Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

DICE

Architecture definition and integration plan - Initial version

Deliverable 1.3

Deliverable:	D1.3
Title:	Architecture definition and integration plan – Initial version
Editor(s):	Ilias Spais (ATC)
Contributor(s):	Giuliano Casale (IMP), Tatiana Ustinova (IMP), Pooyan Jamshidi (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), Madalina Erascu (IEAT), Daniel Pop (IEAT), Gabriel Iuhasz (IEAT), Youssef Ridene (NETF), Josuah Aron (NETF), Craig Sheridan (FLEXI), Darren Whigham (FLEXI)
Reviewers:	Youssef Ridene (NETF), Michele Ciavotta (PMI)
Type (R/P/DEC):	Report
Version:	1.0
Date:	31-Jan-2016
Status:	Final 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

There is a need to define an architecture that will describe the DICE platform and how its tools and components will be integrated. Determining the architectural solution through which the DICE framework will be developed is a problem that must be addressed systematically.
We describe the decision making process that we followed to conclude to the appropriate DICE architecture. We present the architecture styles that we have examined, the trade-off analysis that we have performed and the justification of our decision against the goals and premises of DICE. We also provide an overview of the DICE tools and outline how they are positioned in the DICE methodology. Their interactions are described and sequence diagrams and data flows are also provided.
With the adoption of the plugin architectural style (offered by Eclipse[1]), the DICE IDE (Integrated Development Environment, described in D1.2. "Requirements Specification") provides a methodological workflow, specifying business and technical actors, processes and unit steps needed for designing a data-intensive application. The DICE methodology can be followed through the IDE and allows the user to cover both the design and the pre-production phase of a data-intensive applications.

[1] http://www.eclipse.org/

Table of contents

List of figures

List of tables

1. Overview

After performing a systematic analysis, we have concluded that DICE Technological Tool-Chain should be implemented following a plugin architectural style. We have conducted an architectural trade-offs and integration pattern analysis and presented it in detail in what follows. A summary view of the project architecture is shown in Figure 1:

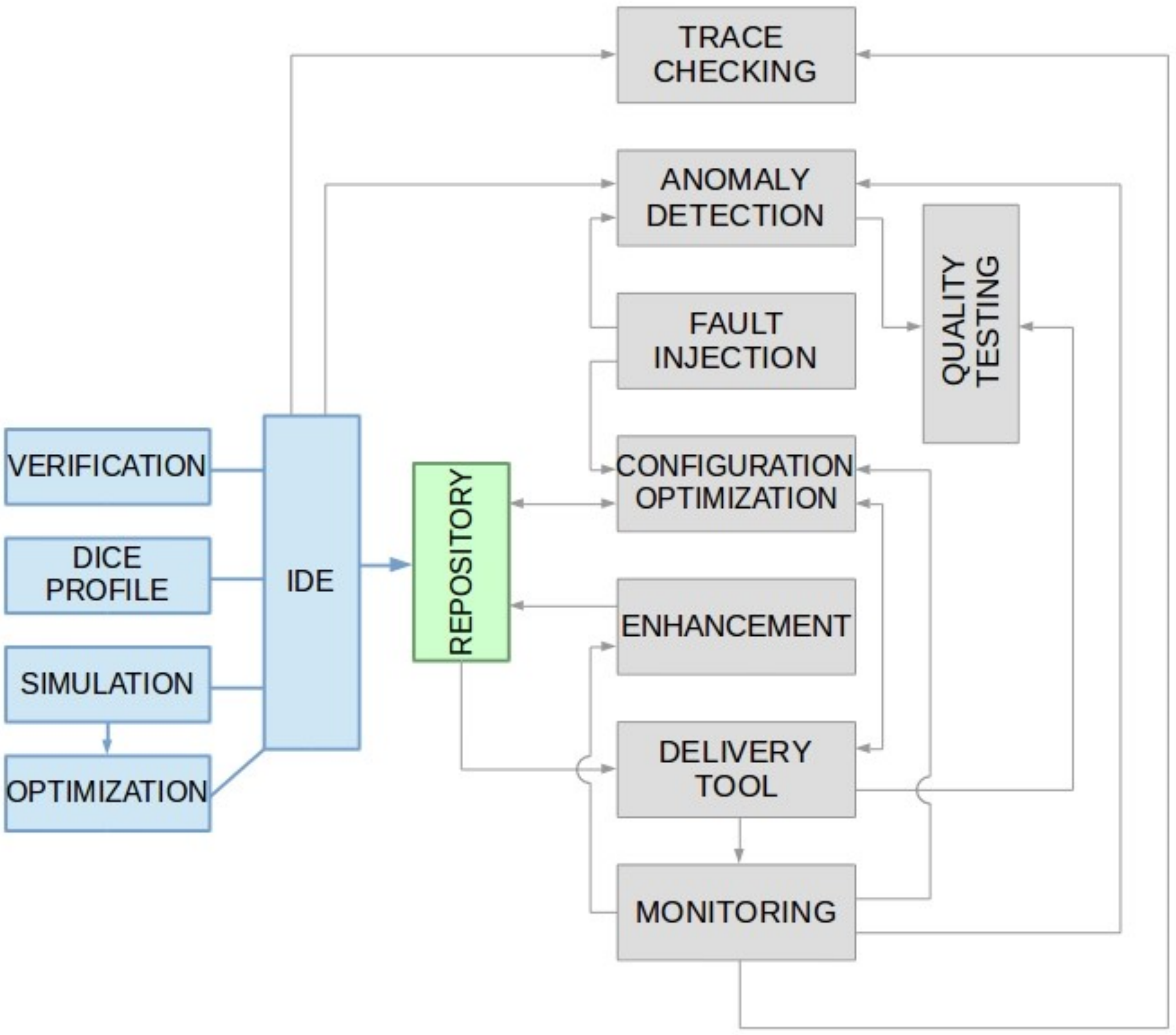


Figure 1. DICE architecture.

For the compilation of this report, we initially did a thorough pass over the DICE project requirements in order to provide a summary of the business and technical requirements as well as the DevOps practices and technologies used. In sequence, we define an architecture overview of the DICE platform and after that we identify DICE integration patterns, focusing on the Eclipse plugin architecture.

An architectural trade-off analysis is presented in the DICE Architecture and Integration section (Section 3). We compare the MicroServices architectural approach [10] versus the plugin architectural style. They are being evaluated with the use of the Architecture Trade-off Analysis Method (ATAM). A description of the DICE tools in relation to the DICE architecture is given in the following section (Section 2. We give a short summary of each tool operation. Finally, the Appendix contains each DICE tools description of interactions, sequence diagrams and data flows.

1.1. Requirements

This section presents an overview of the DICE project requirements. We provide a summary of the business and technical requirements as well as DevOps practices and technologies used.

In general, the DICE project aims at delivering a methodology and a tool chain to help independent software vendors to develop data-intensive applications. From a business requirements perspective, the necessary functional requirements are: to develop a UML profile, a consistent methodology, and the underpinning model-to-model transformations to support model driven engineering approaches (MDE) for Big Data applications. Furthermore, DICE aims at translating such high-level design models into a concrete TOSCA-compliant deployment plan and execute it by means of a model-driven deployment and configuration tool chain.

