Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

Architecture Definition & Integration Plan (Final Version)

Deliverable 1.4

Deliverable:	D1.4
Title:	Architecture Definition & Integration Plan (Final Version)
Editor(s):	Vasilis Papanikolaou (ATC)
Contributor(s):	Giuliano Casale (IMP), Tatiana Ustinova (IMP), Marc Gil (PRO), Christophe Joubert (PRO), Alberto Romeu (PRO), José Merseguer (ZAR), Raquel Trillo (ZAR), Matteo Giovanni Rossi (PMI), Elisabetta Di Nitto (PMI), Damian Andrew Tamburri (PMI), Danilo Ardagna (PMI), José Vilar (ZAR), Simona Bernardi (ZAR), Matej Artač (XLAB), Daniel Pop (IEAT), Gabriel Iuhasz (IEAT), Youssef Ridene (NETF), Craig Sheridan (FLEXI), Darren Whigham (FLEXI), Marcello M. Bersani (PMI)
Reviewers:	José Ignacio Requeno (ZAR), Li Chen (IMP)
Type (R/P/DEC):	R
Version:	1.0
Date:	31-January-2017
Status:	First Version
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2016, 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:	Unversidad 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 current document is an extensive and updated version of "*D1.3 - Architecture definition and integration plan (Initial version)*". A descriptive e update on changes related to both technical and business DICE requirements is presented, highlighting changes per Work Package. Moreover, the final plug-in architecture for the DICE framework, that takes into account all previous brain-storming elements and architectures, presented in D1.3, is presented. Changes and deviations that took place regarding DICE tools and their main roles in the updated DICE architecture are also thoroughly highlighted. Adding to the above, positioning DICE tools in the methodology is also presented by taking account both design-focused and runtime oriented tools and last but not least, an extensive update on the integration plan (so far) is presented while future actions and integration activities are being defined and positioned in a concrete timeframe.

Table of contents

List of figures

List of tables

1. Introduction

1.1. Overview:

This second iteration of *"D1.3 - Architecture definition and integration plan (Initial version)"* includes the upgraded and final version of DICE Architecture and integration plan, to be implemented until the end of the project. Moreover, both the tool owners and demonstrators have updated their business and technical requirements as well as DevOps practices and technologies used in order better meet the desired outcome and objectives of the project.

Regarding the architecture, the final plug-in architecture for the DICE framework has been implemented which takes into account both requirement and needs set by tool owners regarding the functionality and interoperability of all DICE tools and needs and requirements set by the demonstrators. In addition, a number of changes and deviations took place regarding DICE tools and their main roles in the updated DICE architecture, which are being presented in detail in the current iteration.

Regarding the integration, we have introduced a 2 type approach in order to facilitate the integration process and allow both the tool owners and the DICE IDE integrator to independently proceed with their tasks and activities without great delays. Type 1 (Inter-Tool integration) includes a number of integration activities that do not concern the inclusion of components in the IDE but the integration between different tools. In parallel with all the efforts related to the IDE, significant efforts have been made for integrating and creating stable interdependencies among tools in order to facilitate the transfer of data, for the benefit of the end user. For these tools, integration with the IDE will be the last milestone as integration between specific tools is a necessary and critical step. On the other hand Type 2 (IDE Based Integration) is dedicated to the integration of tools in the DICE IDE.

1.2. Structure of the Deliverable

The current document is structured into 6 interdependent Sections (Section 1 to 4) and 3 Appendixes (Appendix A to C).

- **Section 1:** Introduction
- **Section 2:** Requirements Update & Overview
- **Section 3:** Architecture Revision & Update
- **Section 4:** Integration Progress, Plan & Next Steps

Adding to the above, the appendices also provide important information which is complementary to all sections.

- **Appendix A:** Updated Tools Summary
- **Appendix B:** Tools Interfaces
- **Appendix C:** Interaction Matrix

Moreover, the current deliverable is accompanied by two companion documents which are not complementary to report. The first companion document is **DICE_D1.4-Companion Document 1** which includes an extensive analysis of for each tool and the second companion document is a complete list of all **Requirements & Usage Scenarios**.

2. Requirements Update and Overview

2.1. Requirements Revision

This section is dedicated on presenting an overview of all changes related to all DICE requirements defined by each Work Package leaders and contributor. Requirements are related to business and technical requirements as well as DevOps practices and technologies used. What needs to be mentioned is that a complete list of all Requirements & Usage Scenarios will be sent as a companion document to this report.

a) WP 1: DICE Integrated Framework

As part of the work in WP1, The methodology has been reviewed as well as the DICE framework tools. Regarding the tools based on Eclipse, the intention was to keep them always updated with the last available version of Eclipse. Currently, Eclipse Neon first revision 4.6.1, released on September 2016. Regarding the work with the Tools, the intention was to keep all the integration plug-ins updated within the IDE. In this way, in each pre-release build of the IDE, always the last versions will be integrated. There are limited changes regarding the Integration plan. There appeared a new Tool called DICER, and a new unforeseen Plug-in for the methodology. Both of them will be available on the next release by M24. In order to integrate the framework tools, the DICE IDE use Eclipse Cheat Sheets, helping users to start working with it and with the included Tools.

The consortium has also decided to change the priority list of the technologies to support in the integrated framework. It has been noted that MongoDB is required in some of the case study activities and therefore has been prioritized over Amazon S3, which is not really used throughout the project activities.

b) WP 2: Methodology and Data-Aware Models

As part of the work in WP2 and the transformation responsibility issues we reported at the end of Y1, the revision of the requirements focused mainly in shaping and restructuring the role of transformations as a means to address deployment design and rollout tooling which was not previously envisioned at the phase of requirements engineering - the transformations in question, we observed, are the critical and most required for the scope of automating the DICE DevOps toolchain. On one hand, we worked to design DICE tools and abstractions so as to render the need for model transformation minimal beyond the DIA modelling which is normally required anyway of the designer. For example, we worked to design our remaining abstractions so as to reduce the effort required to transform manually a DPIM into a DTSM and so forth. Similarly, however, in providing effective facilities for Infrastructure-as-code (IasC) within the DICE methodological and integrated development environment (IDE) the WP2 team worked to produce a stand-alone product and plug-in for the Eclipse-based DICE IDE which was not previously foreseen. The need for this stand-alone product is dictated by the relative absence of technologies that are able to provide reliable support for IasC design and production based on the updated TOSCA standard. The DICER tool in question, was designed in strict collaboration with WP2 partners in WP5, given the established and by-design continuity between the IasC solution DICER and the WP5 Delivery Tool already part of the DICE solution. For this reason, several requirements jointly ascribed to both WP2 and WP5 were conveyed within DICER. Requirements (a) “R5.4: Translation of TOSCA models” - DICER is able to consume stand-alone deployment descriptions and rollout TOSCA models which can be deployed on-the-fly; (b) “PR2.18: DICE Deployment Transformation” - DICER encompasses a series of complex model parsing and navigation transformations that abstract deployment models and distil the necessary fully deployable TOSCA components, using the TOSCA Library jointly worked-out as part of WP2 and WP5 work.

c) WP 3: Data-Aware Quality Analysis

As part of the work in WP3, current requirements have been reviewed while additional requirements have been identified in order to better address WP3 needs. Specifically, Requirements (a) R3.5 “Simulation of hosted big data services” and (b) R3.12 “Modelling abstraction level” have been deprecated. The rationale for deprecating R3.5 is that the simulation tool does not completely simulate the infrastructure where the DIA is

hosted but simulate the behaviour of the DIA at a software level. At the moment, whenever the simulation tool is used, the description of the execution characteristics of hosted big data services is not available. The rationale for depreciating R3.12 is that this characteristic is intrinsic to the developing process used in DICE. Therefore, this requirement does not require any additional effort in WP3, since the DIA development framework covers it by default.

On the other hand, an additional requirement related to the IDE (a) R3.IDE7 "Output results of simulation in user-friendly format" has been created. This new requirement refers to the DICE Simulation tool. The rationale for creating this new requirement is that, on the accomplishment of R3.14 "Ranged or extended what if analysis" of the Simulation tool, both tool developers and users have realized that the multiple results provided by the simulation tool for the what-if analysis, are not easy to see at a single glance. The user needed to open many models (concretely, a model for each different scenario simulated) and see the simulation results of each scenario. This process of opening different result models reduces the visibility of the big picture that a what-if analysis can provide. Therefore, the new R3.IDE7 specifies that "the IDE COULD allow the user to see the multiple output results of the SIMULATION_TOOLS in user-friendly format". A solution to accomplish this requirement passes through an option in the IDE that offers the user to automatically plot results of several simulated scenarios within a what-if analysis in a single graph, thus without requiring opening each model and see each single result.

Requirements R3.15 "Verification of temporal safety/privacy properties" and R3.IDE.4.2 "Loading of the property to be verified" are modified to clarify their content. The two updated requirements specify the role of the UML diagram in the definition of the property to consider for verification. In particular, the word "properties" that were mentioned in the previous version is now changed into "class of properties". Distinguishing the property from the class of a property that the user wants to verify reflects the needs of the designer of the application who follows the DICE model-driven approach and the capability of the verification tool, that operates at the DTSM level of the DICE design, i.e., where the technological details of the application allows the synthesis of a formal model suitable for verification. The designer, in fact, first specifies at the DPIM level the class of the property to be considered for a certain component and, later, chooses the actual instance in the selected class that will be used for verifying the model and to configure the verification task. The instance is chosen in the IDE before running the verification tool. The selection is specified by means of templates shown in the IDE or established by selecting it from a set of predetermined analysis available for the implementing technology.

d) WP 4: Iterative Quality Enhancement

The requirement R4.1 has been extended to cover additional technologies, Apache Spark, Apache Storm, Apache Cassandra and MongoDB. Also, the platform collects post-mortem data from YARN server, i.e. data made available after the process is finished.

The requirement R4.26.1 "Report generation of analysis results" has been deprecated. The rationale for deprecating this is that the DICE-FG is different from the MODAClouds-FG. DICE-FG is no longer an independent tool. It is one of modules of Enhancement tool. The DICE-FG will not generate any analysis report, instead the UML model will be parameterized according to the analysis results and it can be inspected via the DICE Eclipse IDE GUI. Therefore, there is no report at this moment, and the requirement can be removed.

Requirement R4.38, "Monitoring Tool integration in DICE IDE", has been added. The rationale for this feature is that the DEVELOPER/ARCHITECT must have a unique point of access to all DICE tools/services.

Requirements R4.28 "Verification of temporal safety/privacy properties" has been changed to consider the distinction between class of properties and instance of properties, as we already remarked in the previous section with the comment for R3.15. The designer specifies the class of properties that he/she wants to verify for the model in the DICE UML models. The specific property that is actually fed to the solver is selected

through the IDE by means of templates that are instantiated with specific information that the user can provides at the moment of the trace checking task configuration.

e) WP 5: Deployment and Quality Testing

The majority of the WP5 requirements were already well defined in Y1 and are still holding well in Y2. What changed since is related to the introduction of the DICER tool in WP2. We made clearer that the R5.4.2 Translation tools anatomy requirement can be fulfilled by separating application-related properties in the DDSM from the platform-related configurations that are stored in the Deployment service.

The R5.6 Test data generation requirement got elevated to the must have priority, considering this is the tool's primary functionality. At the same time, we deprecated the R5.8.1 representative test configuration generation, because this functionality would duplicate that of the Configuration Optimization, granting the Quality Testing tool's a more focused and easier to use function.

f) WP 6: Validation and Demonstrators

All three demonstrators have updated their requirements as below:

- ***NETFECTIVE Demonstrator***

NETF's initial requirements list is still relevant and most of this list is already implemented by the tools. However, an additional requirement has emerged while testing the released DICE products: The IDE must allow the user to use all the tools using a reasonable number of UML diagrams to allow to capture the functional and non-functional aspects of our case study. This requirement is foundational in order to enhance the user-friendliness of the tools and facilitate the adoption of the IDE within our organization.

- ***PRODEVELOP Demonstrator***

Prodevelop's initial requirements related to scalability analysis, predictive analysis, quality and performance metrics are still relevant to the demonstrator. The bottleneck requirement has been removed because is part of the previous requirements. Model continuous integration jobs is also removed from Prodevelop demonstrator, since it is out of the scope of the project

The Verification Tool is out of our proposed DICE methodology for the POSIDONIA Operations use case.

- ***ATC SA Demonstrator***

ATC's requirements, as being defined in previous versions, are still relevant to DICE tools and are being successfully addressed. However, we have defined an additional requirement (ATC 13) related to the Data Availability. This requirement has been issued by our Technical Team and requests that any developed/designed system should be designed by eliminating any single point of failure and by ensuring at the same time that the system is resilient in a network partition case.

2.2. Technologies & DevOps practices update

The DICE ecosystem has grown considerably in terms of resources and available tooling but technologies and DevOps approaches exploited as part of the project have undergone little variations. As stated in several areas of the present document, the major update to the DICE architecture and technological outfit is concerned with the DICER tool, an Eclipse-EMF based stand-alone solution to automate deployment design and Infrastructure-as-Code production. More information on the requirements and architectural contribution of this additional DICE component can be found in Sec. 1.1.WP2 "Requirements Revision" as well as Sec. 2.2 "Final Architecture". A complete specification of the DICER addition, however, is beyond the scope of this document and will be available on deliverable D2.4, to be released at M27.

3. Architecture Revision & Update

3.1. Overview of Architectural analysis

This section presents the final plug-in architecture for the DICE framework, that takes into account all previous brain-storming elements and architectures, presented in D1.3, and the functionality and interoperability of all DICE tools.

a) Final Architecture

Stemming from our initial assessment, it would seem that the choice of a plug-in architecture rotating around the Eclipse IDE and RCP framework were effective with respect to DICE intents and purposes. Along the results outlined draw from the current architectural status as reported in our latest DICE plenary meeting at M22, DICE partners were able to jointly contribute to the creation of a solid bulk of tools to support the project goals and objectives even sooner than expected (M24). As a result of these activities, the resulting final architecture specification is illustrated in the figure below:

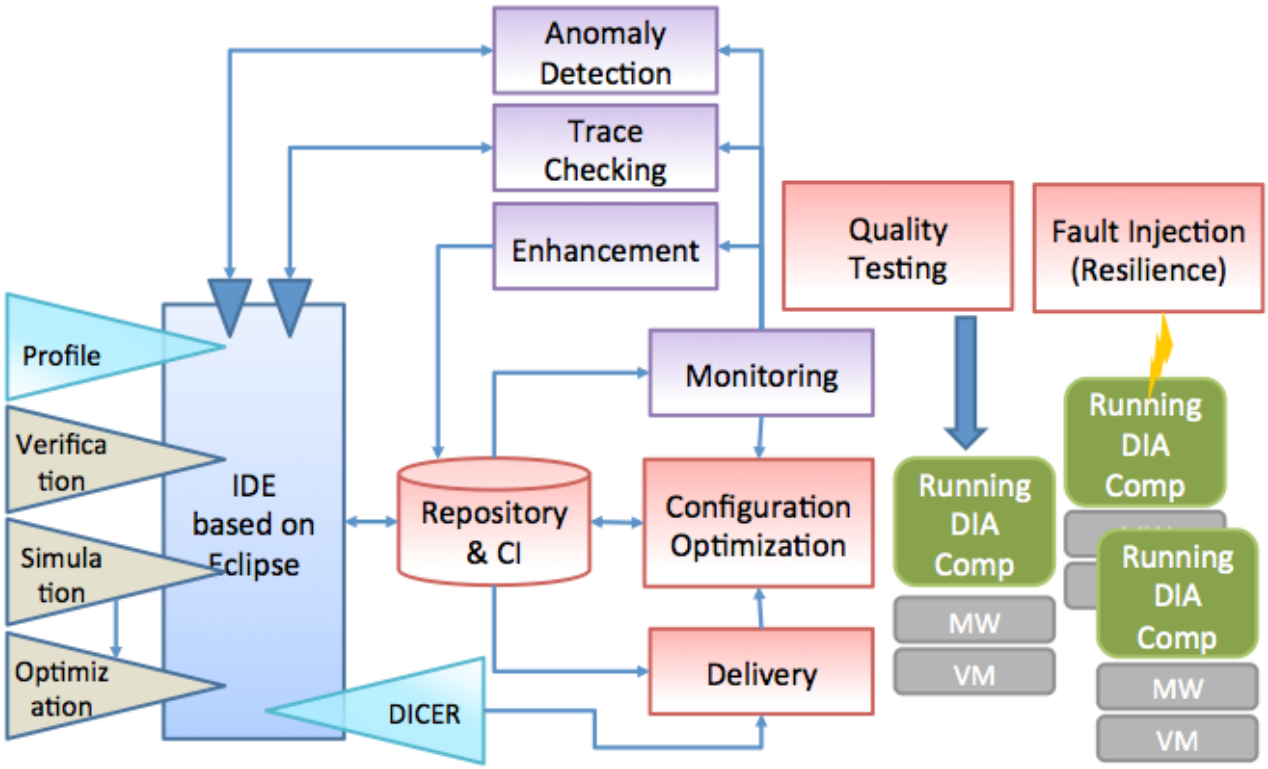

Figure 1: DICE Final Architecture

Architecture Update: with respect to the previously foreseen architectural layout, this architecture reflects and upholds our design decision of a plug-in architecture style, since every functional unit is architecturally de-coupled from other concurrent units and a methodological workflow (i.e., the DICE methodology) dictates the way in which the functional units are meant to cooperate.

