be considered as a primary non-functional aspect of digital systems and privacy enhancing solutions must become much more pervasive than in the past.

Bibliography.

[DtracT] M. M. Bersani, D. Bianculli, C. Ghezzi, S. Krsti´c, and P. S. Pietro. Efficient large-scale trace checking using mapreduce. In Proceedings of the 38th International Conference on Software Engineering, pages 888–898. ACM, 2016.

ANNEX 2 - Addressing Containerisation in the DICE Profile and DICER Tools

Containerisation technology is having a profound impact as to how software applications based on the clouds are being architected, infrastructured, and provisioned for continued service quality. In the context of this impact, we investigated whether the additional modelling of container technology may further structure the infrastructure design and continuous architecting efforts sustained by the DICE Profile and the DICER tools.

To address this challenge we operated as follows: (a) we mapped the definition of containers as concepts to be modelled with previously existing MARTE-DAM or standard UML constructs; (b) we evaluated the addition of transformation rules to produce the concept of containers in the context of TOSCA infrastructure-as-code blueprint descriptions; (c) we investigated the industrial modelling and expected usage with insight and counsel from our case-study owners in DICE.

As a result of step (a), we observed that no concept at DPIM and DTSM level can address the construct represented by containers. Conversely, we also observed that constructs and inferences we defined and tested in the context of PRO's need for *ad-hoc generic deployment nodes can and in fact addresses the modelling of containers as part of DICE DDSM* UML Profile. Therefore, we concluded that the DICE DDSM profile is already able to fully support the modelling of containers as part of infrastructure designs (DDSM).

As a result of step (b), and in continuation with our collaboration with XLAB, we structured ad-hoc transformation rules to produce the conceptual notion of containers into full-fledged TOSCA specifications. However, while designing such transformation rules, we observed that TOSCA YAML CSD 1.0, upon which the current version of DICE DDSM is based, cannot support the modelling of containers as standard concepts. Therefore we concluded that, any support to containerisation technology in the DICE DDSM technical baseline would deviate sensibly from the DoW objective of contributing to standards (i.e., TOSCA YAML 1.1 CSD, upcoming) with respect to Data-Intensive Technology in the cloud. We concluded that, *on one hand, transformation rules need only minor adjustment to address containerisation at the TOSCA modelling and infrastructure-as-code level, on the other hand, any technical extension to the DICER tool or DICE delivery pipeline would work against our project objectives and therefore shall not be investigated further, beyond a simple proof-of-concept experiment.*

In the scope of step (b), we observed that our industrial partner PRO is exploring the usage and context of containerisation for purposes other than data-intensive technology. Their usage is aimed at addressing infrastructure resilience and service continuity. *We conclude that further continuation down this line should investigate the support of data-intensive resilience by means of containers in the appropriate technical analysis WPs (WP3,WP4,WP5), hopefully with DDSM concepts and constructs.*