We can identify several non-functional business requirements. The UML designs will include annotations with performance, reliability, privacy as well as other specific data-intensive applications (DIA) requirements. Then, tools will be developed to predict the fulfilment of these requirements before and during application development. **Security** and **privacy** requirements should be modelled for well-known cloud providers (considering data management policy and encryption among other things) to support the provider selection process. Another non-functional requirement that should be met is **scalability (horizontal and vertical)**. The system should manage the complexity of large software and data-intensive systems. It should be as much as possible linearly scalable to deal with a variable workloads. Auto scaling should be also supported.

DICE envisions the co-existence of multiple simulation, verification, and testing tools that can guide the developer through the quality assessment of early prototypes of the Big Data application. The **testability** requirement implies that the best architecture alternatives according to the workload managed should be identified. For example, deploy and run the application on an isolated simulation environment with historical data to verify that quality tests pass. Analysis of the application architecture using various data sources and computational logic is another example. Testing and load stressing scenarios should be easily configurable.

The **quality** requirement of the project is satisfied in several modules. Quality metrics (response time, throughput etc.) will be automatically extracted to be improved on following versions. The user monitors the performance of the system and evaluates the impact of the data rate in order to re-configure auto scaling policy and desired performance rates. Design anti-patterns and root-causes of quality anomalies will also be detected. Specifying SLA requirements can be mentioned as a related example of non-functional properties.

Fault tolerance is another non-functional requirement that is achieved. It is met in operational system monitoring (e.g. data and logs to detect candidate anomalies), measuring (e.g. provide the input/output operations per second (IOPS)) and management (user reports generation) operations. The simulation tool provides insights on hardware **deployment** requirements (e.g. how much RAM, CPU, etc. will be required to get a certain performance level). The **documentation** requirement is met by the methodology blueprint requirement (graphical representation within the main tooling suite that can be used as a tutorial for the end user).

Regarding the **technical requirements**, in the context of WP1, most of them relate to the task of developing the DICE IDE as front-end to the DICE tool-chain and releasing the integrated framework. In particular, they are the support of stereotyping of UML diagrams with DICE profile, the creation of a dashboard tool that will guide the user through the workflow with code generation capabilities, the invocation of continuous integration tools through the IDE, the launch of annotated UML models (that can trigger simulation and verification tools), the load of verification properties and the graphical representation of the verification tools output, the visualization of the analysis outcomes to the user and finally the loading of safety and privacy properties from the model of the application with the anomaly trace tool.

In the context of WP2, the technical requirements are related to developing systematic methods for the design as well as representational support models to aid the Model-Driven Engineering of DIAs. DICE follows a paradigm inspired to the Model-Driven Architecture of UML, see deliverable D1.2, section D.2 for a review and introduction to core definitions such as DPIM, DTSM, and DDSM. Users require from the DICE profile to support the incremental specification of Data-Intensive Applications following a Model-Driven Engineering approach, to stem every abstraction layer from UML, to allow definition of values of constraints (e.g. maximum cost for the DIA), properties (e.g. outgoing flow from a Storage Node) and stereotype attributes (batch and speed DIA elements) using the UML Profile MARTE (Modeling and Analysis for Real-Time and Embedded Systems; see D1.1) VSL (Value Specification Modeling) standard, to define structural and behavioral constraints typical in targeted technologies (e.g. Apache Hadoop, Apache Storm, Apache Spark, etc.) and to use packages to separately tackle the description of targeted technologies in the respective profile abstraction layers (e.g. DTSM and DDSM). Furthermore, the DPIM must be generic enough so as not to require any specialization, e.g., for domain-specific DIAs, the DTSM layer must support the definition of technology-specific DIA topologies, the DTSM must include extension facilities, the DDSM layer must support the definition of an Actionable deployment view (TOSCA-ready) and the DICE IDE must support the development of DIA exploiting the DICE profile and following the DICE methodology. Finally, the DICE profile and its design shall work under the assumption that their focus of application is limited to providing facilities and methodological approaches to support those properties that are relevant to perform analysis (e.g. for fine-tuning, load-estimation, etc.), testing (e.g. for run-time verification and adaptation towards continuous integration), monitoring (e.g. for flexible continuous improvement, etc.).

In the context of WP3, the technical requirements are related to assessing quality requirements and at offering an optimized deployment configuration for the application under development. For accomplishing its objectives, WP3 will develop transformation tools, simulation tools, verification tools and optimization tools. They require that the transformation tools perform a model-to-model transformation taking the input from a DPIM or DTSM DICE annotated UML model and returning a formal model (e.g. Petri net model or a temporal logic model), that they take into account the relevant annotations in the DICE profile (properties, constraints and metrics) and transform them into the corresponding artifact in the formal model. The verification tools require that they are able from the UML DICE to model a system, to show possible execution traces of the system with its corresponding time stamps. Regarding the optimization tools, their objective is the minimization of deployment costs fulfilling at the same time reliability and performance constraints (e.g., map-reduce jobs execution deadlines). They should also explore the design space and accept the specification of a timeout and return results when this timeout is expired. The transformation tools and simulation tools should have no difference between white box and black box model elements. All tools should permit the user to check their outputs against SLAs included in the UML model annotations.

In the context of WP4, the technical requirements are related to tools and techniques to support the iterative improvement of quality characteristics in data-intensive applications through feedback. This WP will design and implement the monitoring platform that will collect and store traces and logs produced during the execution of data-intensive applications. It will create monitoring tools, anomaly trace tools and enhancement tools. They require from the monitoring tools to perform monitoring data pre-processing (extraction) before storing the data in the data warehouse in order to facilitate usage by other tasks, to support interactive visualization of monitoring data and to provide the data warehouse the ability to prevent unauthorized access to the monitoring data. Once correlation between anomalies in runtime and anti-patterns has been detected, the enhancement tools should propose methods for refactoring the design. The tools should also be able to compare two versions of the application to identify relevant changes and extract or infer the input parameters needed by the simulation tools and optimization tools to perform the quality analyses. Monitoring data must support the reconstruction of a sequence of events and the identification of the time when things occurred (for example a consistent timestamp in a distributed system). The monitoring tools and enhancement tools should capture the growth in the data size for the application. The enhancement tools must be capable of automatically updating UML models with analysis results. The anomaly trace tools must allow the developer to choose and load the safety and privacy properties from the model of the application described through the DICE profile and to be able to check, given a trace of the events of interest of the application, whether that trace is compatible with the desired safety and privacy properties. Finally, there must be a way to link the information that is stored in the data warehouse with the features and concepts of the DICE UML models (operations, attributes, objects, etc.).

In the context of WP5, the technical requirements are related to developing tools which help put the DICE tools users' application to the actual environment and evaluate its runtime. The DICE technical team will develop continuous integration and deployment tools executing TOSCA, testing tools and a testbed environment (Flexiant Cloud Orchestrator). WP5 technical requirements are that the continuous integration tools must record the results of each test, mapping them to the version number and offer a dashboard to consolidate the view of the application deployment to restricted users. The quality testing tools must test the application for efficiency and reliability, safety and provide independent test results. The deployment tools must be able to run automatically and autonomically, to deploy and install any application and the related monitoring tools from a valid topology of the supported DICE building blocks, to be extendible and support multiple IaaS and to support selected PaaS.