One major restructuring and functional addition to the overall DICE architecture is the refactoring of the IDE to accommodate the addition of the DICER component (see mid-lower part of the figure above). From a functional point of view, DICER is set in between the profiles (i.e., from within the DICE IDE) and the Delivery Tool, whereby DICER offers facilities to bridge the gap between the modelling area of DICE and its underlying operational playground. Finally, from a non-functional point of view, the addition of DICER does not provoke any shortcoming in terms of performance and scalability of the DICE solution since its usage is pre-emptive of deployment and straightforward in application - the user experience we designed for DICER, in fact, reflects the possibility to quickly and effortlessly produce deployment specs in TOSCA *.yaml format, at the click of a button from within the DICE IDE. DICER's only working assumption is that the

user/designer has already prepared a deployment diagram either in its native DDSM format or in our own full-fledged DDSM UML profile format from within the UML Papyrus environment.

Another important conceptual improvement during Year 2 has been the methodological realization that a framework architecture design excessively centered on the IDE could have estranged some stakeholders from the DICE platform, in particular operators who are more at ease in using command-line tools. It would not be entirely natural for operators to configure or test system prototypes from a IDE-based interface. The above architecture reflects this by highlighting that the IDE plug-ins are centered.

3.2. Overview of DICE Tools (Update)

In this section we present all changes and deviations that took place regarding DICE tools and their main roles in the updated DICE architecture. One major addition, regarding the previous deliverable D1.3, is the addition of one new tool named DICER.

a) WP level classification

Table 1: WP Level Classification

Tool	Work Package	Lead Maintainer	Major Contributors	Web Link
IDE	1	PRO		https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#idehttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - ide
DICE Profile	2	ZAR	PMI	https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#profilehttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - profile
Deployment Design (DICER)	2	PMI	XLAB	https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#deployhttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - deploy
Simulation Plugin	3	ZAR		https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#simulationhttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - simulation
Optimization Plugin	3	PMI		https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#optimizationhttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - optimization
Verification Plugin	3	PMI	IEAT	https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#verificationhttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - verification
Monitoring Platform	4	IEAT		https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#monitoringhttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - monitoring
Anomaly Detection	4	IEAT	IMP	https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#anomalyhttps://github.com/dice-

				project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - anomaly
Trace Checking	4	PMI		https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#tracehttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - trace
Enhancement Tool	4	IMP		https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#enhancementhttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - enhancement
Quality Testing	5	IMP		https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#qualityhttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - quality
Configuration Optimization	5	IMP	IEAT	https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#configurationhttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - configuration
Fault Injection	5	FLEXI		https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#faulthttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - fault
Delivery Tool	5	XLAB		https://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository#deliveryhttps://github.com/dice-project/DICE-Knowledge-Repository/wiki/DICE-Knowledge-Repository - delivery

b) Development & Runtime tools

The table below, is an updated version of the table presented in the D1.3. We provide a more detailed description of the intended purpose of each tool of the DICE architecture, in two dimensions: **motivation** and **technical innovation**.

Table 2: Motivation & Innovation of Tools

Tool	Motivation	Innovation
IDE	Eclipse is a de-facto standard for the creation of software engineering models based on the MDE approach. DICE further intends to use the IDE to integrate the execution of the different DICE tools, in order to minimize learning curves and simplify adoption.	There is no integrated environment for MDE where a designer can create models to describe data-intensive applications and their underpinning technology stack.
DICE Profile	Existing UML models do not offer mechanisms to describe data characteristics, data-intensive applications, and their technology stack.	The DICE profile extends UML to handle the definition of data-intensive applications. There is no comparable MDE solution in this space, therefore the innovation is to be a first mover.
Deployment Design and Rollout (DICER)	As a direct counterpart of the Delivery Tool, the Deployment Design tool offers facilities to procure and uphold deployment abstractions in the form of a modelling environment that allows to quickly put together a deployment model and obtain its infrastructure-as-code counterpart in a matter of seconds.	DICE offers the first-of-its-kind TOSCA-based solution for infrastructure-as-code of Data-Intensive Applications. No solution to date offers this kind of support in a cascade with UML-based quality-aware modelling. DICE's unique infrastructure-as-code solution works in full continuity with DevOps strategies and is flexible enough to accommodate and encompass

		quickly addable technological extensions to be designed in a model-driven fashion.
Simulation	Once a data-intensive application is designed, simulation can help anticipating the performance and reliability of the software before implementation or throughout revision cycles. Example of questions that can be answered by a simulation tool include: how many resources (VMs, memory, CPU, ...) will be required to achieve a given performance target? What will be the response time and throughput of data-intensive jobs?	There exist tools and environments to translate an application design specification into simulation models, however none copes with the notion of data or can generate models for data-intensive technologies. Instead, the DICE simulation tool will be able to generate and simulate models for specific data-intensive technologies (e.g., Hadoop/MapReduce, Spark, Storm, etc.)
Optimization	Simulation offers the possibility to evaluate a given model. However, thousands of models may need to be evaluated in order to maximize some utility function, e.g. finding an architecture that incurs minimum operational costs subject to data redundancy and reliability requirements. The optimization plugin will perform multiple invocations of the JMT or GreatSPN simulators to support the automated search of optimal solution. This is needed to limit the time and number of alternative deployments inspected by the software architect to obtain a good solution. The optimization tool relies on the simulation plugins for M2M transformations.	Design space exploration has been increasingly sought in traditional multi-tier applications, but not in the design of data-intensive applications. For example, it is not possible today to find optimal architectures subject to constraints on dataset volumes and transfer rates. Delivering this capability will constitute the main innovation of the optimization tool.
Verification	Simulation is helpful to study the behaviour of a system under a variety of scenarios. However, it cannot provide definite answers concerning the impossibility of some events. For example, in safety-critical systems, a designer may want to avoid that certain schedules of operation result in loss of data (e.g., due to timeouts, buffer overflows, etc). Furthermore, simulation requires a substantial effort to answer logical predicates, since it is not meant to be queried through a logic. Verification tools allow to address these problems, offering a logic language to analyze the correctness of a system, generate counterexamples, and expose safety risks.	Verification tools often have a high learning curve for non-experts. The DICE verification tools will be integrated with the IDE ecosystem to simplify the invocation of verification analyses in a user-friendly way. This will be achieved by the use of templates to run specific analyses on specific technologies.
Monitoring Platform	During prototyping and testing it is important to collect operational data on the application and the infrastructure to understand if all the design constraints are satisfied. There is, however, a gap between high-level design metrics (e.g., data throughputs) and the concrete low-level mapping of these metrics to quantities in log files. The monitoring platform takes care of this mapping from data-intensive technologies, of the retrieval of the data and its storage and querying through a data warehouse.	There exist several monitoring open source tools in the public domain. However, the integration of these tools into a solution to support developers (easy deployment, extensible to various Big Data technologies) is atypical use, as these are mostly used by operators. Another innovation is the contribution to simplifying the monitoring process, by offering the default selection of representative metrics across DICE-supported technologies.
Anomaly Detection	As an application evolves it is not always simple to decide if the application performance or reliability have been affected by a change. This requires to perform statistical analysis to compare monitoring data across versions. This tool will perform this analysis based on the different version of the DICE application and models.	Anomaly detection tools exist in the open source domain; however none is specifically tailored to MDE. There is also not yet evidence that such systems can be effective in finding anomalies in data-intensive applications, therefore this prototype will push the boundary in a novel research space.

Trace Checking	Anomaly detection can also be performed by trace checking, which involves ensuring that a sequence of events appearing in a trace is correct with respect to pre-defined characteristics. Compared to anomaly detection, this capability allows users to evaluate logical queries on the trace to check its correctness, as opposed to the idea of the anomaly detection tool of verifying the application behaviour using a statistical analysis.	The DICE trace checking tool will complement the formal verification tool, in that it will help determine, from actual traces of the system execution, whether the parameters with which the formal verification model were initialized are indeed correct; it will also make sure that the properties analyzed at design-time still hold at runtime (they might be violated due to an incorrect configuration of the parameters, as mentioned above). Trace checking plays relevant role in the analysis of privacy requirements as it allows for checking that desired privacy restrictions imposed in the application models are correctly enforced at runtime in the deployed application.
Enhancement Tool	Given monitoring data for an application, a designer needs to interpret this data to find ways to enhance the application design. This is complex to perform, since the components, annotations and abstractions used in a UML model do not semantically match to the concrete low-level metrics that can be collected via a monitoring tool.	The Enhancement Tool will introduce a new methodology and prototype to close the gap between measurements and UML models. No mature methodology appears available in the research literature that can address this inverse problem of going from measurements back to the UML models to help reasoning about the application design.
Quality Testing	The testing of a data-intensive application requires the availability of novel workload injection and actuators (e.g., scripts to automatically change configurations, to instantiate and run the workload generators, etc.)	Most of workload injection and testing tools are specific to multi-tier applications (e.g., JMeter). The DICE quality testing tools will focus primarily on data-intensive applications, for which there is a chronic shortage of such tools.
Configuration Optimization	Once an application approaches the final stages of deployment it becomes increasingly important to tune its performance and reliability. This is a time-consuming task, since the number of different configurations can grow very large. Tools are needed to guide this phase, especially in defining the optimal parameters of the underlying technology platform.	There is a shortage of tools to guide the experimental configuration of complex software systems. This provides an innovative solution in this space, which will combine experimentation with reasoning based on machine learning models. An innovation is also the possibility to tune the level of exploitation and exploration, in order to trade quality of the best configuration for aggressiveness in the change of parameters of a system. This may be useful for example in tuning production systems, which cannot change their configuration too markedly.
Fault Injection	Given an initial prototype of the application, it is important for reliability purposes to understand the resilience of the application to unexpected problems in the operational environments (e.g., faults, contention, etc). The fault injection tool will address this need by offering an application that can create on-demand such problems to explore the application response. This will ensure applications are developed to handle multiple failures.	Some fault injection tools exist on the market, such as ChaosMonkey. These tools however are either platform specific or limited in functionality such as with ChaosMonkey for AWS and termination of VMs. The DICE Fault Injection tool will address the need to generate Faults at the VM level, at the user Cloud Level and on the Cloud platform level. This larger range of functionalities allows a greater flexibility as well as the ability to generate multiple faults from a single tool. In addition, when compared to other fault injection tools, shall be light weight and only install the required tools and components on the target VMs.
Repository	This is an auxiliary system required to store and version the models used by the other tools.	This is an auxiliary system tailored to the integration of the DICE tools.
Delivery Tool	The DevOps paradigm assumes development to be a continuous process, where the application code can be often changed and redeployed through continuous integration	DICE offers a continuous deployment solution that combines some cutting-edge solutions for cloud computing, namely the emerging TOSCA profile and the Cloudify deployment tool. The

	tools to examine the application response. The infrastructure and the whole applications are described in code as well. DICE aims at emphasizing the adoption of this paradigm during the pre-production stages, in order to accelerate the generation of the initial prototypes and proceeding to develop production-grade DIAs.	tool can be used as stand-alone, but works even better as a natural extension of the model-driven development into realization of the model in the target environment. This powers a complete design-deployment-testing ecosystem in DICE. Complete with the TOSCA deployment library, it provides a valuable package for clear and simple to understand blueprints that result in a live application in a matter of minutes.

3.3. Positioning tools in the methodology update

Some of the DICE tools are design-focused while some are runtime-oriented. Moreover, some have both design and runtime aspects. The following table outlines the tools mapping them to this categorization scheme. All of the tools relate to the runtime environment.

Table 3: DICE tools (Final)

	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 modelling 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 a service must meet in order to be useful for the application.

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 behaviour 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 works allows to simulate the behaviour of a data intensive application during the early stages of development, based on the DPIM and DTSM specification. It relies on high-level abstractions that are not yet specific to the technology under consideration.
- Differently from the simulation tool which focuses on a single deployment model, the optimization tool explores in parallel a very large number of configurations and the minimum cost deployment is finally found. The optimization tool focuses on DTSM and DDSM, and relies on the simulation plugin to support the transformation of the DICE UML models to JMT and GreatSPN models.
- 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 models of it.

- The monitoring tool collects runtime metrics from Big Data frameworks, indexes it in the data warehouse and provides infrastructure management and data query REST APIs.
- The quality testing tool and the fault injection tool respectively inject workloads and force failures into the runtime environment; for instance, shutdowns of 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 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 DICE UML model on DTSM level. Moreover, the Enhancement tool also updates the UML model with performance and reliability information.