Technologies and tools

Regarding the tools used, Papyrus [1] is chosen as UML modeler. ECore EAnnotation [2] will be used to annotate papyrus UML models in order to extend metamodel properties. Based on their experience with MOSKitt CASE tool [3], PRO proposes to create an IDE based on the last Eclipse Framework version.. Jenkins [4] will be invoked through a provided Eclipse plugin to allow continuous integration. Eclipse plugins and wizards will be created for the custom tools developed. Eclipse Modelling Facilities (EMF) [5] will provide basic meta-model consistency validation techniques. The monitoring platform will use Elastic Search [6], Logstash and Kibana [7] on Flexiant Cloud Orchestrator.

DevOps practices

A fundamental assumption of the DICE project is that the models co-existing in our MDA methodology (UML & TOSCA models) will act as a vehicular language to integrate different tools across the DICE tool chain. It will also allow different actors to see a complex Big Data systems at different level of abstractions, such as abstract level, architecture level, and deployment level. Therefore, the DICE approach is going to be fully compliant with the Model-Driven philosophy, and should be deemed as a possible extension of MDA to the realm of Big Data.

Following the OMG guidelines, the DICE profile and methodology supports the incremental specification of DIAs following a Model-Driven Architecture approach. It mimics the standard assumptions behind Model-Driven Engineering, including the separation of concerns across three disjoint but related layers (Platform-Independent and Platform-Specific). Separation of concerns is one of the basic principles behind model-driven engineering and related technologies. The DICE Profile must use packages to separately tackle the description of targeted technologies in the respective profile abstraction layers (e.g. DTSM and DDSM).

Several notations are being considered in the scope of DICE (e.g. MDA, MDE, MARTE, SecureML). These notations already provide diagramming facilities that may be assumed as directly related to the needs and requirements of the DICE profile. For example, following the MDA paradigm, ModaCloudML[2] offers modeling facilities to reason on cloud-based applications from multiple, functionally-complete perspectives. An example of following DevOps practices [12] is the interactive design component that will be offered to allow the graphical representation of the workflow. The DICE IDE will guide the developer through the DICE methodology. This interactive component will promote communication and collaboration between development, QA and IT operations, as DevOps assumes.
The MDE approach underpins the necessity to bridge the gap from Dev and Ops by proposing to use UML models as a way to share a global view of the system. Such global view of the system is a key element of the DevOps vision. DICE therefore wants to emphasize the convergence of MDE and DevOps as a way to achieve an integrated, harmonized system view and orchestration between Dev and Ops.

1.1.1. Architectural overview

The DICE architecture offers a comprehensive set of tools that cover both the design and the pre-production phase of a data-intensive application development. A diagram summarizing the overall DICE architecture is given in Figure 1. The different colors distinguish the two main components of the architecture:

- **Development tools,** which are primarily centered on the development stage of the data-intensive application. The IDE implements the DICE quality-driven methodology that guides the developer through the different stage of refinement of design models up to implementing the initial prototypes of its application. The IDE supports multiple quality analyses offered by the verification, simulation and optimization tools.
- **Runtime tools,** which collect data during the application quality testing to characterize the efficiency, reliability and correctness of the components. This data is consolidated in the DICE monitoring platform and used to semi-automatically detect anomalies, optimize the system configuration, and enhance the design.

The purpose of the tools within each group of tools is explained in details in the following sections.

1.1.2. Development tools

The central element of the DICE architecture is the **Integrated Development Environment (IDE)**, where the developer specifies the data-intensive application using a model-driven engineering approach. To support this activity, the DICE Eclipse-based IDE embeds the **DICE UML profile** which provides the stereotypes and tags needed for the specification of data-intensive applications in UML.

[2] http://www.modaclouds.eu/wp-content/uploads/2012/09/MODAClouds_D4.2.1_MODACloudMLDevelopmentInitialVersion.pdf

Following an MDE approach, models are defined by the user in a top-down fashion, stemming from platform-independent specifications of components and architecture (**DPIM UML models**), through assignment of specific technologies to implement such specifications (**DTSM UML models**), and finally to the mapping of the application components into a concrete TOSCA-compliant deployment specification (**DDSM UML models**). Such models can be related by DICE model-to-model transformations that are automatically executed within the IDE, to reduce the amount of manual work required from the user. For example, initial DTSM and DDSM models can be generated from the DPIM models.

Throughout the application design, the DICE IDE offers the possibility to automatically translate certain DICE models into *formal models* for assessment of quality properties (efficiency, costs, safety/correctness, etc.). Each analysis requires to run dedicated tools that reside outside the IDE environment, in order to obtain prediction metrics. The **simulation**, **optimization** and **verification plugins** take care of translating models in-between IDE and these external tools. They also collect via REST APIs the outputs of these tools that are shown to the user inside the IDE. The interface of these plugins assumes the user to be unskilled in the usage of the formal models. Furthermore, the quality properties are defined in terms of constraints using appropriate language constructs.

As the developer progressively refines the application model and the application code, s/he is going to periodically commit them to a **repository, i.e., a version control system (vcs)**. In DICE we will ensure that every commit increases the **version number** of the application, which is a unique identifier used across tools to keep synchronized models and code. The repository acts as a shared source of models and code across different versions for all the DICE tools that need to access them. The repository will reside externally to the IDE and will be accessed through appropriate tools (e.g. SVN, GIT, etc.).

1.2. Runtime tools

After the initial prototyping of the application, the developer will request to deploy the current prototype. After an automatic commit of all models and code to the external repository, the **continuous deployment tool** will retrieve a copy of both of them from the repository, build the application, and internally store the outputs and their associated artifacts. The delivery tool will then initialize the deployment environment (if not already created), consisting of VMs and software stack, and deploy (or update the existing deployment of) the application. The delivery operation also connects the DICE **monitoring platform** to the deployed application. The monitoring platform will be started and stopped by REST APIs and will acquire a pre-defined set of metrics that will be continuously stored in a performance data repository.

The **anomaly detection** and **trace checking tools** will also feature an IDE plugin and will be able to query the monitoring platform for relevant metrics and use them to generate analyses concerning anomaly in performance, reliability or operational behavior of the application at a given release version. The anomaly detection tool will reason on the base of black-box and machine-learning models constructed from the monitoring data. Conversely, the trace checking tools are going to analyze the correctness of traces. These analyses will be manually invoked by the user from the IDE plugin. Similar to these, the **enhancement tool** will automatically annotate the DICE models stored in the repository with statistics on the inferred and recorded monitoring data, in order to help the user to inspect the root-causes of performance or reliability anomalies.

The **quality testing tools** will support the generation of test workloads to the application. Such workloads are those that will be used to assess the quality of prototypes. Similarly, the **fault injection tool** will generate faults and malicious interferences that can help verifying the application resilience. Both tools integrate a heterogeneous set of actuators and can be run manually by an operator through a command-line interface and configuration files. They will also be exploited by the **configuration optimization** tool to generate an experimental plan automatically given a time budget. The output of this tool is to confirm the optimal configuration of the deployment for an application in its final stages before being pushed to production. Compared to the optimization plugin, configuration optimization will also deal with fine-grained systems parameters (e.g. buffer size, block size, JVM configuration, etc.), which are difficult to model in design-time exploration. Moreover, configuration optimization is black-box and solely measurement-driven, whereas design space exploration is primarily model-driven.

1.3. WP-level architecture

The following table summarizes the WP-level responsibilities of the different components of the architecture, the lead maintainer and the major contributors:

Table 1: DICE tools and work packages

Tool	Work Package	Lead Maintainer	Major Contributors
IDE	1	PRO	
DICE Profile	2	PMI	ZAR, NETF
Simulation Plugin	3	ZAR	IMP
Optimization Plugin	3	PMI	
Verification Plugin	3	PMI	IEAT
Monitoring Platform	4	IEAT	
Anomaly Detection	4	IEAT	IMP
Trace Checking	4	PMI	
Enhancement Tool	4	IMP	
Quality Testing	5	IMP	
Configuration Optimization	5	IMP	IEAT
Fault Injection	5	FLEXI	
Repository	5	XLAB	
Delivery Tools	5	XLAB	

2. DICE tools

2.1. Overview of DICE tools

In the previous section, we have provided a high-level description of the DICE tools, explaining their main role in the DICE architecture. Here, we describe in more details the technical characteristics of each DICE tool. We begin by providing a more detailed description of the intended purpose of each tool of the DICE architecture. We cover in particular two dimensions: **motivation** and **technical innovation**.

Table 2: DICE 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 DevOps where a designer can create models to describe data-intensive applications and their underpinning technology stack. In particular, a core innovation is the fact of being a complete IDE for going from design to development.
DICE Profile	Existing UML models do not offer stereotypes and tags 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.
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, etc.) 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 simulation tool to support the automated search of optimal solution. This is needed to limit the time needed to complete the search and obtain a good solution.	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 behavior 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 results in loss of data (e.g. due to timeouts, buffer overflows, etc.).	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.

	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.	
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 behavior 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).
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. For example, reading a threading level at a data base does not explain what business-level operation was performed by the DB.	The enhancement tool will introduce a new methodology and prototype to close the gap between measurements and UML diagrams. No mature methodology appears available in the research literature that can address this inverse problem of going from measurements back to the 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.	There is a shortage of tools to guide the experimental configuration of complex software systems. This will provide an innovative solution in this space, which will combine experimentation with reasoning based on machine learning models.
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.	Some fault injection tools exist on the market, such as ChaosMonkey[3]. These tools however are either platform specific or limited in functionality such as with ChaosMonkey for AWS (Amazon …) and termination of VMs (Virtual Machines). 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.	N/A. 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 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.	DICE will offer a novel continuous deployment solution that combines some cutting-edge solutions for cloud computing, namely the emerging TOSCA profile and the Cloudify deployment tool, naturally extending the model-driven development into realization of the model in the target environment, providing a complete design-deployment-testing ecosystem in DICE. These tools will be empowered with blueprints to support the deployment of the data-intensive technologies supported by DICE.

2.2. Positioning tools in the methodology

The methodology and DICE tools are conceived with a high desire to reduce the time to market of business-critical data-intensive applications (DIA). Therefore, the DICE IDE provides a *methodological workflow*, specifying business and technical actors, processes and unit steps needed for designing a data-intensive application. This general-purpose methodology allows the design of data-intensive cloud applications for different domains.

[3] http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html

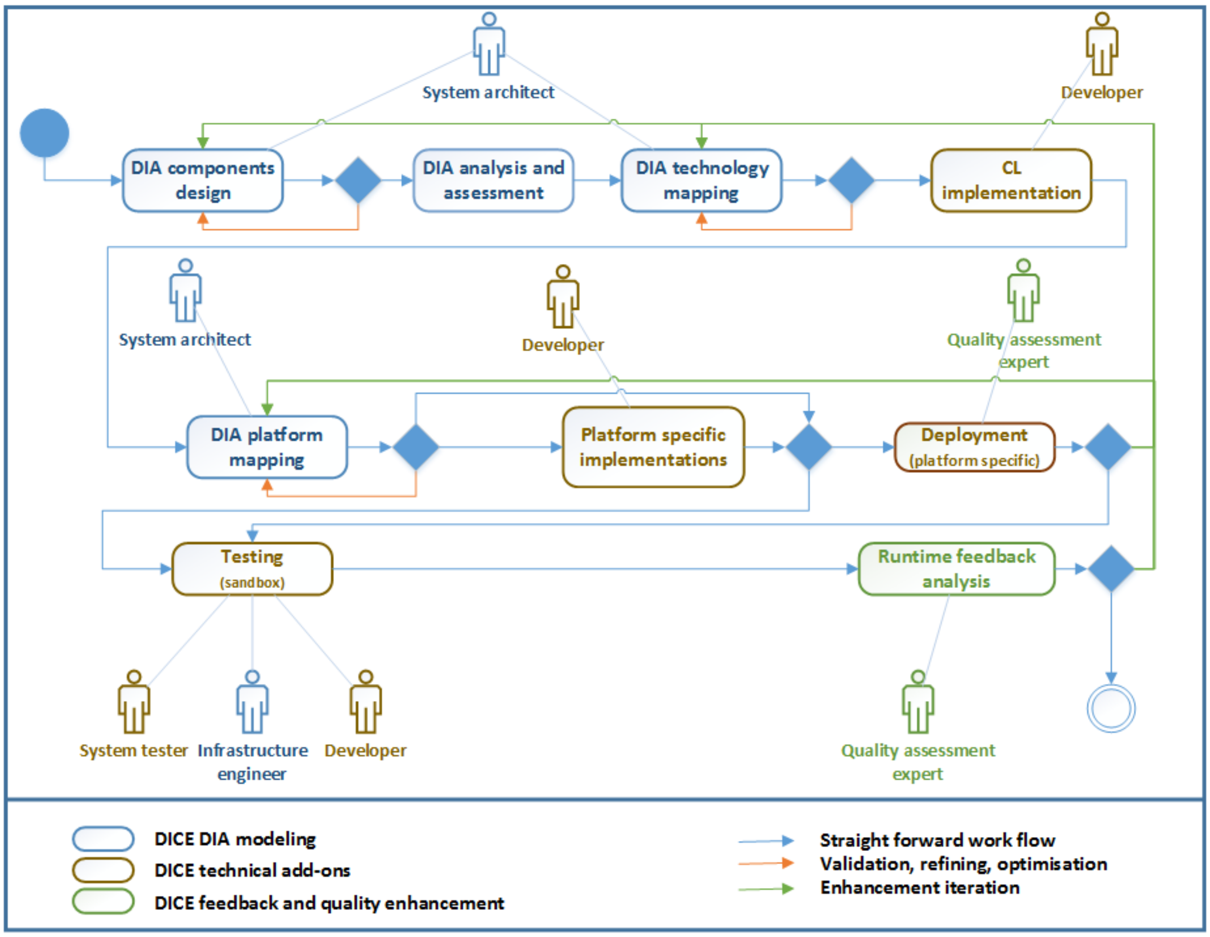

System architect
Developer
DIA components design
DIA analysis and assessment
DIA technology mapping
CL implementation
System architect
Developer
Quality assessment expert
DIA platform mapping
Platform specific implementations
Deployment (platform specific)
Testing (sandbox)
Runtime feedback analysis
System tester
Infrastructure engineer
Developer
Quality assessment expert
DICE DIA modeling
DICE technical add-ons
DICE feedback and quality enhancement
Straight forward work flow
Validation, refining, optimisation
Enhancement iteration

Figure 2. DICE Methodology

Methodology steps	DICE Tools Used
DIA Component design	DICE Profile
DIA Analysis and Assessment	Verification Tool Enhancement Tool
DIA Technology Mapping	DICE Profile
CL Implementation	IDE (Natively built-in technological components)
DIA Platform Mapping	DICE Profile
Platform Specific Implementation	IDE (Natively built-in technological components)
Deployment	Delivery Tool (CI, Deployment repository…)
Testing	Quality Testing Tools: • Anomaly Detection • Fault Injection • Configuration/Optimization
Runtime feedback analysis	Monitoring Tool

3. Architecture and Integration

This section elaborates on the possible architectural solutions through which the DICE framework may be developed. In order to procure the reasonable starting points of our decision-making we enacted a series of brainstorming sessions to evaluate: (a) the tenets and challenges behind the DICE H2020 project; (b) the current market concerns and stakeholders envisioned by DICE; (c) the trends in current architecture styles consistent with DICE goals.