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

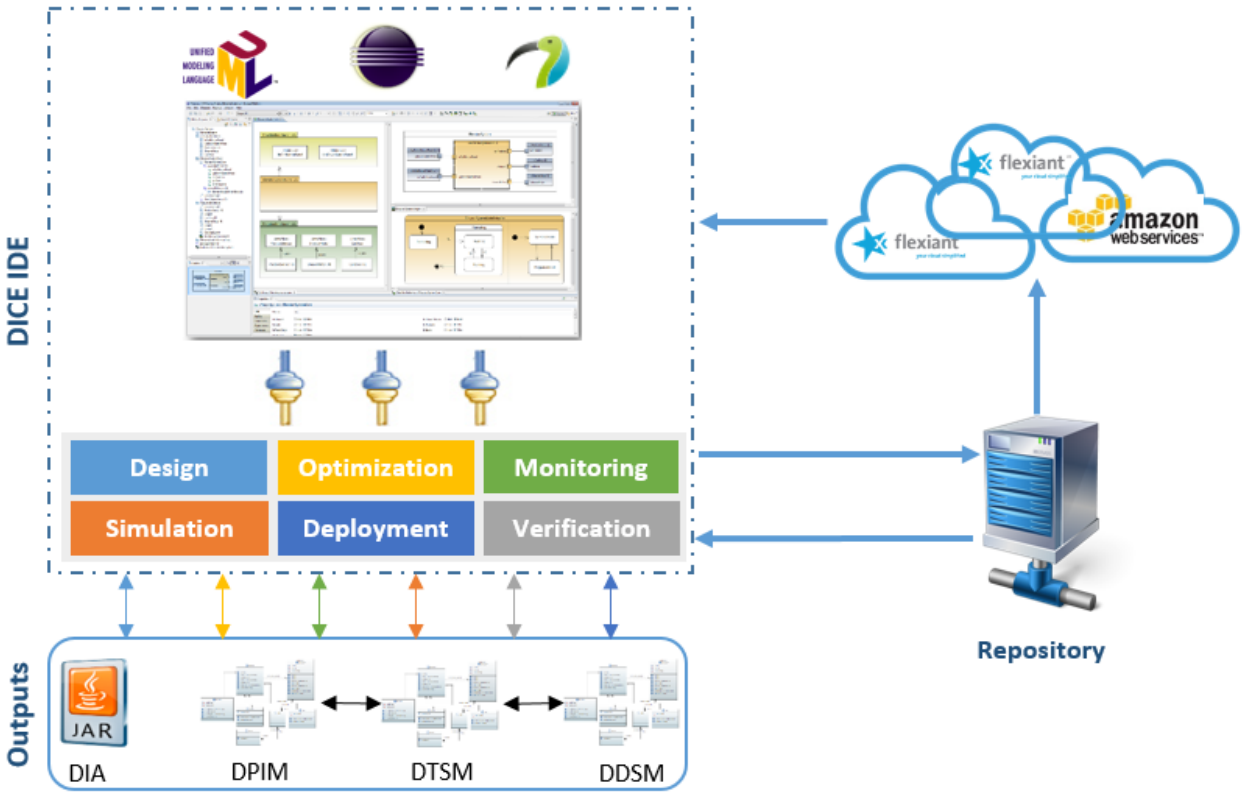

Figure 2: DICE ecosystem

The table 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	DTSM, DDSM
Enhancement	Activity	DTSM
	Deployment	
Optimization	Activity	DTSM, DDSM
	Deployment	
Monitoring*	Not Applicable	Not Applicable
DICER	Class, Deployment	DTSM, DDSM
Delivery tool	Deployment	DDSM
Quality testing	Deployment	DDSM
Configuration optimization	Deployment	DDSM
Anomaly detection*	Not Applicable	Not Applicable
Fault injection	Deployment	DDSM

The **DICE Knowledge Repository**, described in detail in D7.2, 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. In fact, this latter is the central entry-point for DICE users in the context of practice and putting hands in the DICE IDE and related tools. This collaborative wiki was launched in M18 (July 2016) and is progressively enriched and updated with new contents. For each tool, the user has a short description, links to a detailed description, a tutorial (Getting Started), a video (hosted on the project Youtube Channel), a link to the source code (hosted also on Github), the licence and the email of the tool owner.

*In the case of the **Monitoring** and **Anomaly Detection tools** UML is not required as input. These tools are at the end of the toolchain and are required only after the DIA is up and running. As such they receive their information from other tools. For example the Monitoring Platform receives all of its inputs from the Deployment Tool. In essence the information from the DDSM level is filtered to the Monitoring via this link. Similarly, **Anomaly Detection** requires information from both DDSM and DTSM level. The information available in these profile levels is however, not enough to detect anomalies. The DIA needs to already be running. The best solution is to encode the information from DDSM and DTSM via the query used by the Anomaly detection to get the data from the Monitoring.

4. Integration Progress, Plan & Next Steps

The current section is dedicated to present an overview of the integration progress (so far) and plan/foreseen activities for the remaining of the project. In order to facilitate the whole integration process, we have decided to divide the integration approach in two different types: **Type 1: Inter-Tool Integration** and **Type 2: IDE Based Integration**.

Type 1 includes a number of integration activities that do not concern the inclusion of components in the IDE but the integration between different tools. This type of integration is applicable to all tools in the categories defined in Table 3 as Runtime, Design-to-runtime and Runtime-to-design and to some of the tools in the Design time category. For these tools, significant effort has been made for integrating and creating stable interdependencies among tools in order to facilitate the transfer of data, for the benefit of the end user. For these tools, integration with the IDE (Type 2 integration) will be the last milestone as integration between specific tools is a necessary and critical step. Type 2 integration has been, instead, adopted in the first iteration for those tools that belong to the Design time category (**Table 3: DICE tools (Final)**). A detailed description of type 2 integration approach and of the IDE-based integration activities that have been developed so far can be found in **Deliverable 1.5. DICE Framework – Initial version**.

4.1. GitHub – General Description for a Source Code Perceptive

All the public software artefacts developed in the DICE project are available through github platform. GitHub is a web version control repository that includes interesting features for collaborative contribution as: access control, bug tracking, feature requests, task management and wikis.

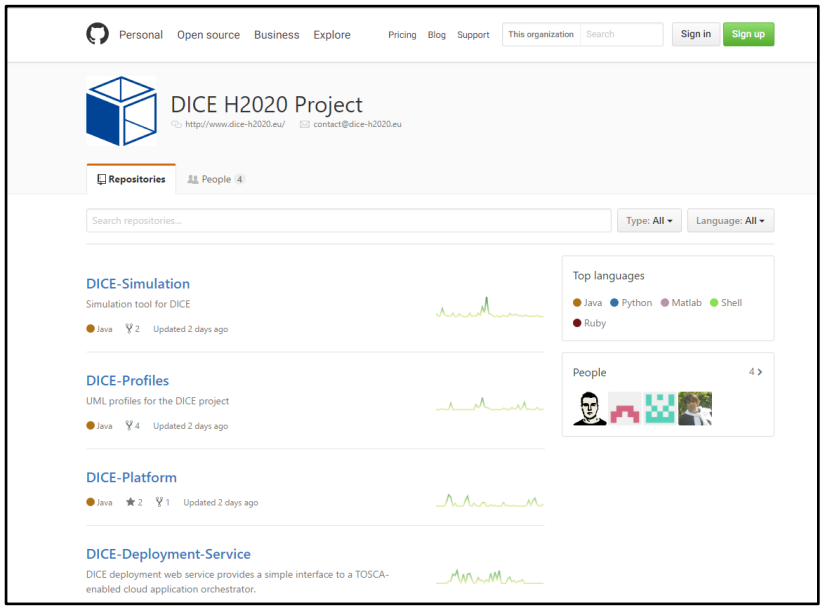

Figure 3: github DICE project

The main page of the Dice project in the github platform is shown at Figure3 and its url is https://github.com/dice-project. Inside the dice project you can found a set of repositories, each tool has its own repository. **Table 1: WP Level Classification** contains the URLs of the repository of each tool.

The repository of a tool contains the source code for the different versions of the tool, description of the repository with setup information, issues detected and future plans, etc.

4.2. Model-Based DICE Interoperability

a) DICE tool to DICE Diagrams mapping

As any model-driven solution, several functional areas of the DICE technological ecosystem rely on models as information interchange and interoperability mechanisms. As such, the DICE ecosystem entails a mapping between DICE tools and the input UML diagrams produced/consumed within the DICE Eclipse IDE. An overview of this mapping is reported here below and will be further elaborated in the final DICE methodology deliverable (D2.5) due at M30:

Table 5: UML Diagrams Mapping across DICE tools

DICE tool	Input UML diagram (with appropriate DICE stereotypes)	Profile level
Simulation	Activity	DPIM, DTSM
	Sequence	
	Deployment	
Verification	Class	DTSM
Trace checking	Class, Deployment	DTSM, DDSM
Enhancement	Activity	DTSM
	Deployment	
Optimization	Activity	DDSM
	Deployment	
Monitoring	Deployment	DDSM (Indirect)
Deployment Modelling	Deployment	DDSM
Delivery tool	Deployment	DDSM
Quality testing	Deployment (Indirect)	DDSM (Indirect)
Configuration optimization	Deployment (Indirect)	DDSM (Indirect)
Anomaly detection	Object	DTSM (Indirect)
Fault injection	Deployment	DDSM

b) Model-Based DICE Tool Interactions

As highlighted in the previous section, there are several functional areas in the DICE ecosystem that interact by means of DICE tools. The key principle and general rule to intuitively map which tool interoperates with which other in a model-driven fashion is fairly simple:

"any tool that works on diagram type X at DICE abstraction level Y, may be able to interoperate with tools that use/refine/act on <X,Y> as well."

For example, a known model-based interoperability statement reflects the Simulation tool to essentially bootstrap the DICE quality-aware analysis by modelling an UML activity diagram reflecting the specific behavior of the data-intensive application to be. By means of that diagram, DICE-based **Simulation** will be

carried out using standard DICE approaches and guidelines defined in the scope of WP2 and WP3. Similarly, the **Optimization** tool can work with the simulated diagram to further improve it for deployment using the concept of a "deployment candidate"; a deployment candidate is any activity diagram that can be mapped with an infrastructure design devised within and by means of the **Deployment Modelling** tool (i.e., a UML DDSM deployment diagram). Once the preferred deployment candidate is devised, the **Deployment Modelling** tool can be used to infer a TOSCA blueprint for further processing. At this point, that very same DDSM is "transposed" into an orchestration-ready TOSCA blueprint which is:

- Consumed and deployed by the **Delivery Tool**;
- Analysed and improved with the **Configuration Optimization** tool;
- Observed by the **Monitoring Tool**;
- Exercised by the **Fault Injection** and **Quality Testing** Tools;

All the above interoperations are foreseen, highlighted and supported from a methodological point of view. With respect to these foreseen model-based DICE interoperability features, here follows a detailed overview of said features in a tool-per-tool fashion.

Simulation

- **Model-input:** UML diagrams: activity diagram plus deployment diagram, or sequence diagram plus deployment diagram. Profiles: DICE DPIM (used for modelling at DPIM level), DICE DTSM (used for modelling at DTSM level), MARTE (used for modeling properties for performance analysis and probabilistic behaviors), DAM (used for modeling properties for reliability analysis).
- **Model-output:** The output of the tool are the expected values of performance and reliability metrics. It does not create any output diagram.

Verification

- **Model-input:** UML class diagram at DTSM level defining a Storm topology or a UML activity diagram representing a directed-acyclic graph (DAG) of a Spark job
- **Model-output:** the output is a boolean value; no diagram are produced.

Trace checking

- **Model-input:** UML class diagram at DTSM level defining computational nodes of the application and their implementing technology and deployment diagram at DDSM to represent deployment details.
- **Model-output:** the output is a boolean value; no diagram are produced.

Enhancement

- **Model-input:** Enhancement tool is primarily concerned with Big-Data technology requirements modelled at the DTSM layer. The corresponding UML diagrams are deployment diagram and activity diagram. Certain functionalities of the tool may in principle apply also to DPIM models, but the focus is on DTSM models.
- **Model-output:** Enhancement tool has two submodules, FG and APRI. The output of the APR tool is the refactoring recommendations detected in the current UML models, these require manual change by the user and at least in the first version of the tool will therefore not automatically generate a new UML model. Conversely the FG tool annotates the existing DTSM model with the required information to simulate the relevant technology.

Optimization

- **Model-input:** The optimization tool requires as input a DICE DTSM model including an activity diagram (describing task dependencies) and a deployment model for the application under study and a DICE DDSM model including a deployment model.

- **Model-output:** The optimization tool identifies the deployment configuration of minimum costs and updates the input DDSM model accordingly. The DDSM model will be then translated by the DICER tool into a TOSCA blueprint.

Monitoring

- **Model-input:** The monitoring tool requires input from the delivery tool which in turn gets a TOSCA blueprint generated using DDSM. Because the monitoring is at the end of the toolchain it doesn't require direct access to DDSM, instead it receives the data derived from DDSM from the Delivery Tool
- **Model-output:** The monitoring tool's outputs the performance metrics for the Big Data platform underlying the DIAs.

Deployment Modelling the deployment modelling tool, a.k.a., DICER allows using the DDSM UML Papyrus to produce TOSCA blueprints which are consumed by the delivery tool - the DICE UML profile is the source of such diagrams and DICER will understand models produced by means of that profile regardless of the technological specifications therein. DICER focuses on understanding what infrastructure is needed stemming from the deployment details specified as part of a DICE deployment diagram.

Delivery tool

- **Model-input:** TOSCA blueprint, which is a transformation of the DDSM. The DICE profile is reflected in the node types appearing in the resulting TOSCA blueprint along with any configuration entries. The delivery tool interprets the model to enable orchestration of the DIA to be deployed.
- **Model-output:** the tool does not produce any new models. The main output of the tool is an instantiation of the input DDSM's TOSCA representation. In other words, the tool's output is a DIA's runtime in the testbed.

Quality testing

- **Model-input:** The quality testing tool provides a Java API for specifying and injecting custom application workloads. As such there is no component in this DICE tool that deals with UML models.
- **Model-output:** The quality testing tool results are monitoring data which is stored in the DICE Monitoring Platform and can subsequently be analyzed by tools such as DICE Enhancement Tool for automated annotation of the DICE UML models, primarily at DTSM layer.

Configuration optimization

- **Model-input:** The configuration optimization tool does not require a modelling abstraction of the application to work. It can therefore operate agnostic of the UML diagrams. Since the underpinning engine is based on blackbox optimization, it is against the technique to assume white-box information such as the one provided in UML diagrams. Indirectly, since the tool requires successive re-deployments of the application, it is leveraging the DDSM and TOSCA models via the Deployment service.
- **Model-output:** The tool produces in output a recommended configuration, which can be used to annotate DTSM models.

Anomaly detection

- **Model-input:** The anomaly detection tool is tightly coupled with the monitoring platform. Because of this it receives all relevant data from the monitoring platform. This data ranges from performance metrics for each monitored nodes coupled with some DIA specific metrics to what role each node is assigned.
- **Model-output:** Because of the tight coupling of the monitoring platform and the anomaly detection tool all detected anomalies are sent and stored inside the monitoring platform as a separate index in the elasticsearch core service.

Fault injection

- **Model-input:** The Fault Injection Tools (FIT) will look to receive a *list of a living nodes and their IPs and types/roles from the DDS. This input will be provided to the FIT to automatically start faults onto the*
- **Model-output:** Faults injected into each VM will be detected by the monitoring solution. In addition a generated log output from the FIT will be available for users to timestamp the created faults.

Beyond the above-defined statements, any unforeseen interoperation by means of DICE-models which is not foreseen here is not sanctioned by the DICE consortium nor supported by methodological means. This notwithstanding, the DICE pipeline can be run in any desired way, shape or form which is consistent with stakeholder concerns and desires but the outcome results may deviate considerably with the expectations and assumptions we have been using to design our research solution (see D1.2 and D1.3) - that being said, we commit to supporting any desired alternative applications of the intended DICE pipeline as part of our pay-per-use and/or consultancy/based value offer beyond M36.

4.3. Integration Status & Next Steps

a) Type 1: Inter-Tool Integration

- ***Deployment Design***

<table>
<tr><td colspan="2">Tool Name: Deployment Modelling (DICER)
Owner: PMI</td></tr>
<tr><td>Type 1: Inter-Tool Integration</td><td>
Current Status of Integration with other tools: From an integration point of view DICER is both fully integrated within the DICE IDE and existing as a stand-alone tool. DICER supports the DICE-Profiles by being able to consume DICE UML DDSM models, i.e. UML Papyrus models tagged with stereotypes from the DICE DDSM profile. DICER is structured according to a Façade Design Pattern meaning that:
1. it provides an easily accessible Graphical User Interface (GUI) directly embedded within the Eclipse DICE IDE RCP;
2. it provides a larger body of code in the form of a JAR executable server that is entrusted with consuming papyrus models or DICER DSL instances to generate deployable TOSCA blueprints.
Integration with other tools:
● Delivery tool: Although it does not directly use the DICE TOSCA Types and Templates from the DICE technology library worked out and embedded within the DICE Deployment service, the tool (i.e., the JAR executable file with the core back-end server of the DICER tool) hardcodes the modelling image (i.e., the set of infrastructure-as-code notations that allow for automated deployment of TOSCA blueprints) of the library inside its transformations and is therefore able to interface correctly with the DICE Deployment service by direct method invocation and REST, passing its TOSCA blueprint output as an argument of the invocation.
● Optimization tool: in line with interaction dynamics outlined as part of the DICE delivery tool, the DICER tool provides a command line interface that can take as input a DDSM model part of a configuration optimisation DICE model instance and feed that instance to the DICE deployment service itself by means of the DICER back-end. In essence, as soon as the DICE UML DDSM diagram has been prepared directly within the DICE IDE, the DICER tool back-end can be invoked as a simple invocation using an Eclipse DICE IDE run configuration from the run-configuration menu or clicking the run-configurations menu-button. Finally, the DICER JAR executable file is invoked and will be running at the time the Eclipse DICE IDE is used to invoke a DICER transformation.
● Configuration Optimization tool: DICER produces the TOSCA Blueprint from DDSM diagrams that are already enriched with properties and parameters needed to run instances of the configuration optimization tool based on the BO4CO baseline - DICER itself, however, does not foresee any direct interaction dynamics between itself, its back-end and the configuration optimization tool. Conversely, DICER produces the blueprint in such a way so as to allow the delivery tool to instrument the configuration optimization tool acting as an intermediary.
No further integration is planned for the DICER tool beyond what outlined here.
</td></tr>
<tr><td></td><td>Interactions (i.e. RestAPI etc.):</td></tr>
</table>

- **DICE IDE Papyrus Profiles -> DICER Tool:** Although as a standalone tool DICER can consume models conforming to its Domain Specific Modeling Language, from an interaction point of view, DICER is a pipeline element that, within the DICE IDE assumes previous usage of the DICE UML profile and outputs data towards the DICE deployment service. In this case DICER expects the DICE UML DDSM profile to be properly used (i.e. according to the proposed DICE methodology for deployment modeling) to structure a UML DICE Papyrus Diagram containing a UML Deployment Diagram (at least)**.** In this diagram the DICE user needs to specify the concrete infrastructure design from a structural and configuration point of view. The DICER frontend IDE plugin is able to transform such UML model into a DICER specific model using an embedded ATL model-to-model transformation.