In addition, the section features an application of the industrial-strength method architecture decision making method called the "Architecture Trade-off Analysis Method" [8] to make a decision and establish its value against the goals and premises of DICE.

Finally, this section elaborates on how, given the architecture choices elaborated in the following section, the resulting DICE models will be shared across the DICE technological solution.

3.1. Architectural Tradeoff Analysis and Integration Patterns

As previously specified, in order to come up with a decision concerning the DICE technological architecture we proceeded as follows.

First, in a series of brainstorming sessions and focus groups we established a number of valuable architectural styles alternatives (e.g. a plugin style vs. a service-oriented style, etc.). The result of this very first investigation yielded a number of styles rotating around services (e.g. typical SOA web-services or more modern and DevOps consistent Microservices [11]) as well as a number of more design-level styles (e.g. the plugin architecture style).

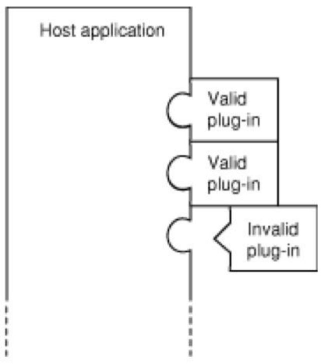
Second, in a preliminary decision evaluation session, we narrowed the architecture decision down to two alternatives, namely, a plugin architecture style (e.g. think of the Eclipse IDE) against a Microservice architecture style (e.g. think of modern web-service transaction systems such as Netflix).

Third, through an application of the ATAM method, we evaluated the best-possible option from our two choices. The rest of this subsection focuses on elaborating further details on the two architectural choices that we faced at the decision-making phase, i.e., the second step.

3.1.1. Plugin Architectural Style

In the plugin architecture style, software architects are constrained to develop their applications in terms of the following architectural elements:

1. Plugin: a bundle that adds functionality to an application, the host application;
2. Host application: offers the mechanisms to add new plugins during operation;
3. Extension-point (a.k.a. plugin interface): a stub in the host that can be extended by hostable plugins;



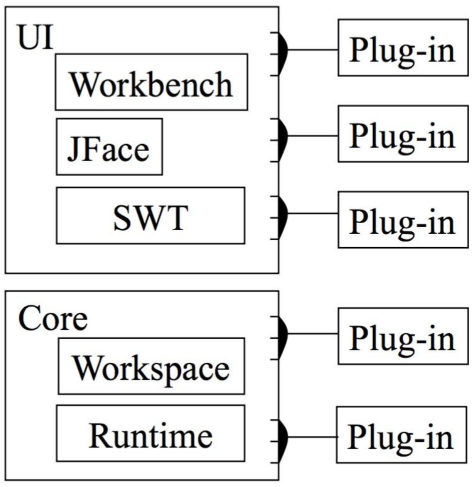
Software architects are constrained to adhere to a single restriction concerning this architectural style, namely, that Plugins have to comply to the extensibility constraints defined in architecture of the application hosting extendable plugins. For example, architects elaborating a plugin for the Eclipse IDE need to adhere strictly to restrictions superimposed by the nature, structure and limitations of the Eclipse IDE (e.g., memory limit, etc.).

In a nutshell, there are several advantages connected to using a plugin architecture style in the scope of DICE. For example, members of the DICE consortium (as well as future users and vendors connected to the DICE IDE) can implement and incorporate application features very quickly. Also, since plugins are separate modules with well- defined interfaces, you can quickly isolate and solve problems. Moreover, creating custom versions of DICE applications would become easier and without source code modifications. Furthermore, Eclipse marketplace[4] is an appropriate place that we can used to disseminate the DICE tools.

3.1.2. Implementation Consequences for a Plugin Styled DICE Solution

Several mechanisms would have to be defined in connection to a plugin styled DICE solution. More in particular:

1. Define a list of methods/functions a plugin must implement or define a base class that plugin must use;
2. Define mechanisms for registering callbacks;
3. Define what type of behavior each method or function must exhibit;



However, there are several frameworks that the DICE consortium may consider as a starting point behind styling the DICE solution with plugins. For example, the Eclipse IDE offers a valuable platform and a platform extension system that fits almost perfectly with the DICE tenets (e.g., Big-Data architectural design in a model-driven fashion) and challenges (e.g., supporting continuous architecting in a DevOps fashion [12]). Indeed, the Eclipse IDE (see the overall structure in the Figure on the right-hand side) would provide the DICE solution with:

1. An extensible platform/customizable IDE via the RCP framework;
2. Core services for controlling tools working together;
3. Runtime platform to support system development by composition of plugins - also, the runtime platform discovers plugins at startup and manages auto plugins loading;
4. Multi-layering featuring a platform specific layer, a java-development tools (JDT) layer and an overall IDE layer;

3.1.3. Microservices Architecture Style

The Microservice architecture [9][10][11] is a new architectural style that has been introduced after new paradigms like continuous delivery and DevOps. In this architectural style, applications can be composed of fully independent services that communicate with each other via light weight API such as REST. The micro-services are built around business capabilities and are typically independently deployable by fully automated deployment services.

The services in this architectural style can be implemented using different programming languages and technology stacks. This style as opposed to traditional hierarchical architectures (e.g. composite services), is more symmetric and follow the principles of publish and subscribe communication style. This architectural style is the most suitable style for integrating applications around business capabilities and in DevOps paradigm [12] where there the team structures are typically shaped around business capabilities.

3.1.4. Implementation Consequences for a Microservice Styled DICE Solution

In a classical plugin style, the DICE IDE consolidates plugins that do not talk much to each other and dialog in the IDE environment through the shared UML models. Each plugin calls the external tools via REST APIs. For example, testing and deployment tools are orchestrated by their respective plugins inside the IDE. The IDE Plugin X would therefore develop the logic to connect to external Tool X. The other plugins would be entirely agnostic of this. Each IDE plugin can be called by the user irrespectively of the other plugins, but requires a DICE m to be loaded in the

[4] http://marketplace.eclipse.org/

IDE. There is a clear unique selling point at the end of the project, the IDE with the powerful DICE plugins inside it, but methodology and workflow may be difficult to customize or adapt to specific needs of end users.

Conversely, Microservice are a cloud-native architecture [10] through which a software system can be realized as a set of small independent services [9]. Each of these services are capable of being deployed independently on a different platform and run in their own process while communicating through lightweight mechanisms like RESTFul APIs [10]. In the DICE setting, this means that each service is a DICE business capability that can use various programming languages and data stores [11].