- **DICER Tool front-end -> DICER Tool back-end**: Once the above details are available a DICER-specific launch configuration can be invoked from the DICE IDE run-configuration panel. The run-configuration in question sends a DICER model to the DICER back-end using a REST endpoint.
  The DICER backend, which is a stateless service, parses the diagram upon which the launch is made and returns a viable TOSCA Blueprint ready for deployment.
  REST endpoint used for sending the DDSM model to the DICER backed:
  - POST /v1/**generateToscaBlueprint**
  - input: an .xmi file containing the DICER model or a .uml file containing a DICE UML DDSM model
  - output: a .yml file containing the generated TOSCA blueprint
- purpose: obtain a TOSCA deployable blueprint from a DICER or a DICE UML DDSM model

  DICER Tool front-end -> Delivery tool: At this point, a context action in the DICE Eclipse IDE right-click menu exercised on the TOSCA *.Yaml file produced by DICER can be used for its deployment. That context action uses a REST API invocation (POST /containers/**CONTAINER_ID**/blueprint) to the front-end service component of the DICE Delivery tool; this component is entrusted with type-checking and sending that deployable blueprint to the DICE Deployment service cloud orchestrator engine, i.e., the DICE-specific Cloudify implementation we devised as part of the deployment service itself.

- **DICE Delivery Tool - DICER**: DICE Delivery Tool can accept a DICER model as input and uses the DICER's CLI internally to transform the DDSM model into a TOSCA document:

  $ java -jar "**$DICER_TOOL_PATH**/dicer-core-0.1.0.jar" \
  -inModel "**IN_XMI_FILE/IN_UML_FILE**" \
  -outModel "**OUT_TOSCA_FILE**"

Functional Description: From a functional point of view, DICER is a bridge between the DICE IDE and the DICE Deployment Service. DICER produces TOSCA blueprints that average in 150 lines of infrastructure-as-code. This data (in the range of 5-10 KB) is sent according to the above procedures to the DICE Deployment Service. The DICER (client) and the Deployment Service (server) are therefore an actuator and orchestrator of infrastructure-as-code and automated deployment in the scope of the DICE IDE. In this vein, UML is merely a means to convey infrastructure-as-models using the DICE familiar notation of DDSMs. DICER, in turn, allows using the DDSM UML Papyrus profile and will understand models produced by means of that profile regardless of the technological specifications therein. DICER focuses on understanding what infrastructure is needed stemming from the deployment details specified as part of a DICE deployment diagram. Conversely, however, DICER also allows abstracting away all possible technological details (if needed) using a simple notation for Domain Specific Modelling of infrastructure-as-code. This feature, however, is available in a separate and stand-alone version of the DICER tool.

DICER backend service provides:

- Transforming a .xmi file into a deployable TOSCA bleuprint:
  - RESTful:
    - POST /v1/**generateToscaBlueprint**
    - input: a .xmi file containing a DICER specific model
    - output: a .yml file containing the generated TOSCA blueprint

	▪ purpose: obtain a TOSCA deployable blueprint from a DICER or a DICE UML DDSM model **DICER backend core provides:** ● Transforming a .xmi into a deployable TOSCA blueprint using the DICER backend .jar executable file with command line parameters. **DICER frontend (i.e. DICE IDE Eclipse plugin) provides:** ● Creating DICER specific models (in .xmi format) and validating them using OCL constraints defined over the DICER metamodel. ● Transforming DICE UML DDSM models into DICER specific models using a model-to-model transformation. ● Sending a .xmi DICER model to the DICER backend via an Eclipse run configuration invoking a DICER backend's REST endpoint. ● Setting up a default DICER project with default DICER specific models for various supported technologies. ● Checking if the DICER backend server is alive.
	Integration Testing Scenario: ● **Testing Objective "DICER can be used to produce a TOSCA blueprint for my DDSM":** DICER can be tested freely using the DICER-specific run-configuration. Users need to: o Prepare a UML DDSM Diagram (this is actually the longest part of any testing scenario relating to DICER); o click on the "Run" Eclipse DICE IDE RCP menu entry; o navigate to the "Run-Configurations" menu entry; o click on the "DICER Run" entry; o specify input and output model; o [Optionally] DICER users can test if the JAR executable file acting as a serving layer for the DICER invocations is still alive – this same call will restart the service as needed; o A Deployable TOSCA blueprint shall be produced in no more than 5 minutes; ● **Testing Objective "DICER allows me to deploy the optimised infrastructure design which I just edited":** DICER is able to allow comparative deployment of multiple DDSM models prepared and optimised in the context of the optimization tool. Users need to: o prepare a UML DTSM+DDSM Diagram containing a DDSM deployment candidate in the context of the optimization tool; o optimize the DTSM+DDSM deployment models; o select the appropriate DDSM deployment candidate with best desired results; o click on the "Run" Eclipse DICE IDE RCP menu entry; o navigate to the "Run-Configurations" menu entry; o click on the "DICER Run" entry; o specify input and output model; o [Optionally] DICER users can test if the JAR executable file acting as a serving layer for the DICER invocations is still alive – this same call will restart the service as needed; o A Deployable TOSCA blueprint shall be produced in no more than 5 minutes; ● **Testing Objective "DICER actually produces Deployable TOSCA blueprints":** o prepare a UML DDSM Diagram; o click on the "Run" Eclipse DICE IDE RCP menu entry; o navigate to the "Run-Configurations" menu entry; o click on the "DICER Run" entry; o specify input and output model; o [Optionally] DICER users can test if the JAR executable file acting as a serving layer for the DICER invocations is still alive – this same call will restart the service as needed; o A Deployable TOSCA blueprint shall be produced in no more than 5 minutes; o Given the DICE Deployment Service is available at its address; o And I have a DDSM in the file 'storm.xmi'; o When I submit the DDSM 'storm.xmi' to DICE Deployment Service; o Then the deployment should succeed after no more than 90 minutes;

	Next Steps: DICER will be published as soon as possible and is due to being elaborated in the final version of the Deployment Abstractions deliverable, i.e., D2.4 upcoming at M27. Immediately after that and within M30, we will continue DICER maintenance and evolution, hopefully evaluating DICER in even larger DIA scenario to conclude DICER evaluation. Between M30 and M36 we will reflect and evaluate the extensibility around DICER, trying to reduce as much as possible DICER learning curve and extension costs. More concretely: ● M25: DICER conference paper submission; ● M27: DICER architecture elaboration as part of D2.4; ● M26-27: DICER architecture integration with Delivery tool; ● M30: DICER methodological overview evaluation and consolidation; ● M36: DICER extension evaluation and experimentation; ● M36+: Continued DICER support and keep-alive;

- ***Simulation Plugin***

Tool Name: Simulation Tool **Owner: ZAR**	
Type 1: Inter-Tool Integration	**Current Status of Integration with other tools:** The Simulation tool does not require any other DICE tool to achieve its functionality. Nevertheless, beyond the DICE IDE, there are expected interactions with two tools for the well lubricated execution of the flow of DICE framework, one of them developed within DICE and the other external. ● Regarding internal interaction, there is a *Filling the Gap (DICE FG)* tool that belongs the DICE Enhancement module which produces annotated UML models that will be later evaluated with the Simulation tool. This flow of executions happens when the Simulation tool is used for systems whose characteristics are already monitored by DICE monitor. On the one hand, to ease the human interaction with the Simulation tool, the tool allows the user to first define variables, then create the model characteristics using these variables, and finally assign values to the variables using a specific GUI before launching the actual simulation. This is useful for the user because s/he does not need to traverse the model when s/he just want to change a couple of values, but all the possible variables appear together in the same configuration screen. On the other hand, this capability of the Simulation tool creates some issues when *DICE FG* executes. The variables are defined, but not used, because *DICE FG* writes actual values in the stereotypes where variables were first used by the user. To keep a fluid execution of the different tools of DICE framework, action taken by the Simulation tool is to relax its necessities about utilization of variables. Now, if a variable is defined –because the user defined it in the first design- but later it is not used in the model –because the *DICE FG* has overwritten its utilization with actual values computed from the monitored information, the Simulation tool continues its execution and behaves as if the variable were not defined and did not exist. By including this characteristic, the simulation tool accepts a broader set of model definitions. ● Regarding external interaction, the Simulation tool has currently implemented a call to a Petri net simulation engine; i.e,. GreatSPN. Although the Simulation tool is open to include other types of engines, as JMT or even a new one internally implemented, at present, the type that invokes GreatSPN simulation is the only one that has been implemented, tested and released. Therefore, at present, the Simulation tool needs of an SSH accessible machine with GreatSPN installed. This machine can be the localhost, a virtual machine running in the localhost, or any other server in a reachable network. The status of the integration with machines with GreatSPN is: completed. A module of the Simulation tool connects through SSH to the machine, copies the generated Petri nets into the sever, executes the GreatSPN engine using the copied Petri nets as inputs, retrieves the GreatSPN results again to the Simulation tool in the DICE IDE.
	Interactions: ● The interaction with *DICE FG* tool is through UML models, what it is produced by one of the tools has to be consumed later by the other, but none of them calls each other. Therefore, there is not any kind of "call" from one to the other ● The interaction with GreatSPN simulation engine is done through an SSH connection, and then invoking simple Linux command-line commands, like create a directory for storing the Petri nets and execute through the command-line an already executable file.
	Functional Description: ● The Simulation tool and *DICE FG* do not directly interchange any information; none of them invokes the other. However, the Simulation tool may require to read UML models

	whose stereotype have been modified by the *DICE FG* tool from monitored data.
	● The Simulation tool and GreatSPN interchange the following:
	  ● The simulation tool invokes GreatSPN with the Petri net files generated during the model-to-model transformations internal to the simulation tool.
	  ● The Simulation tool receives from GreatSPN simulation engine its standard output. This standard output contains performance results in the domain of stochastic Petri net; namely average number of tokens in each Place of the Petri net, and average throughput of each of its Transitions.

	Integration Testing Scenario: For the interaction with *DICE FG:* ● The user creates a UML model that includes the definition and utilization of variables ● The user executes the simulation tool ● The user runs the *DICE FG* ● The user is able to execute again the Simulation tool giving as input the UML model that whose values have been updated by *DICE FG* For the interaction with GreatSPN engine: ● The user configures his/her DICE IDE ● The user creates a UML model ● The user executes the simulation tool ● The user is able to see quality results of the model simulation
	Next Steps: Integration is completed and no further steps are planned. Nevertheless, if other simulation engines that require external connections were considered in the future, new implementations and test cases will have to be planned and developed.

- ***Optimization Plugin***

Tool Name: Optimization tool **Owner: PMI**	
Type 1: Inter-Tool Integration	**Current Status of Integration with other tools:** The optimization tool requires as input a DICE DTSM model including an activity diagram and a deployment model for the application under study and a DICE DDSM model including a deployment model. Such models need to be provided, possibly, for multiple applications which can be based on different big data technologies and be characterised by different QoS constraints and cloud technologies. The optimization tool relies on the model to model transformations implemented within the simulation tool package to obtain a model suitable for performance analysis (i.e., stochastic well-formed nets or queuing networks) that will be performed by GreatSPN or JMT. The integration with the simulation package M2M transformations started recently. In particular, the support to Storm technology is ongoing, while the integration for Hadoop applications will start when the corresponding transformations will be released (planned at M24). The plan is to support also Spark technology by month 30. For what concerns the performance analysis, the optimization relies on GreatSPN and JMT. The tool is already fully integrated with such environments. Future extension will require some work to consider additional performance metrics. For what concerns the output file and integration with DICER, the optimization model will modify its DDSM input file adding the description of the minimum cost deployment identified by the optimization process. Such model will be then stored on the UML project repository and used by the DICER tool to automate the deployment of the applications.
	Interactions (i.e. RestAPI etc.): The transformations from DTSM to the performance models are performed in two stages. In the first stage the optimization tool provides the UML models to the simulation M2M transformation plugin obtaining a PNML description of the application under analysis. In the second stage the PNML description is transformed by the simulation M2M transformation tool into GreatSPN or JMT specific input file format. Then, multiple simulations are run in parallel by the optimization tool and GreatSPN/JMT output files are analysed.
	Functional Description: The optimization tool supports the software architect to explore automatically multiple design configurations in order to identify the deployment of minimum cost that satisfies a priori QoS guarantees (e.g., jobs execution time is lower than a given threshold). The optimization tool is implemented by two components: an Eclipse plugin which acts as GUI and is based on the Façade Design Pattern and allows the end user to provide the relevant input and output files, and a backend web service which performs the design space exploration implementing a local search hill climbing algorithm and interact with the simulators. The interaction of the two components is based on the REST API.

	Integration Testing Scenario: The testing scenario we are envisioning is based on the following steps: 1. DTSM and DDSM files are selected by the end optimization tool end user including an activity diagram and a deployment model and a deployment model respectively. Models are available on the UML project repository. 2. The M2M transformation is then run to obtain the input file for the simulation 3. The simulation is run and output files are processed by the optimization tool 4. The optimization tool updates the DDSM model according to the final optimal solution identified and stores such file on the project repository to make this available to the DICER tool
	Next Steps: On going work focuses on the development of the optimization tool GUI Eclipse plugin and its integration with the simulation package M2M transformation. The integration for Storm technology will be completed by M26, while the integration for Hadoop at M27. The current plan is to support also Spark technology by M30.

- ***Verification Plugin***

Tool Name: D-verT **Owner: PMI**	
Type 1: Inter-Tool Integration	**Current Status of Integration with other tools** D-verT is a fully integrated plugin of the DICE framework. It is based on a client-server architecture that allows for decoupling the verification engine, on the server side, from the front-end on the client side that resides in the DICE IDE. The verification service does not interact with any DICE tool except for the IDE; it is a REST service that can be accessed by means of suitable APIs supplying: ● the execution of a verification task; ● lookup functionalities to inspect the result of the verification. The input to the service is a JSON object which defines an instance of a verification problem. The front-end of D-verT builds the instance from the information specified by the user in the IDE when verification is launched. An instance describes the DIA model undergoing verification and some user information that are useful to configure the verification engine. The client is the unique component of D-verT interacting with the DICE IDE, as the latter includes all the necessary UML diagrams to carry out verification. To this end, the client implements a model-to-model transformation that translates the DTSM diagram of the application into the JSON descriptor. Running verification involves, first, the translation of the application model and, afterwards, a REST call to the server that is executed through a POST method conveying the JSON descriptor. The client is then able to obtain the status of the verification task by performing a GET request to the server. It is also possible to download results and configuration files by means of a specific call.
	Interactions (i.e. RestAPI etc.): D-verT does not interact with other DICE tools.
	Functional Description: The D-verT RESTful service on the server is structured through the following methods: ● Launch verification task: o RESTful: ▪ POST /longtasks ▪ input: JSON descriptor of a verification instance ▪ output: the URL through which it will be possible to track the status of the task. ▪ purpose: the method creates and launches a thread running the verification engine. o No CLI counterpart ● Status of a verification task: o RESTful: ▪ GET /status/**TASK_ID** ▪ output: a JSON descriptor specifying the status of the task with identifier **TASK_ID**. Possible values are PENDING, PROGRESS, SUCCESS, REVOKE and TIMEOUT. ▪ purpose: allows the user to obtain information on the verification tasks that were launched. o No CLI counterpart ● List of tasks: o RESTful: ▪ GET /task_list

	▪ output: JSON containing the list of all the task status object, ▪ purpose: provides information about the status of all the tasks in the system. o No CLI counterpart ● Task-related files: o RESTful: ▪ GET /tasks/**RELATIVE_FILE_PATH** ▪ output: the file located at the specified URL ▪ purpose: allows the client to get all the relevant files related to a specific task, such as configuration files, output traces, graphical representation. o No CLI counterpart In the next few months, D-verT will be provided with new functionalities that allow designers to verify temporal properties of Spark jobs. The architecture of the service will not be modified; however, the POST /longtasks method will be adapted to fit the new verification requirements characterizing Spark verification. To this end, the content of the JSON structure will be modified in order to allow the definition of the verification parameters for the Spark analysis.
	Integration Testing Scenario: The interaction between the the client and the server was tested in the following scenario. 1. The client calls */Longtasks* method by proving a JSON descriptor for the verification task. 2. The client checks the status of a specific verification process by invoking */status/**TASK_ID*** 3. The client gets the list of all the tasks started in the server.
	Next Steps: No further steps related to the integration of the tool are planned.

- ***Monitoring Platform***

Tool Name: DICE Monitoring platform (DMON) **Owner: IeAT**	
Type 1: Inter-Tool Integration	**Current Status of Integration with other tools:** The DICE Monitoring platform (DMON) is responsible for collecting and serving performance metrics of big data platforms which are at the base of Data Intensive applications (DIAs). We currently support monitoring of; YARN, Spark, Storm as well as Cassandra and MongoDB. It is easy to see that DMON servers as the main source of performance metrics for the rest of the DICE toolchain via a RESTful API. As such we have already integrated with the following tools: ● Anomaly Detection Tool: All performance metrics collected are served via the DMON RESTful API to then be checked for anomalies. The anomaly detection tool is able to collect raw as well as aggregated metrics. By offloading some preprocessing tasks to DMON we are able to only execute data processing tasks on a method by method basis (normalization, dimensionality reduction etc.) It is also important to mention that the detected anomalies are pushed into a special index from DMON. Thus allowing the reuse of the same REST resources for querying performance metrics for anomalies. Also, the predictive models created by the anomaly detection tool are also sent to DMon for storing and later retrieval. ● Delivery Tool: The collection of the metrics must be automatic from the moment the DIA gets deployed and started. The Delivery tool is responsible for DIA deployment and configuration as well as the installation and initial setup of the dmon-agent component on all deployed hosts. All addresses and ports are standardized and documented and can be setup by the administrator in the Deployment Service. Application versioning is done via tags assigned to the DMON by the Delivery tool. ● Configuration Optimization (CO): Is able to collect performance related metrics via the query REST resources from DMON. This integration has been tested in the case of Storm DIA's. Special REST resources for resource polling periods are also used by CO. ● Filling-the-Gap module (FG): is able to collect performance related metrics (resource consumption and throughput). Concretely, FG pulls monitoring data from DMon platform that comes from YARN History server (e.g., number of jobs, execution time) as well as system metrics (e.g., CPU utilization) **Planned integrations:** ● Trace Checking Tool (TCT): The trace checking is able to utilise DMON to get performance related metrics. However, because it also requires raw log files it is able to request them via special REST resource in DMON. These allow the asynchronous fetching and serving of the required logs.

Interactions (i.e. RestAPI etc.):

DICE Monitoring Platform - Anomaly Detection Tool: The Anomaly Detection tool uses the query REST API Resources for fetching performance monitoring data. It also is able to fetch different information about the monitored hosts. These resources are:

- The anomaly detection tool requires the number and name of all registered hosts that are monitored. It get them issuing the request found below
  - GET /v1/overlord/nodes
    - Output:JSON that contains the current FQDN and IP of all registered nodes.
- Metrix are queried using the simple REST resource or the aggregated query resource.
  - POST /v2/observer/query/{csv/json}
    - Input: JSON that contains aggregation type, index name, aggregation interval, size, start and end time
- Detected anomalies and predictive models are stored in the monitoring platform

DICE Monitoring Platform - DICE Delivery Tool: The Delivery tool uses the DMON REST API to register and report different host metadata:

- Nodes, roles of the nodes and node metadata of the the deployed nodes. DMON accepts a JSON documents:
  - PUT /dmon/v1/overlord/nodes
    - input: JSON structure containing node data (name, IP, etc.)
    - purpose: register new nodes with DMon to be monitored
  - PUT /dmon/v1/overlord/nodes/roles"
    - input: JSON structure containing node names (as registered in the previous call) and their roles (e.g., storm, spark, etc.)
    - purpose: assign a role for DMon to know what to monitor
- application ID of the application to be deployed:
  - PUT /dmon/v1/overlord/application/**DEPLOYMENT_ID**

Configuration Optimization - DICE Monitoring: The configuration optimization tool uses the query REST Resources in order to get the required Storm performance metrics:

- The required resources are:
  - POST /v1/observer/query/{csv/json/plain}
    - you can specify the return format csv, json or plain text
    - input:response size, ordering, query string, query period start and finish
    - output is depending on the request format

Filling-the-Gap tool - DICE Monitoring: The FG tool query uses the query REST resources in order to get system and YARN History server related performance metrics:

- POST /v1/observer/query/{csv/json/plain}
  - you can specify the return format csv, json or plain text
  - input:response size, ordering, query string (e.g. type: "collectd" AND plugin: "CPU" OR "type: "yarnHistory"), query period start and finish
  - output is depending on the request format

Functional Description:

DMon platform provides a REST API structured in two sections:

- First we have the management and deployment/provisioning component called Overlord(Monitoring Management API). It is responsible for deployment and management of the Monitoring Core components: ElasticSearch, Logstash Server and Kibana. Besides it is also responsible for the auxiliary component management and deployment. These include: Collectd, Logstash-forwarder.
- Second, we have the interface used by other applications to query the DataWarehouse represented by ElasticSearch. This component is called Observer. It is responsible for returning the monitoring metrics in various formats (CSV, JSON, simple output).

Overlord (Monitoring Management API)

The Overlord is composed from two major components:

- Monitoring Core represented by: ElasticSearch, LogstashServer and Kibana
- Monitoring Auxiliary represented by: Collectd, Logstash-Forwarder

More information ragarding the API can be found in https://github.com/dice-project/DICE-Monitoring/wiki/Getting-Started

	Integration Testing Scenario: As DMON is at the end of the DICE toolchain integration with other tools is almost always one sided. Meaning that most tools query DMON in order to get performance data however, DMON does not require any input from the tools in order to fulfil its function. Notable exceptions are the Delivery Tool which registered the nodes to be monitored and the Anomaly Detection Tool that stores both detected anomalies as well as created predictive models in DMON. For both functionality and load testing of the RESTful API we use the locust.io library.
	Next Steps: The development of DICE Monitoring platforms ends M24. We will continue to support the integration of DICE demonstrators and other DICE tools, but no further features are foreseen for the platform.

- ***Anomaly Detection***

Tool Name: Anomaly Detection Tool **Owner: IeAT**	
Type 1: Inter-Tool Integration	**Current Status of Integration with other tools:** The Anomaly Detection Tool (ADT) is responsible for the detection of contextual performance anomalies in DIA's execution. It can accomplish this using supervised, unsupervised and semi-supervised methods. In short the ADT is integrated with: ● DICE Monitoring Platform: In order to fulfill its function it requires access to the monitored data. Because of this requirement ADT is tightly integrated with DMON. Also, the created predictive models and detected anomalies are stored in DMON.
	Interactions (i.e. RestAPI etc.): *DICE Monitoring Platform - Anomaly Detection Tool:* The Anomaly Detection tool uses the query REST API Resources for fetching performance monitoring data. It also is able to fetch different information about the monitored hosts. These resources are: ● The anomaly detection tool requires the number and name of all registered hosts that are monitored. It get them issuing the request found below o GET /v1/overlord/nodes ▪ Output:JSON that contains the current FQDN and IP of all registered nodes. ● Metrix are queried using the simple REST resource or the aggregated query resource. o POST /v2/observer/query/{csv/json} ▪ Input: JSON that contains aggregation type, index name, aggregation interval, size, start and end time ● Detected anomalies and predictive models are stored in the monitoring platform

	Functional Description:
	Command Line arguments
	
	$ python dmonadp.py -h
	● h -> This argument will list a short help message detailing some basic usage of for ADT
	$ python dmonadp.py -f <file_location>
	● f -> This argument will ensure that the selected configuration file is loaded.
	$ python dmonadp.py -e <es_endpoint>
	● e -> This argument allows the setting for the elasticsearch endpoint.
	NOTE: It is important to note that in future versions ADT will be integrated with DMon and will be able to query the DMon query endpoint not just the elasticsearch one.
	$ python dmonadp.py -a <es_query> -t -m <method_name> -v <folds> -x <model_name>
	● a -> This represents the query that is to be issued to elasticsearch. The resulting data will be used for training. The query is a standard elasticsearch query containing also the desired timeframe for the data.
	● t -> This instructs ADT to initiate the training of the predictive model.
	● m -> This represents the name of the method used to create the predictive model.
	● v -> This instructs ADT to run cross validation on the selected model for a set of defined *folds*.
	● x -> This allows the exporting of the predictive model in PMML format.
	NOTE: The last two arguments, v and x, are optional.
	$ python dmonadp.py -a <query> -d <model_name>

- d -> This enables the detection of anomalies using a specified pre-trained predictive model (identified by its name)

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

- ESEndpoint -> sets the current endpoint for DMON, it can be also in the form of a list if more than one elasticsearch instance is used by DMON
- ESPort -> sets the port for the elasticsearch instances (*NOTE*: Only used for development and testing)
- DMonPort -> sets the port for DMON
- From -> sets the first timestamp for query (*NOTE*: Can use time arithmetic of the form "now-2h")
- To -> sets the second timestamp for query
- Query -> defines what metrics context to query from DMON
    - each metric context is divided into subfields as follows:
        - yarn-> cluster, nn, nm, dfs, dn, mr
        - system -> memory, load, network
        - spark -> not for this version (v0.1.0)
        - storm -> not for this version (v0.1.0)
- Nodes -> list of desired nodes, if nothing specified than uses all available nodes
- QSize -> sets the query size (number of instances), if set to 0 then no limit is set
- QInterval -> sets aggregation interval

NOTE: Each large context is delimited by ";" while each subfield is divided by ",". Also *QInterval* must be set to the largest value if query contains both system and platform specific metrics. If the values are two far apart it may cause issues while merging the metrics.

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

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

The Filter section is used to filter the collected data. The options are as follows:

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

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

- Method -> sets the desired anomaly detection method to be used (available 'skm', 'em', 'dbscan')
- Type -> type of anomaly detection
- Export -> name of the exported predictive/clustering model
- Load -> name of the predictive/clustering model to be loaded

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

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

- head -> sets heap space max value
- checkpoint -> If set to false, all metrics will be saved as csv files into the *data* directory otherwise all data will be kept in memory. It is important to note the if the data is kept in memory processing will be much faster.
- delay -> sets the query delay for point anomalies
- interval -> sets the query interval for complex anomalies
- interval-> if set to True the anomalies index will be reset and all previously detected anomalies will be deleted. resetindex:false

The MethodSettings section of the configuration files allows the setting of different parameters of the chosen training method. These parameters can't be set using the command line arguments

DBSCAN is a density based data clustering algorithm that marks outliers based on the density of the region they are located in. For this algorithm we support two versions with the following method settings.

DBSCAN Weka

	● E -> epsilon which denotes the maximum distance between two samples for them to be considered as in the same neighborhood (default *0.9*) ● M -> the number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself (default *6*) ● D -> distance measure (default *weka.clusterers.forOPTICSAndDBScan.DataObjects.EuclideanDataObject*) ● I -> index (default *weka.clusterers.forOPTICSAndDBScan.Databases.SequentialDatabase*) **DBSCAN scikit-learn** ● eps -> epsilon which denotes the maximum distance between two samples for them to be considered as in the same neighborhood (default *0.5*) ● min_samples -> The number of samples (or total weight) in a neighborhood for a point to be considered as a core point. This includes the point itself (default *5*) ● metric -> metric used when calculating distance between instances in a feature array (default *euclidean*) ● algorithm -> the algorithm used by the nearest-neighbour module to compute pointwise distance and find nearest neighbour (default *auto*) ● leaf_size -> leaf size passed to BallTree or cKDTree, this can affect the speed of the construction and query, as well as the memory required to store the tree (default *30*) ● p -> the power of the Minkowski metric used to calculate distance between points (default *None*) ● n_jobs -> the number of parallel jobs to run (default *1*, if *-1* all cores are used) **IsolationForest** The IsolationForest 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. Since recursive partitioning can be represented by a tree structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node. This path length, averaged over a forest of such random trees, is a measure of normality and our decision function. Random partitioning produces noticeably shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies. ● n_estimators -> number of base estimators in the ensemble (default *100*) ● max_samples -> number of samples to draw to train each base estimator (default *auto*) ● contamination -> the amount of contamination of the dataset, used when fitting to defined threshold on the decision function (default *0.1*) ● max_features -> the number of features to draw to train each base estimator (default *1.0*) ● boostrap -> if true individual trees are fit on random subsets of the training data sample with replacements, if false sampling without replacement is performed (default *false*) ● n_jobs -> the number of jobs to run in parallel for both fit and predict, (default *1*, if *-1* all cores are used)
	Integration Testing Scenario: The integration was tested in 2 main scenarios: ● Querying for performance metrics for Yarn, Storm, System Metrics o Future development will focus on adding Spark, Cassandra and MongoDB support ● Pushing detected anomalies to DMON
	Next Steps: ● **M30:** Final version of anomaly detection will be delivered. No further integration will be required until that point.