Moreover, Microservices architecture is different from a canonical Service Oriented Architecture. SOA is an architectural pattern where the services are self-contained units that communicate with each-other via communication protocols. However, they have several shortcomings, as listed in the following table, they are stateful, synchronous and technology dependent (all based on enterprise service bus - ESB) and also the integration needs to be hardcoded into few available languages such as BPEL.

In addition, using microservices, we can develop DICE tools separately and the integration can be based on the assumptions that all tools needs to emit some messages and receive some events from yet another topic. So the integration becomes reactive, asynchronous and event driven. Finally, from a technological point of view, the integration connectors can be written in any language and hosted even as yet another service (e.g. Kubernetes (for integration) + Docker[5] (for tools)).

3.2. Architectural Tradeoff Analysis and Architecture Decisions for DICE

The goal of the Architecture Tradeoff Analysis Method (ATAM) is to determine how quality attributes interact in such a way that a best fitting architectural decision may become apparent. The steps for ATAM are the following:

1. Present method to stakeholders
2. Present business drivers (by project manager)
3. Present architecture (by lead architect)
4. Identify architectural approaches
5. Generate quality attribute tree
6. Elaborate architectural approaches
7. Brainstorm and prioritize scenarios
8. Analyse architectural approaches
9. Present results

In the following text we elaborate on the (architectural) quality attribute tree (step 5) wherefore DICE quality attributes are evaluated against architectural alternatives pruned from software architecture research and practice (see Section 3.1). The root node of the tree (see the figure below) is termed “utility”. It expresses the overall quality of the architecture. The next level contains the architectural quality attributes that were evaluated for a concrete option. These are again broken down into more detailed constituents. Finally, the leaf nodes are concrete scenarios where said DICE quality attributes are enacted.

Steps 1 through 3 as well as 6 through 9 of the procedure actually took place during several DICE online and plenary meetings and are not documented here (meeting minutes provide additional elaboration and details of said steps).

[5] https://www.docker.com/

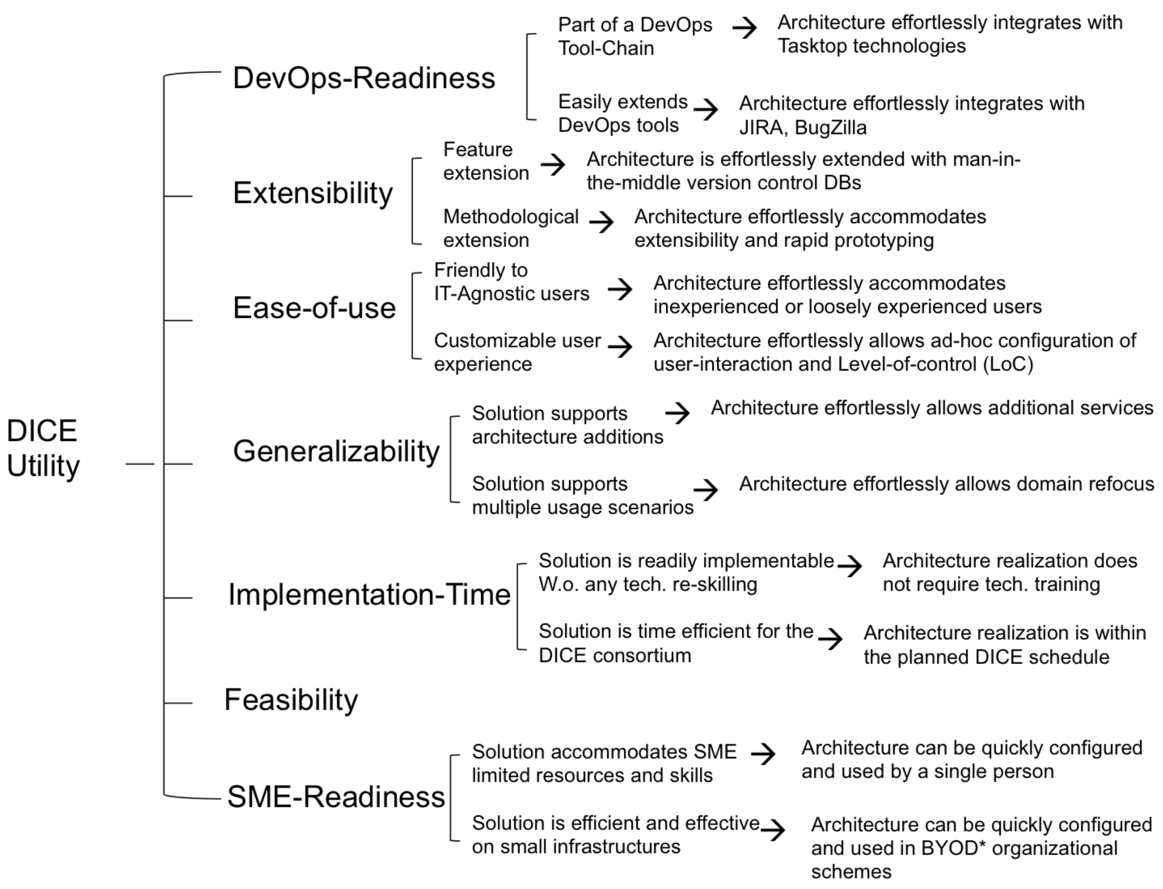

DICE Utility
DevOps-Readiness
Part of a DevOps Tool-Chain
Architecture effortlessly integrates with Tasktop technologies
Easily extends DevOps tools
Architecture effortlessly integrates with JIRA, BugZilla
Extensibility
Feature extension
Architecture is effortlessly extended with man-in-the-middle version control DBs
Methodological extension
Architecture effortlessly accommodates extensibility and rapid prototyping
Ease-of-use
Friendly to IT-Agnostic users
Architecture effortlessly accommodates inexperienced or loosely experienced users
Customizable user experience
Architecture effortlessly allows ad-hoc configuration of user-interaction and Level-of-control (LoC)
Generalizability
Solution supports architecture additions
Architecture effortlessly allows additional services
Solution supports multiple usage scenarios
Architecture effortlessly allows domain refocus
Implementation-Time
Solution is readily implementable W.o. any tech. re-skilling
Architecture realization does not require tech. training
Solution is time efficient for the DICE consortium
Architecture realization is within the planned DICE schedule
Feasibility
SME-Readiness
Solution accommodates SME limited resources and skills
Architecture can be quickly configured and used by a single person
Solution is efficient and effective on small infrastructures
Architecture can be quickly configured and used in BYOD* organizational schemes

Figure 3. DICE Quality Attribute Tree: General Form

The tree rotates around a set of seven key evaluation characteristics, namely: **DevOps-readiness** - our solution should accommodate and support wherever possible the DevOps strategy and ways of thinking; **extensibility** - our solution should allow full extensibility while supporting said activity with appropriate and adequate tooling; **Ease-of-use** - our solution should make no assumption as to the level of skill of the user and should be ready to support multiple possible users with multiple possible concerns and ability levels; **Generalizability** - our solution should make no assumption as to where and in which context should the DICE solution be used in practice, also the solution should be ready to accommodate rapid and unforeseen contextual changes; **Implementation-time** - our solution should not force additional skills on the DICE consortium and should accommodate our planned schedule and prototypal timing; **Feasibility** - our solution should be feasible in the allotted time given the desirable other DICE characteristics; **SME-readiness** - our solution should be ready to accommodate the organizational scenarios and variables typical in SMEs and should remain efficient in and on top of infrastructure size and expectations typical for SMEs as well.