o **Trace Checking**

Tool Name: Dice-TraCT **Owner: PMI**	
Type 1: Inter-Tool Integration	**Current Status of Integration with other tools:** Dice-TraCT has not been integrated yet in the DICE framework and it can only be used as a standalone tool. However, the integration has been recently started and it will be based on the same client-server architecture as the one implementing the verification tool. A client-server architecture allows for decoupling the trace checking engine, residing on the server side, from the front-end that will be realized as an Eclipse plugin. The interaction schema between the client side and the server side is similar to the one of the verification tool. However, Dice-TraCT is part of the monitoring services as it performs analysis of log traces coming from the deployed applications. Hence, the server side has to implement suitable functionalities allowing the interaction with the monitoring services. In particular, the server will

	interact with the monitoring through methods that allow Dice-TraCT to retrieve the logs of the application undergoing analysis. To this end, Dice-TraCT should first check the existence of an active monitoring agent on the nodes of the application that one wants to verify. D-mon provides look-up functionalities that return the list of active Dmon agents installed on the nodes. Based on the outcome of the inspection, Dice-TraCT can further proceed. First, it verifies the existence of a log trace that matches the temporal requirements specified by the user (in the client side) when the verification task was launched. In the positive case, Dice-TraCT collects a compressed archive by calling a D-mon POST method that streams the file from the monitoring platform. Afterwards, the tool pre-processes the trace and runs the trace-checking engine by inputting the trace and the property to be checked. At any time, after the trace-checking task was invoked, the user can probe the server to check whether the analysis is completed and to collect the result. The interaction is allowed by means of a REST method that is provided by the server and that is called by the client from the DICE IDE.
	Interactions: ***Intra-tool interaction*** The trace checking service is a REST service that can be accessed by means of suitable APIs supplying: ● the execution of a trace checking task; ● lookup functionalities to inspect the result of the analysis. The input to the service is JSON file which defines an instance of a verification problem. The front-end of Dice-traCT builds the instance from the information specified by the user in the IDE when trace checking is launched. An instance describes the trace checking problem and some user information that are useful to configure the trace checking engine. In particular, the problem can be one of the following: ● temporal analysis of some non-functional parameters of Storm topologies; ● temporal analysis of log traces of Posidonia use case; ● temporal analysis of privacy requirements in Cassandra nodes. The client interacts with the DICE IDE and behaves similarly to the client of D-verT: it implements a model-to-model transformation that translates the UML diagram of the application into the JSON descriptor. Running trace checking involves, first, the translation of the application model and, afterwards, a REST call to the server, that is executed through a POST method conveying the JSON descriptor. The outcome of the analysis can be obtained by the user which exploits a pull request that is invoked on the server to retrieve the result. ***Inter-tool interaction*** Dice-traCT requires log traces of the monitored applications to carry out the trace checking analysis. Therefore, the server part of the tool must interact with D-mon to collect the log before running the trace checking engine. The interaction is based on REST methods calls that are invoked by the server when the Dice-traCT user issues a request for activating the analysis. The target service in D-mon is Observer, the component that is responsible for returning the monitoring metrics. **Dice-traCT - DICE Monitoring Tool:** The following methods will be used by the Dice-traCT server to operate the analysis: 1. GET /v1/observer/nodes a. input: empty b. output: return the current monitored nodes list (node FQDN and current node IP). 2. GET /v1/observer/nodes/{nodeFQDN}: a. input: the node name b. output: return information of a particular monitored node. 2. POST /v1/observer/query/{csv/json/plain}: a. input: JSON structure containing the query and the time window defining the portion of the trace to be analyzed. b. returns the required metrics which is specified in the JSON payload through the field value "queryString". Beside the previous methods, more specific calls are used to collect data that are related to a specific technology. For instance, in the case of Storm topology, Dice-traCT will query the monitoring platform with the following methods: ● GET dmon/get_storm_logs o input: empty o output: return the list of the logs that can be collected from the monitoring platform. ● GET dmon/v1/overlord/storm/logs/<log_name>

	o input: a log name (the list of all the log names can be obtained by the previous method). o output: return a tar file containing all the worker logs that are saved in the log_name file.
	Functional Description: The functionalities that the trace checking tool can offer mostly depend on the monitoring features implemented in DMON. The main ones will be implemented as RESTful calls on the server, that will be structured with the following baseline methods. ● Monitored node: o RESTful: ▪ GET /monitored/<nodename> ▪ input: name node that is used to check if the node is monitored ▪ output: a JSON structure containing information of the node ▪ purpose: check the presence of a monitoring agent on the nodes that are going to be verified. o No CLI counterpart ● Run a task: o RESTful: ▪ POST /runtask ▪ input: a JSON structure defining the descriptor of a trace checking instance (it contains the description of the property to be checked such as non-functional properties of Storm nodes or privacy requirements and the nodes that are involved in the verification). ▪ output: a JSON structure containing information related to the trace checking task ▪ purpose: the methods creates and launches a thread running the trace checking engine. o No CLI counterpart ● Status of a verification task: o RESTful: ▪ GET /status/<task_id> ▪ input: a task identifier ▪ output: a JSON descriptor specifying the status of a tasks. Possible values are PENDING, PROGRESS, SUCCESS, REVOKE and TIMEOUT. ▪ purpose: allow the user to obtain information on the trace checking tasks that were launched. o No CLI counterpart
	Integration Testing Scenario: The interaction between the the client and the server will be tested in the following scenarios. We assume that DMON is active. 1. [Monitored node]: a. the user selects the names of the nodes that are going to be verified through a selection list in the trace checking configurator window in the IDE. b. the user query Dice-traCT to verify that the selected nodes are currently monitored. c. the user is notified of the nodes that are monitored. 2. [Run a task]: a. the user specifies in IDE the nodes and the property to be verified. b. the user launches the trace checking analysis. c. the user checks (in a pull way) from time to time whether the analysis is terminated. 2. [Status]: same as “Run a task”.
	Next Steps: The integration with D-mon will be completed by April 2017 (M28).

- ***Enhancement Tool***

Tool Name: Enhancement Tool **Owner: IMP**	
Type 1: Inter-Tool Integration	**Current Status of Integration with other tools:** Enhancement tool aims at feeding results back into the design models to provide guidance to the developer on the quality offered by the application at runtime. Thus, DICE Enhancement tool needs to integrate with DICE Monitoring tool to obtain runtime data on the DIAs. Currently, DICE FG, one of the major modules of DICE Enhancement tool, can issue JSON queries to the DICE

Monitoring Platform (DMON) to obtain the quality metric of Big Data scenario (e.g., Storm, Spark)

Interactions (i.e. RestAPI etc.):
Enhancement tool is a pipeline element that obtains DIAs runtime information from DICE Monitoring tool and previous usage of the DICE UML model and outputs updated UML model with the refactoring decisions to the DICE IDE. Thus, the interactions mainly happens between the DICE Enhancement tool (i.e., DICE FG) and DICE Monitoring tool.

DICE FG - DICE Monitoring tool: DICE FG uses DICE Monitoring Tool's RESTful interface to report:

- CPU utilization, Response time and Throughput, ect. The following example shows the structure of the JSON query string sending to DMON:

```
DMON{
        "fname": "output",
        "index": "logstash-*",
         "ordering": "asc",
         "queryString": "type:\"collected\" AND plugin:\"CPU\" OR "type:\"yarn-history\""
OR "type:\"yarn_jobstatsks\""
    }
```

Functional Description:
DICE Enhancement tool includes two major tools, **DICE Filling the Gap** (FG) and **DICE Anti-Patterns & Refactoring** (APR). DICE FG is able to estimate and fit application parameter related to execution times of requests and correspondingly annotates UML models. DICE APR (due M30) will suggest architecture refactoring decisions to developer if performance anti-pattern detections are detected. Currently, the DICE APR is still under development. Once the UML model is parameterized, an APR-specific launch configuration will be invoked from the DICE IDE run-configuration panel. The run-configuration in question will invoke the APR back-end and performs the model-to-model transformation that parses the diagrams and returns a layered queueing networks (LQN) model for performance anti-pattern detection.

DICE Filling the Gap Provides:

- UML parameter estimation
  - Parameter fitting
    - Input: location of data trace, target distribution to be *fitted*, output UML model, tags to be annotated inside the UML models
    - Output: fitted distribution annotation in specified UML model tags
  - Parameter inference
    - Input: location of data trace, target distribution to be *estimated*, output UML model, target tags inside the UML models
    - Output: estimated distribution annotation in specified UML model tags
  - Extract-transform-load
    - Input: log data for a specific DICE technology (e.g., Spark)
    - Output: data extraction and DICE-FG template configuration file
    - Purpose: convert technology-specific log files into DICE-FG-compliant data files (e.g. JSON) and configuration files

DICE Anti-Patterns & Refactoring Provides:

- Model-to-Model Transformation (UML model to LQN model),
  - Model acquisition:
    - Input: location of the source (UML model) and target model (LQN model)
    - Purpose: identifying the annotated UML model and specifying the location where the corresponding LQN model will be stored.
  - Pre-Transformation:
    - Input: annotated UML model
    - Output: updated UML model with *inPartition* information
    - Purpose: Identifying the *controlflow* among the different partition of the activity diagram.
  - Main Transformation:
    - Input: updated UML model
    - Output: initial LQN model
    - Purpose: Generating the initial LQN model in XML format from the annotated UML model

	○ Post Transformation ▪ Input: initial LQN model ▪ Output: formalized LQN model ▪ Purpose: Modifying the initial LQN model to conform to the XML schema of LQN for adapting to the existing LQN solver tool.
	Integration Testing Scenario: The following integration testing scenarios are described with the **Gherkin** language. Some keywords used in our scenarios are ***Given***, ***When***, ***Then*** and ***And***. **DICE Enhancement tool – DICE Monitoring Tool** **Given** the DMON Service is available at its address **And** a developer has an UML model in the file 'Model.uml' **When** a developer uses DICE Enhancement tool to query DMON Observer with a JSON string for runtime information **Then** DICE Enhancement tool should receive a JSON file **Then** DICE Enhancement tool should parameterize the 'Model.uml' with performance and reliability parameters **Then** the updated uml model should be transformed into LQN model for Anti-patterns detection **DICE Enhancement tool – DICE Simulation Tool** **Given** the simulation service is available to be invoked **And** developer has an UML model in the file 'Model.uml' **When** developer uses DICE Enhancement tool to parameterize the 'Model.uml' **with** performance and reliability parameters **Then** simulation tool should be able to use those parameters for simulation **DICE Enhancement tool – DICE Optimization Tool** **Given** the optimization service is available to be invoked **And** developer has an UML model in the file 'Model.uml' **When** developer uses DICE Enhancement tool to parameterize the 'Model.uml' **with** performance and reliability parameters **Then** optimization tool should be able to perform the quality analysIS later.
	Next Steps: DICE Enhancement tool will external integrate with the DICE IDE, thus, it will not integrate with other inter-tools directly except DICE Monitoring tool. The following capability will be provided: **DICE FG – Invocation**: DICE FG will allow the invocation of an analysis from the IDE via a simple REST interface that specifies the location of the DICE FG configuration file and the relevant UML models to be updated. At the end of the invocation DICE FG will update the target models directly at their indicated location.

- ***Quality Testing***

Tool Name: Quality Testing Tool **Owner: IMP**	
Type 1: Inter-Tool Integration	**Current Status of Integration with other tools:** Quality testing (QT) tool aims at automating the execution of tests in a streaming-based application. In the initial M24 release, the tool provides a Java library (QT-LIB) for the end user to specify a relevant workload for the test, and a binary tool (QT-GEN) to generate a JSON trace that is similar, but nevertheless non-identical, to a user-provided trace, for the sake of producing alternative workloads for load testing. Both tools are substantially independent of the rest of the DICE toolchain, therefore there is no envisioned need for a complex inter-tool integration. QT-LIB will operate inside the IDE, as any standard Java library included by the developer in his project. It will be shipped directly inside the DICE IDE and automatically included in new projects. QT-GEN is a simple command to be run on the command line, it does not require integration.
	Interactions (i.e. RestAPI etc.): **DICE QT - DICE IDE:** Integration of DICE QT jar in the DICE IDE will be sufficient for the end user to access all the relevant API features. **DICE QT - DICE Continuous Integration:** It is possible to highlight the test results directly in the

	Jenkins dashboard, which is part of the DICE Continuous Integration toolchain. The user has to provide a TOSCA blueprint for the application's "test mode", which will be executed in the DICE Continuous Integration's Quality Testing job. The DICE Continuous integration uses the QT-WRAPPER to obtain the performance metrics of the "test mode" run.
	Function Description: **DICE QT Provides:** l Load injection for streaming-based DIAs ○ Trace generation from a reference JSON trace Input: JSON trace, desired output trace length, configuration options. Output: new JSON trace with desired characteristics. ○ Custom spout load injection ■ Input: input configuration file or MongoDB query instance to be used to extract the workload specification. ■ Output: automatic execution of experiments on the DIA once deployed via the DICE deployment service.
	Integration Testing Scenario: The following integration testing scenarios are described with the **Gherkin** language. Some keywords used in our scenarios are ***Given***, ***When***, ***Then*** and ***And***. It can be used in a Cucumber executable specifications tool. **DICE Quality testing tool – DICE IDE** **Given** the developer is using the DICE IDE **And** s/he is writing a performance test **Then** DICE QT functions are readily available to the user without generating compilation errors in the DICE. **DICE Quality testing tool – DICE CI** **Given** the developer has deployed the DIA in testing mode **And** the QT experiment has concluded **Then** DICE CI's Jenkins dashboards shows the relevant execution time of the experiment and history of the past experiments. **DICE Quality testing tool – DICE CI** **Given** the developer has deployed the DIA in testing mode **And** the QT experiment has concluded **Then** DICE CI's Jenkins dashboards shows the relevant execution time of the experiment and links to the DICE Monitoring Platform to simplify the user access to the relevant monitoring data.
	Next Steps: **M30**: IMP will release the final version of DICE QT including DICE CI integration.

- ***Configuration Optimization***

Tool Name: Configuration Optimization Tool **Owner: IMP**	
Type 1: Inter-Tool Integration	**Current Status of Integration with other tools:** Configuration optimization (CO) tool aims at automatically optimizing the configuration of a data-intensive application (DIA) by means of auto-tuning. Thus, the CO tool needs to integrate with the DICE Delivery tools to instantiate and configure the application and the CO tool instance, with the DICE Continuous Integration to schedule the experiments, and with the DICE Monitoring tool to obtain monitoring data on the results of the experiments on the DIAs. Currently, CO is integrated with DICE Continuous Integration, through which it can be deployed using a Chef cookbook.

	Interactions (i.e. RestAPI etc.):
	DICE CO - DICE Deployment service: CO can be automatically installed using a Chef cookbook included in the DICE Chef Repository. This requires the inclusion of a configuration file to inform CO about the IP of relevant DICE services:
	{
	"dice-h2020": {
	"conf-optim": {
	"ds-container": "4a7459f7-914e-4e83-ab40-b04fd1975542"

<table>
<tr><td></td><td>
<pre>
},
"deployment-service": {
  "url": "http://10.10.50.3:8000",
  "username": "admin",
  "password": "LetJustMeIn"
},
"d-mon": {
  "url": "http://10.10.50.20:5001"
}
}
}
</pre>
DICE CO - DICE Monitoring platform: DICE CO receives the monitoring platform IP and port via the above configuration file and upon completion of the integration will query it to obtain experimental data. We point to the DICE-FG tool description for an example of JSON query for monitoring data.
DICE CO - DICE Continuous integration: DICE CO will instantiate via REST API calls Jenkins jobs that will start the execution of the experiments at pre-defined times. For example, an end user will be able to schedule a nightly run.
</td></tr>
<tr><td></td><td>
Function Description:
DICE Configuration Optimization Provides:

Auto-tuning

Optimal configuration search

Input: URL/port of application, deployment-service, monitoring service, experimental time limit, parameters to be used for the experimentation.
Output: optimal assignments of the configuration parameters.

</td></tr>
<tr><td></td><td>
Integration Testing Scenario:
The following integration testing scenarios are described with the Gherkin language. Some keywords used in our scenarios are Given, When, Then and And. It can be used in a Cucumber executable specifications tool.
DICE Configuration tool – DICE Deployment Service
Given that the DIA once started automatically performs an experiment (e.g., this can be done using the DICE Quality Testing tools)
And that the developer specifies a range of interest for a given target technology
When a developer uses DICE Configuration tool to optimize the DIA
Then DICE CO runs automatically the experiment on the DIA, using Deployment Service to redeploy the application with a different configuration.

DICE Configuration tool – DICE Monitoring Platform
Given that CO has completed an experiment on the DIA
Then DICE CO obtains the results of the last experiment from the DICE Monitoring Platform
And decides either to stop and report results or run the successive experiment.

DICE Configuration tool – DICE Continuous Integration
Given that the developer wants to start a DICE CO experiment
And s/he has indicated the set of configuration parameters to analyse via the IDE
Then a rule is installed in the DICE CI Jenkins instance to start the experiment automatically.

DICE Configuration tool – DICE Continuous Integration
Given that DICE CO has completed an auto-tuning experiment
Then results are reported to the DICE CI Jenkins for visualization to the user via the Jenkins platform.
</td></tr>
<tr><td></td><td>
Next Steps:
IMP is now defining templates for specific technologies to be instantiated in Jenkins as part of the DICE CI integration.
</td></tr>
</table>

- ***Fault Injection***

Tool Name: Fault Injection Tool **Owner: Flex**	
Type 1: Inter-Tool Integration	**Current Status of Integration with other tools:** Currently the DICE Fault Injection Tool (FI)T has no direct in integration with the other tools of the DICE framework. However the created faults can be detected via the DICE monitoring tools. The monitoring tools will then alert the user as to the state of the VM and application. This data can then be used to resolve issues within the applications.

	The DICE DUSK collaboration will logically combine the DICE Quality Testing Tool (DICE QT). The DICE QT will provide a range of faults and VMs. The FIT will then use this information to create the faults within the VMs.
	Interactions (i.e. RestAPI etc.): The user interacts with the FIT using the command line interface. Once the fault has been started the log ach tool owner to show what interaction is expected for each tool with other tools and which types of interfaces can be used.
	Functional Description: FIT allows VM Admins, application owners and Cloud Admins to generate various faults. It runs independently and externally to any target environment and allows the tool user to generate faults either at VM or cloud level. The purpose is to allow cloud platform owners a means to test the resiliency of a cloud installation as an application target. **Input Functions** To use the FIT a user would first deploy the FIT as an executable jar file, from here a user can call the jar file and pass the following parameters - > -f,--file <arg> Load from properties file. > -h,--help Shows help. > -m,--stressmem <memorytesterloops,memeorytotal,host,vmpassword,sshkeypath> Stress VM Memory. > -r,--randomVM <cloudusername, cloudpassword, cloudUUID,cloudapiurl> Shutdown random VM within FCO. > -s,--stresscpu <cores,time, host, vmpassword, sshkeypath> Stress VM CPU. > -b,--blockfirewall <host,vmpassword,sshkeypath> Block external communication from Firewall. > -k,--killservice <host,vmpassword,service,sshkeypath> Stop service running on VM. > -w,--whitelist <cloudusername, cloudpassword, vmpassword, filepath> Shutdown random VM within FCO from testfile list > -n,--stressnetwork <host, vmpassword, iperfserver, time, sshkeypath> High bandwitdh usage of VM > -d,--diskstress <host, vmpassword, n,memeorytotal, loops, sshkeypath> High Disk usage of VM > -y,--ycsb <host, vmpassword, workloadname, threads, sshkeypath> Start ycsb workload on MongoDB db > -j,--jmeter <hostm vmpassword,workloadname,planname,sshkeypath> Start pre-made Jmeter path Example command: java - jar FIT.jar --stressmem 2 512m ubuntu@111.222.333.444 -no c://SSHKEYS/VMkey.key **Output Functions** From here the fault is passed and instigated on the target VM/Cloud. To access the VM level and issue commands the DICE FIT uses JSCH[5] to SSH to the Virtual Machines and issue the commands. By using JSCH, the tool is able to connect to any VM that has SSH enabled and issue commands as a pre-defined user. This allows greater flexibility of commands as well as the installation of tools and dependencies before running any of the below functions. ● Shutdown node ● High CPU for Node ● High Memory usage for Node ● High Bandwidth usage. ● Shutdown random VM ● High CPU for VM ● High Memory usage for VM ● Block VM external access ● High Bandwidth usage. ● High Disk I/O usage ● Stop service running on VM ● Shutdown random VM from whitelist provided by user ● YCSB on VM running MongoDB to begin workload test ● Run JMeter plan In addition the FIT generates internal logs which can be used for time analysis of when faults are started and the duration of these faults on each VM. These log files are stored locally where the tool has been ran. From a user perspective the fault results can then be detected by any monitoring services that have

	been deployed onto the VM and passed to any reasoning engine.
	Integration Testing Scenario: First an application will deployed onto the target cloud platform. Once this application has been deployed the DICE monitoring will be deployed onto the VMs. Once the monitoring has been deployed the user can then use the FIT.
	Next Steps: Further development will focus on a GUI access for the fault injection tool. Other aspects that will be investigate is a logical pairing of tools in the form of DICE DUSK & integration. In addition the DICE deployment service will be used which would give the users a list of relevant nodes only applicable for an application for a fault to be deployed to. The DDS will provide list of a living nodes and their IPs and types/roles, this will then be used by the Fault Injection tool to generate faults.

- ***Delivery Tool***

Tool Name: Delivery Tool **Owner: XLAB**	
Type 1: Inter-Tool Integration	**Current Status of Integration with other tools:** The DICE Delivery Tool is at the centre of turning DICE deployment models into a runtime. As such, we have already integrated it with the following DICE tools: ● DICE Monitoring Tool: enhancement of deployment on the basis of the DIA's runtime can only be evaluated if metrics of the runtime are available. The collection of the metrics must be automatic from the moment the DIA gets deployed and started. We have therefore already integrated the DICE Delivery Tool with the DICE Monitoring Service by building into the DICE TOSCA Library (a DICE Delivery Tool component) the triggers to register each of the supported nodes with the DMon. Parameters of the DMON's end points (addresses and ports) have standardised and documented names, so that the Administrator can set them at the DICE Deployment Service. Users can then choose to flag the node templates in their TOSCA blueprints as monitored. Currently, we support monitored Storm nodes and Storm topology, Spark nodes and Cassandra nodes. ● Configuration Optimization (CO): integrated with the DICE Continuous Integration (CI) to ensure automated execution of the search for the optimal configuration. The CO can, if so configured, take half an hour or even more to perform its search, so it is important that this process occurs unattended and without intruding on the users' interactive processes. Thus execution by CI is suitable, because CI can also make sure to collect all the search's outcomes. CO, in turn, is fully integrated with the DICE Deployment Tool by relying on it for (re)deployments of the application. Planned integration includes: ● DICER: the DICE Delivery Tool and DICER are naturally complementing each other, because DICE Delivery Tool from DICER enables a one-click deployment of a DDSM, while DICER's model to text transformation functionality exploited by the DICE Deployment Service widens the supported input formats for the DICE Deployment Service. ● Quality Testing Tool (QT): from the CI, usage of the QT will be similarly powerful as the use of the CI. ● Enhancement Tool, Anomaly Detection: they do not interact directly with DICE Delivery Tool. However, indirectly it depends on the application tag that the DICE Delivery Tool assigns to the application in the DMon. As a result, the Enhancement Tool and the Anomaly Detection tool need to be aware of this application tag before they can access the DMon for specific DIA's version's data.
	Interactions (i.e. RestAPI etc.): **DICE Delivery Tool - DICE Monitoring tool**: DICE Delivery Tool uses DICE Monitoring Tool's RESTful interface to report: ● Nodes, roles of the nodes and node addresses as the nodes get deployed. DMon receives a JSON structure: o PUT /dmon/v1/overlord/nodes o input: JSON structure containing node data (name, IP, etc.) o purpose: register new nodes with DMon to be monitored o PUT /dmon/v1/overlord/nodes/roles" o input: JSON structure containing node names (as registered in the previous call) and their roles (e.g., storm, spark, etc.) o purpose: assign a role for DMon to know what to monitor ● application ID of the application to be deployed: o PUT /dmon/v1/overlord/application/**DEPLOYMENT_ID** **DICE Delivery Tool - DICE Configuration Optimization**: the CO is a command line tool, thus the DICE Delivery Tool needs to prepare the input TOSCA blueprint, the input data and experiment

configuration for the CO in the appropriate folders and execute the CO using its command line utility. After the CO's run is finished, CI retrieves the resulting files: data, configuration, updated TOSCA document.

DICE Configuration Optimization - DICE Delivery Tool: The CI uses RESTful calls to interact with the DICE Delivery Tool. It needs to a) submit a TOSCA blueprint to initiate deployment (POST /containers/**CONTAINER_ID**/blueprint), b) monitor the status of the deployment (GET /containers/**CONTAINER_ID**), and c) obtain the unique topology ID of the deployed topology (GET /containers/**CONTAINER_ID**/blueprint).

DICER - DICE Delivery Tool: A user can request a direct deployment from the DDSM. To enable this, DICER uses the DICE Delivery Tool's RESTful API to submit a TOSCA blueprint for deployment (POST /containers/**CONTAINER_ID**/blueprint).

DICE Delivery Tool - DICER: DICE Delivery Tool can accept a DDSM as an input. Internally, it will use DICER's CLI capability to transform the DDSM into a TOSCA document:

$ java -jar "**$DICER_TOOL_PATH**/dicer-core-0.1.0.jar" \
-inModel "**IN_XMI_FILE**" \
-outModel "**OUT_TOSCA_FILE**"

DICE Delivery Tool - Quality Testing Tool: Quality Testing Tool will need to provide an API in terms of a CLI and a format of the configuration to be able to describe the details of the experiment. The experiment itself will run on a modified Storm application, which is a part of the tool's invocation. In return, the QT needs to provide a report on the performance of the application. The report needs to be a flat JSON document listing the measured metrics and their established values.

DICE Delivery Tool - Enhancement Tool/Anomaly Detection: DICE Delivery Tool's IDE plug-in can provide a look-up of the deployments started from the IDE. This can let the users directly associate monitoring data in DMon with a specific deployment, which in turn represents an application's version.

Functional Description:

DICE Delivery Tool's Deployment Service provides:

- Authentication:
  - RESTful:
    - POST /auth/get-token
    - input: JSON structure containing credentials (username, password)
    - output: JSON structure containing authentication token
    - purpose: obtain an authentication token, which is mandatory with all the subsequent RESTful calls
  - CLI:
    - $ dice-deployment-cli authorize **MY-USERNAME Password**
- Submission of a blueprint to be deployed in a logical deployment container with ID CONTAINER_ID. The payload FILE can be a TOSCA YAML file (bare blueprint) or a .tar.gz bundle with TOSCA YAML and supplemental resource files (rich blueprint). Upcoming support also for an .xmi DDSM in both a bare blueprint or a rich blueprint presentation. The return message contains the **DEPLOYMENT_ID**, which is equivalent to the application id in DMon:
  - RESTful:
    - POST /containers/**CONTAINER_ID**/blueprint
    - input: contents of **FILE**
    - output: a JSON structure describing properties of the logical deployment container, the newly created blueprint (deployment), which itself contains the assigned **DEPLOYMENT_ID**
    - purpose: submit and deploy the blueprint
  - CLI:
    - $ dice-deployment-cli deploy **CONTAINER_ID FILE**
- Status of the deployment and outputs:
  - CLI:
    - $ dice-deployment-cli deploy **CONTAINER_ID FILE**
- Outputs of the deployment:RESTful:
  - GET /containers/**CONTAINER_ID**
  - output: a JSON structure describing properties of the logical deployment container, the blueprint (deployment) within the container, the status of the deployment, outputs from the deployment
  - purpose: query the properties and the status of the logical deployment container

	● o RESTful: ▪ GET /containers/**CONTAINER_ID** o CLI: ▪ $ dice-deployment-cli status **CONTAINER_ID** ▪ output: **OUTPUTS** **DICE Delivery Tool's Configuration Optimization provides**: ● Execution of local or remote scripts and applications. ● Storing results of application executions for each distinct run of the application. ● Serving the results from a permanent, well-defined URL. **DICE Delivery Tool as a client requires from other servers**: ● DIA's deployment specification as a TOSCA blueprint. Optionally, it needs any supplemental files and resources required by the TOSCA blueprint, but in this case all the files need to be supplied as a .tar.gz bundle. ● A RESTful interface for registering application and nodes to be monitored. The client can provide simple JSON structures such as flat lists and flat dictionaries to announce existence of particular nodes and their details. ● A report on a DIA's performance as a flat JSON dictionary of metric_name: value entries. ● A capability to transform a DDSM file into a TOSCA blueprint file.
	Integration Testing Scenario: We provide integration scenarios using the gherkin language, which can be used in a Cucumber executable specifications tool. **DICE Delivery Tool - DICE Monitoring tool**: Given the DICE Deployment Service is available at its address And the DMon Service is available at its address When I query DMon Observer for '_type: storm-topology' Then the DMon query result should be empty When I submit the blueprint 'storm-monitored.yaml' to DICE Deployment Service Then I should receive a deployment ID And the deployment should succeed after no more than 90 minutes When I query DMon Observer for '_type: storm-topology' Then the DMon query result should NOT be empty When I query DMon Observer for records with deployment ID as application ID Then the DMon query result should NOT be empty **DICE Delivery Tool - DICE Configuration Optimization**: Given the DICE Deployment Service is available at its address And the DICE Continuous Integration is available at its address And the DICE CI has a configured project 'WikiStats-Configuration-Optimization' When I request to build 'WikiStats-Configuration-Optimization' Then the build should succeed When I download the updated TOSCA blueprint Then the new TOSCA blueprint should be valid And the new TOSCA blueprint should be different from the original one **DICE Configuration Optimization - DICE Delivery Tool**: Given the DICE Deployment Service is available at its address And the DICE CO is configured to optimize blueprint 'storm-monitored.yaml' And the blueprint 'storm-monitored.yaml' defines output 'topology_id' When I submit the blueprint 'storm-monitored.yaml' to DICE Deployment Service Then the deployment should succeed after no more than 90 minutes And I should be able to obtain non-empty 'topology_id' deployment output **DICER - DICE Delivery Tool**: Given the DICE Deployment Service is available at its address And I have a DDSM in the file 'storm.xmi' When I use the DICER to submit the 'storm.xmi' to the DICE Deployment Service Then the deployment should succeed after no more than 90 minutes **DICE Delivery Tool - DICER**: Given the DICE Deployment Service is available at its address

	And I have a DDSM in the file 'storm.xmi' When I submit the DDSM 'storm.xmi' to DICE Deployment Service Then the deployment should succeed after no more than 90 minutes **DICE Delivery Tool - Quality Testing Tool**: Given the DICE CI has a configured project 'WikiStats-Quality-Testing' When I request to build 'WikiStats-Quality-Testing' Then the build should succeed And the ci page 'WikiStats-Quality-Testing/lastBuild/build-metrics/' build should contain valid metrics **DICE Delivery Tool - Enhancement Tool/Anomaly Detection**: Same as the integration scenario for DICE Delivery Tool - DICE Monitoring tool.
	Next Steps: ● M25: integration with Enhancement and Anomaly Detection tools. ● M26: integration with the Quality Testing tool ● M27: integration with DICER ● M28: integration with Fault Injection Tool

b) Type 2: IDE Based Integration

Table 6: IDE Based Integration

Tool	Responsible	Description
Deployment Design	PMI	**Current Status of Integration with the IDE:** The DICER front-end is successfully integrated within the DICE IDE. DICER invocation is possible through run-configuration menu entries and the front-end itself is able to send through to the DICE deployment and delivery service the produced TOSCA blueprint. The DICER back-end, however, requires its own deployment infrastructure and is currently supported as part of the same infrastructure area devoted to the delivery tool within our FCO infrastructure provider. **Next Steps:** IDE integration will be further investigated to understand if and how can the DICER back-end be integrated further into the Eclipse DICER IDE baseline.
Simulation Plugin	ZAR	**Current Status of Integration with the IDE:** The Simulation tool is composed of Eclipse plugins and therefore it is fully integrated in the DICE IDE. **Next Steps:** No next steps planned regarding the integration of the Simulation tool in the IDE
Optimization Plugin	PMI	**Current Status of Integration with the IDE:** The optimization tool initial version is integrated within the IDE as an external tool deployed on a Web server. **Next Steps:** The new version of the optimization tool Eclipse plugin will be integrated within the IDE at M26 (as an intermediate version) and at M30 for the final release. After M30, we will continue the optimization tool maintenance and evolution according to the feedback we will receive from the case study owners.
Verification Plugin	PMI	**Current Status of Integration with the IDE**: The front-end of D-verT is an Eclipse plugin that is available in the DICE dashboard. It interoperates with the DICE IDE as the user activates a verification task on a given UML model that has been loaded through its Papyrus file. In particular, the plugin is able to operate the translation of the UML diagrams representing Storm topologies into a JSON file by traversing their XMI descriptors. The user interaction with the D-verT user interface is basic as the user only edits few fields, providing the following informations: The name of the verification task to be run on the server; The XMI descriptor of the the UML diagram defining the application to be verified; The verification engine to use and a parameter that is required to set the verification instance; The nodes of the Storm topology to verify.