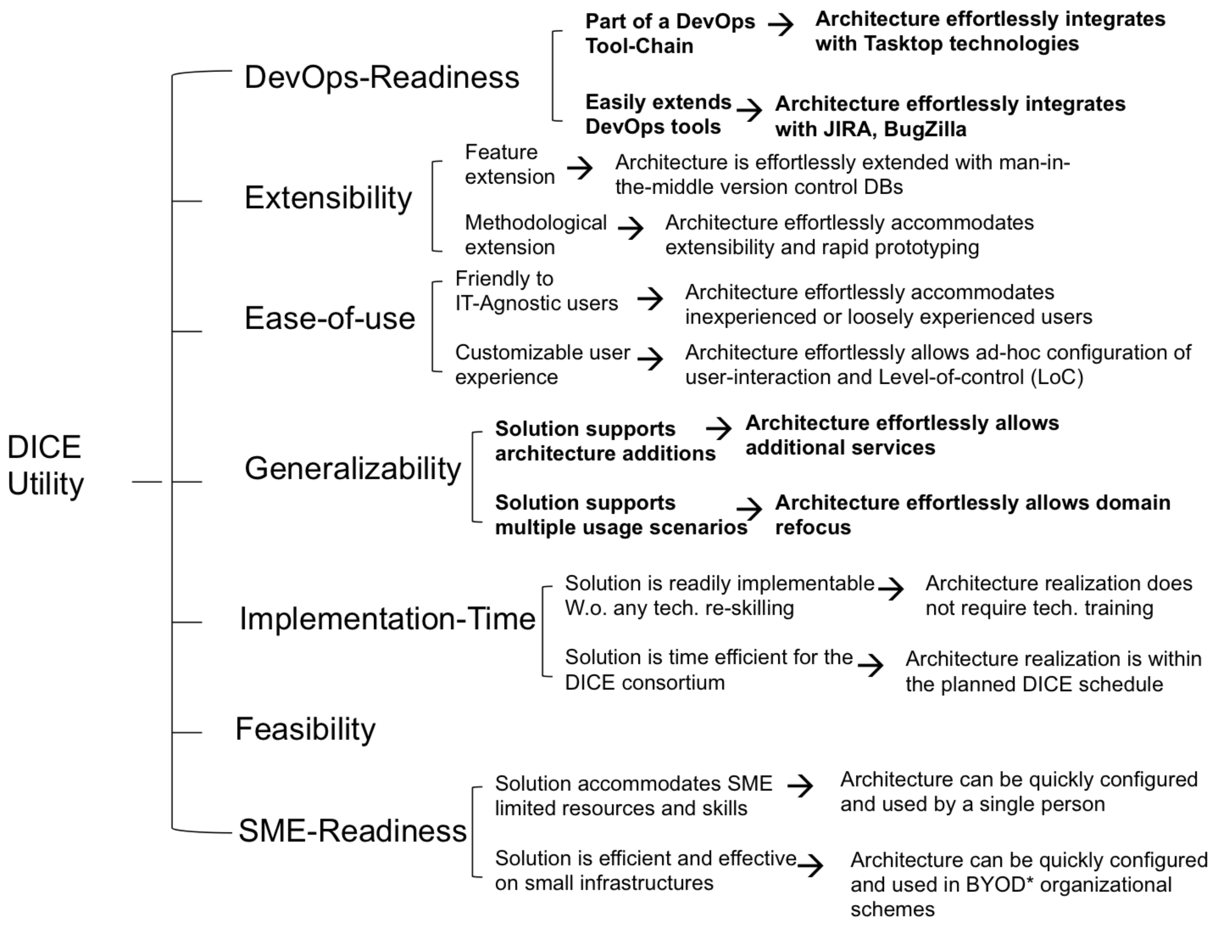

DICE Utility
DevOps-Readiness
Part of a DevOps Tool-Chain
Architecture effortlessly integrates with Tasktop technologies
Easily extends DevOps tools
Architecture effortlessly integrates with JIRA, BugZilla
Extensibility
Feature extension
Architecture is effortlessly extended with man-in-the-middle version control DBs
Methodological extension
Architecture effortlessly accommodates extensibility and rapid prototyping
Ease-of-use
Friendly to IT-Agnostic users
Architecture effortlessly accommodates inexperienced or loosely experienced users
Customizable user experience
Architecture effortlessly allows ad-hoc configuration of user-interaction and Level-of-control (LoC)
Generalizability
Solution supports architecture additions
Architecture effortlessly allows additional services
Solution supports multiple usage scenarios
Architecture effortlessly allows domain refocus
Implementation-Time
Solution is readily implementable W.o. any tech. re-skilling
Architecture realization does not require tech. training
Solution is time efficient for the DICE consortium
Architecture realization is within the planned DICE schedule
Feasibility
SME-Readiness
Solution accommodates SME limited resources and skills
Architecture can be quickly configured and used by a single person
Solution is efficient and effective on small infrastructures
Architecture can be quickly configured and used in BYOD* organizational schemes

Figure 4. DICE Quality Attribute Tree: Microservices[6]

A Microservices solution is almost explicitly tied to a DevOps way of working since it envisions reducing services to be made available to their smallest form possible in such a way so as to divide responsibility and reduce coordination where possible. This way of working however may not be feasible with the DICE tenets and challenges. Although extremely novel, this alternative would force re-training and refocusing of skills across the DICE consortium since many partners may not be familiar with technologies involved in Microservices (e.g. WS, messaging systems, REST, ESB, etc.). Also, the architecture may limit the methodological extensibility and user-experience connected to the DICE solution since this would inextricably be tied with a DevOps way of working, i.e., by means of Microservices. Finally, the solution may not accommodate well the small resources in SMEs or organizational schemes such as Bring-Your-Own-Device, very common in SMEs as well.

[6] Matched points are highlighted in bold.