		Next Steps: No further integration activities are needed for the integration of D-verT with the IDE. The next efforts will involve the development of the functionalities related to Spark applications. The interaction with the user and with the DICE IDE will follow the same workflow adopted for Storm applications. We will release a new version of the DICE Verification Tool plugin with support for Spark and other minor improvements to the already implemented functionalities.
Monitoring Platform	IEAT	**Current Status of Integration with the IDE:** DICE Monitoring Platform is integrated in the DICE IDE as an external service, by opening a web view from the Eclipse. The DICE Monitoring Platform plug-in in Eclipse provider's end-users with access to the platform's controller service REST API, the administration interface, and to the visualization engine. The default end-points for DICE Monitoring Service can be configured from Eclipse's Preferences window. The DICE Monitoring Service Administration interface and DICE Monitoring Service Visualization UI are available as menu items in DICE Tools menu. **Next Steps:** As DICE Monitoring Platform's main purpose is to be deliver monitoring data to other DICE tools via REST API calls, hence the current level of integration in the IDE suffices. Note that the end-users of the platform have a Kibana UI to get insights out of collected data.
Anomaly Detection	IEAT	**Current Status of Integration with the IDE:** Currently we are working towards integration of the Anomaly Detection Tool in the IDE. At this point we have designed the layout of the form that has to be filled in by users in order to properly configure the Anomaly Detection Tool from Eclipse. The configuration form is defined as an eclipse plugin that enables users to select either pre-defined values for each of the options as described on the tool wiki page or to introduce custom values where possible. In the current implementation the form allows users to introduce values for a subset of the supported options. **Next Steps:** Implement the actual form to support predefined values used by the ADT and add the start functionality. All the integration work should be done by M30.
Trace Checking	PMI	**Current Status of Integration with the IDE:** The front-end of Dice-traCT will be similar to the one of D-verT (verification tool). It will be developed as an Eclipse plugin and it will be available in the DICE dashboard. The front-end component interoperates with the DICE IDE as the user activates trace checking by specifying the UML model that can be loaded through its Papyrus file and the temporal property to be considered in the log analysis. **Next Steps**: The integration with DICE IDE will be completed by April 2017 (M28).
Enhancement Tool	IMP	**Current Status of Integration with the IDE:** We provide integration status of DICE FG and DICE APR of DICE Enhancement tool respectively. o **DICE FG**: the current DICE FG tool can be executed through the command line. To run the DICE FG, user firstly needs to define a XML format configuration file, which specifies the general FG configuration, the input data, and a specific estimation or fitting algorithm for analysis. The runtime data can be stored in JSON file and retrieved by querying the DICE Monitoring tool. Then, user can go to the DICE FG directory to run DICE FG via the command line. o **DICE APR**: as an intermediate step for later anti-patterns detection & refactoring, we have developed a novel Model-to-model (UML to LQN) transformation module, which relies on the parameterized UML model with performance and reliability information annotated by DICE FG. This module can be invoked through the DICE IDE run-configuration panel. The user then has to interact with configuration panel to (1) specify the location of the source model (i.e., UML model) and the target model (i.e., LQN model); (2) include meta models for UML, LQN and Trace; (3) identify the location

		transformation scripts (e.g., *.eol, *.etl). The transformed LQN model can be viewed as an XML format file in the DICE IDE. **Next Steps:** The DICE Enhancement tool is planned for official release by M30 with the final version of the report. We plan to use command line tool to expose DICE FG functionalities, and it will be integrated transparently to the final user. The plugin will be available on the DICE repository to expose DICE APR functionalities. **DICE FG:** o **M28:** We plan to finalize the integration test of DICE-FG with the DICE toolchain in terms of DICE profile model acquisition and automated annotation. o **M30:** we plan to extend the features of DICE-FG towards supporting the modelling of technologies supported by DICE but that have been investigated to a limited extent in DICE-FG tool. For example, technologies such as Spark, Storm or Kafka where the computation proceeds according to a direct-acyclic graph (DAG) **DICE APR:** o **M25:** we plan to finalize our transformation module and test it on a selected UML model with DICE profile (DPIM and DTSM layers) for Big-Data scenario, e.g, Storm. The previous Model example might be extended for the further testing. o **M27:** we plan to identify the typical anti-patterns (1 or 2) of the Big-Data scenario, e.g. Storm, and formally specify the anti-patterns problem and solution for the selected anti-patterns. o **M28:** we plan to provide initial refactoring decisions (e.g. replacement, reassignment) to achieve the architecture refactoring manually or automatically according to the results of the anti-patterns detection. o **M30:** we plan to finalize the integration with the DICE FG and DICE IDE. This will let DICE APR retrieve the UML model annotated by DICE FG and provide refactoring decisions to DICE IDE.
Quality Testing	IMP	**Current Status of Integration with the IDE:** At present DICE CO is not integrated with the IDE, but it appears technically feasible to add the DICE QT jar to the next DICE IDE release. **Next Steps:** M26: IMP will work with PRO to finalize the integration of the DICE QT JAR within the DICE IDE.
Configuration Optimization	IMP	**Current Status of Integration with the IDE:** At present DICE CO is not integrated with the IDE, as it is a batch tool for overnight execution of a chain of experiments and therefore there is no strong need for an interactive GUI-based interface. IMP is however exploring the possibility to offer a simple plugin to simplify the instantiation of Jenkins rules to start/stop the execution of the tool automatically. **Next Steps:** DICE CO is currently integrated at the inter-tool integration level. The IMP unit has now started developing an Eclipse plugin to allow the end-user to instantiate the Jenkins rules automatically.
Fault Injection	FLEXI	Fault injection tool is not planned to be integrated within the IDE. Tool owners decided to keep this tool independent and keep outside the IDE and in an independent way.
Delivery Tool	XLAB	**Current Status of Integration with the IDE:** The current release (version 0.1.2) of the DICE Delivery tool's IDE integration works by opening a web view from the Eclipse into the DICE Deployment Service's Web graphical user interface. The plug-in also enables configuring the address of the DICE Deployment Service and the logical delivery container to show from the web view in the IDE. The user then has to interact with the service as if it was in a web user interface. For example, to submit a blueprint for deployment, the user has to click on the Upload button in the web view and then select in the provided system's Open dialog the file to be uploaded. No API for other tools' IDE plug-ins is available.

		As an intermediate step for a better integration with the user's working process, we have created a prototype external build tool, which relies on a DICE deployment tool's configuration file located in the DIA's project, specifying details about the DICE deployment service, credentials, and paths to any resource files that need to be bundled with the blueprint. This enables that the user can submit a specified blueprint in a bundle to the deployment. The progress and the deployment's results can be viewed in the Eclipse's standard output console. **Next Steps:** o **M24**: We will release the next version of the DICE Delivery Tool's IDE plug-in. This plug-in will replace the web-view-based interface with a plug-in that is fully integrated into Eclipse. The preference section of the plug-in will enable specifying multiple named DICE Deployment Service instances, extending the usability of the plug-in to enable handling multiple test-beds and, possibly, multiple platforms. The interface will have a dedicated panel with live updates of the statuses of past and momentary deployments. The deployment list will contain information about VCS commit that was relevant for each deployment. In this way, the users will be able to associate a version with a deployment, and use this information when running Enhancement tools. o **M27**: If required by other tools' IDE plug-ins, we will provide an API for our IDE plug-in. This API will let other tools query the status and history of deployments, e.g., for populating a drop-down menu where a user can select an application tag relevant in querying the DMon.

Appendix A. Updated Tools Summary

- **IDE:** Minor changes. The version of the Eclipse the IDE is based on has been updated. Also all the components included in the IDE are updated too. A new plug-in for methodology has been developed, including multiple Cheat Sheets for the user. The integrated tools were updated too.
- **Simulation:** Minor changes. The initial tool was architected for fully integration in the DICE IDE. Development updates during 20117r have extended the functionality of the tool but they have not modified its initial design for IDE integration. None of the added functionalities require components external to the IDE but they are integrated in it.
- **Optimization:** Major Changes. The tool has been split in two parts. An eclipse plug-in is the front end and interact with the simulation plug-in for running M2M transformations facilities. The optimization is then performed by a back-end web service which act as an engine and runs multiple simulations in parallel relying on JMT and GreatSPN.
- **Verification:**. Minor changes. Some clarifications are provided in the document mainly related to the internal architecture of the verification tool (D-VerT) and to the interactions of its components with the DICE environment.
- **Monitoring:** Minot changes. The Monitoring platform was architected for external integration in the DICE IDE. During the last year the tool has suffered a number of improvements including new features and additional supported technologies.
- **Enhancement:** Minor changes. DICE-FG is no longer an independent tool. It is one of modules of Enhancement tool. The DICE-FG will not generate any analysis report, instead, the UML model will be parameterized according to the analysis results and it can be inspected via the DICE Eclipse IDE GUI. Therefore, there is no report at this moment.
- **Trace checking:** No changes
- **Anomaly detection:** No changes
- **Delivery tool:** Major changes.
    - The delivery tools sequence diagrams now express the ability of a user to use an IDE or a command line tool directly with the deployment service. In the sequence diagrams we also explicitly show that the delivery tool returns a unique deployment ID, which is relevant for various enhancement tools, because it relates the monitoring data with a specific deployment.
    - The Application deployment use case now shows a general sequence that could either be initiated from an IDE from a Continuous Integration, or any other tool that depends on delivery tools (e.g., the Configuration Optimization). We remove the explicit use of Repository here, because in the Y2 implementation it is no longer a mandatory mediator of the deployments.
    - In the Continuous Integration diagram, we abstracted the operation of the Quality Testing Tool to a single actor to increase clarity. The description also better describes the process, clearly mentioning the use of logical deployment containers in the deployment service.
    - In the Obtaining configuration recommendation, we switched the role of storing optimal configuration and (past) data from Repository to the Delivery Tool. This is because the Continuous Integration part of the Delivery Tool is better suited for storing binary side products of tool runtimes during and after the lengthy CI job execution processes.
- **Configuration optimization:** Major changes. The main integration pattern is based on installation of the CO tool as a batch Jenkins job that periodically performs the configuration tuning. This is more effective than a IDE interactive interface since the CO activity can last several hours, therefore it is batch by nature. This change has been fostered during discussions with external stakeholders.
- **Quality testing:** Major changes. In the initial plans QT was meant to be a stand-alone load-injection tool. It has been however increasingly realized that this vision is suited for certain kinds of applications with standard architectures, such as n-tier websites, but it is less appropriate for Big data applications, such as streaming-oriented systems based on Apache Spark. In these systems the nature

of the data pushed to the topology varies with the application, and the architecture itself depends on how the application was programmed. This has therefore been reflected in the design of QT, which is now defined as a Java API for embedding automated stress testing as part of the application itself. The end user can now inject stress-testing bolts in the application and collect the results of the tests performed with his/her own custom stream data.

- **Fault injection:** No changes
- **Deployment Design (DICER):** New Addition

Deployment Design	PR2.11 PR2.17 PR2.18	PR2.11: DICER upholds separation of concerns by providing facilities to distinguish specific deployment design activities with respect to regular UML architectural design and refinement. PR2.17: DICER upholds ad-hoc transformation focus for DICE since it fully automates essential phases that are needed to provide support to DevOps-based infrastructure-as-code generation and analysis. PR2.18: DICER addresses the need for deployment transformations in a straightforward fashion and using also its own easy-to-use GUI for users to put together a deployment design in a matter of minutes - DICER support however is still limited to DICER's own deployment design format and will be further extended to encompass DICE UML DDSM designs as well.	PR2.11 : 100% PR2.17 : 100% PR2.18 : 75%

Appendix B. Tools Interfaces

Table 7: Tools Interfaces

	Name	Responsible	Available Interfaces (e.g. command line interface, RESTful API, Java method invocation, etc.)	For what each Interface is used for?(e.g. management of platform parameters, management of logical deployment containers, management of deployments)
1	IDE	PRO	Java method invocation	For creating UML models and launching the Tools
2	DICE Profile	ZAR	Java method invocation	Papyrus framework calls these methods to offer the user the possibility to use the set of stereotypes in the profiles.
3	Deployment Design	PMI	Eclipse Launch Configuration and command line interface	Both are used to invoke TOSCA blueprint production from input model
4	Simulation Plugin	ZAR	Java method invocation	Eclipse Framework invokes the simulation tool entry method when its execution is selected in the IDE. Optimization tool invokes the methods of the M2M and M2T transformations
5	Optimization Plugin	PMI	Java method invocation (front end), RESTful API (back end)	Eclipse Framework invokes the optimization front end when it is selected in the IDE. Front end runs back end engine to perform optimization and multiple simulations (by JMT or GreatSPN) in parallel
6	Verification Plugin	PMI	RESTful API (client-server) and command line interface	Execution of verification for user specified models
7	Monitoring Platform	IEAT	RESTful API (client-server)	Setting up and querying monitoring data.
8	Anomaly Detection	IEAT	Command line interface	Detecting anomalies, training predictive models
9	Trace Checking	PMI	RESTful API (client-server) and command line interface	execution of trace checking for user specified models
10	Enhancement Tool	IMP	Command line interface	Estimating and fitting application parameter related to memory and execution times and annotating UML models; refactoring the model if anti-patterns are detected.
11	Quality Testing	IMP	Java method invocation	Linking of automated stress-testing code to the application. This will be executed automatically once the application is deployed.
12	Configuration Optimization	IMP	Command line interface	Start and stop of automated experimentation to self-configure the system.
13	Fault Injection	FLEXI	Command line interface	Execution of faults within VMs
14	Delivery Tool	XLAB	RESTful API, command line interface	Administration of platform properties, management of logical deployment containers, management of deployments, management of CI jobs

Appendix C. Interactions Matrix

Table 8: Interaction MATRIX

Client \ Server	IDE	DICE Profile	Deployment Design	Simulation Plugin	Optimization Plugin	Verification Plugin	Monitoring Platform	Anomaly Detection	Trace Checking	Enhancement Tool	Quality Testing	Configuration Optimization	Fault Injection	Delivery Tool
IDE		/	/	/	/	/	/	/	/	/	/	/	/	RESTful API
DICE Profile	Eclipse plugin		Use	/	/	/	/	/	/	/	/	/	/	/
Deployment Design		Use		/	/	/	/	/	/	/	/	/	/	RESTful API
Simulation Plugin	Eclipse plugin	Use	/		Eclipse plugin	/	/	/	/	/	/	/	/	/
Optimization Plugin	Eclipse plugin	Use	/	/		/	/	/	/	/	/	/	/	/
Verification Plugin	Eclipse plugin	Use	/	/	/		/	/	/	/	/	/	/	/
Monitoring Platform	Web plugin	/		/	/	/		REST API calls	REST API calls	REST API calls	REST API calls	REST API calls	/	REST API calls
Anomaly Detection	Eclipse plugin	/	/	/	/	/	/		/	/	/	/	/	/
Trace Checking	Eclipse plugin	Use	/	/	/	/	/	/		/	/	/	/	RESTful API
Enhancement Tool	/	/	/	/	/	/	/	/	/		/	/	/	RESTful API
Quality Testing	/	/	/	/	/	/	RESTful API	/	/	/		/	/	/
Configuration Optimization	/	/	/	/	/	/	RESTful API	/	/	/	/		/	CLI
Fault Injection	/	/	/	/	/	/	/	/	/	/	/	/		/
Delivery Tool	Eclipse plugin	/	RESTful API	/	/	/	RESTful API	/	/	/	CLI	CLI	CLI	