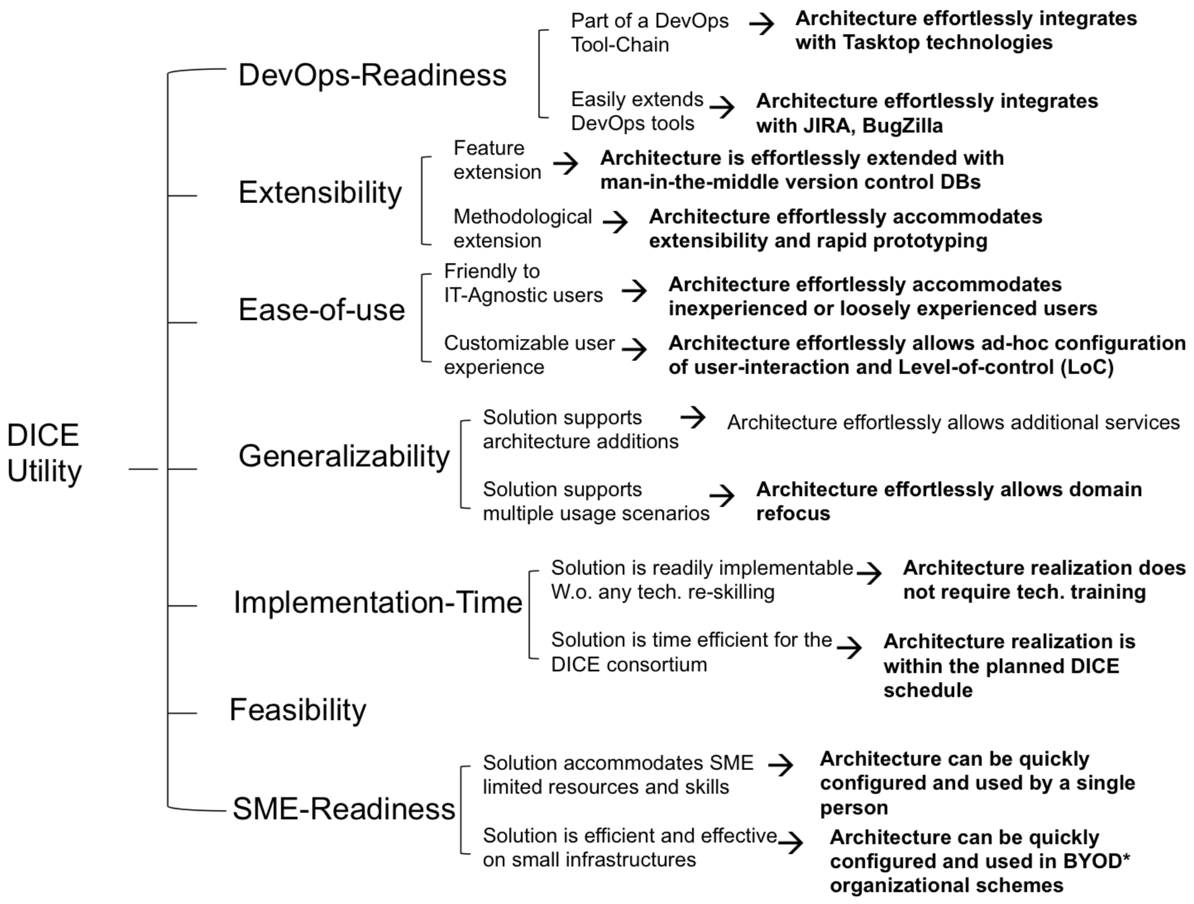

DICE Utility
DevOps-Readiness
Part of a DevOps Tool-Chain
Architecture effortlessly integrates with Tasktop technologies
Easily extends DevOps tools
Architecture effortlessly integrates with JIRA, BugZilla
Extensibility
Feature extension
Architecture is effortlessly extended with man-in-the-middle version control DBs
Methodological extension
Architecture effortlessly accommodates extensibility and rapid prototyping
Ease-of-use
Friendly to IT-Agnostic users
Architecture effortlessly accommodates inexperienced or loosely experienced users
Customizable user experience
Architecture effortlessly allows ad-hoc configuration of user-interaction and Level-of-control (LoC)
Generalizability
Solution supports architecture additions
Architecture effortlessly allows additional services
Solution supports multiple usage scenarios
Architecture effortlessly allows domain refocus
Implementation-Time
Solution is readily implementable W.o. any tech. re-skilling
Architecture realization does not require tech. training
Solution is time efficient for the DICE consortium
Architecture realization is within the planned DICE schedule
Feasibility
SME-Readiness
Solution accommodates SME limited resources and skills
Architecture can be quickly configured and used by a single person
Solution is efficient and effective on small infrastructures
Architecture can be quickly configured and used in BYOD* organizational schemes

Figure 5. DICE Quality Attribute Tree: Plugin Style

A more classical and established plugin style (e.g. as supported by the Eclipse IDE and related extension frameworks such as RCP[7]) may be configured to accommodate a DevOps way of working (e.g. as accompanied by Eclipse tools such as MyLyn). Also, rotating around well-established Eclipse technological extension frameworks, architecture extensibility and generalizability may be kept to the highest level. In addition, Eclipse and similar plugin based technologies are already very pragmatically versed into allowing configuration and preparation of ad-hoc distributions, depending on the level of experience or desired control by the user. This would allow the DICE solution to accommodate scenarios in which both neophytes and gurus may be interested in using our technology. Finally, Eclipse and similar plugin styled platforms already accommodate the resource limits and organizational practices typical in SMEs.

It is therefore our decision, that the DICE Technological Tool-Chain be implemented following a plugin architectural style, as supported by the technological solution which best accommodates previous design decisions within the DICE consortium, that is, the DICE IDE.

[7] https://wiki.eclipse.org/Rich_Client_Platform

3.3. Shared Models

Stemming from the above decision it is our intention to make available the following shared models:

DPIM	XMI 2.11	Standard EMF Format	DAM-Compatible UML-Papyrus Format
DTSM	XMI 2.11	Standard EMF Format	DAM-Compatible UML-Papyrus Format
DDSM	XMI 2.11	TOSCA V1	TOSCA-JSON, TOSCA-YAML

Integration Patterns
Prepare → ***Commit*** → ***Modify***: Models should be prepared in draft format, committed to version-control and modified (e.g. according to analyses or further modelling and design);
Retrieve → ***Analyse*** → ***Modify*** → ***Commit***: Before being analysed, updated versions of the models should always be retrieved from version-control, should always be analysed before a modification, i.e., the analysis is the rationale of the modification and should always be committed after modification (would be addressed in details in WP2 deliverables);
Observe → ***Retrieve*** → ***Modify*** → ***Commit:*** Models which need to undergo modifications as a consequence of observed monitoring evaluations are first retrieved from version-control, then modified and then re-committed again;

3.4. DICE architecture

A plugin architectural style has been chosen to implement the DICE technological tool chain. As examined in previous section, the plugin architecture offers several advantages such as extensibility, separation of concerns and many other best practices. The Eclipse IDE itself follows this architecture. It offers a platform extension system through plugins which fits with the DICE principles and challenges. The DICE architecture components are the development tools which are integrated in the IDE, the runtime tools which are called from the IDE, the IDE itself and the external repository. They are distinguished with different colours in Figure 1.
The DICE profile and methodology support the incremental specification of Data-Intensive Applications (DIAs) following a Model-Driven Engineering approach. The DICE IDE exploits the DICE profile and follows the DICE methodology. It offers wizards to guide the developer through the steps envisioned in the DICE methodology. The DICE tools described will be used at different stages of the methodology. In the following table we provide a summary of the architecture of each DICE component, a short behavioural description and the link to the respective section in the Appendix.

Tool	Architecture component	Behavioural description	Appendix
IDE	The Integrated Development Environment is the core component. The Plugin Architecture Style is implemented by customizing the Eclipse IDE.	The user starts the IDE. This latter will execute the requested actions and delegate the requests to the tool(s) that can serve the request.	A.1
DICE Profile	This component belongs to the development tools. It provides the stereotypes and tags needed for the specification of data-intensive applications in UML.	The user loads the DICE Profile model from the resources. He selects the desired elements to annotate. Once finished, he submit the model to the repository.	A.1
Simulation	Simulation is part of the development tools. It gives information about the predicted	The user starts a new simulation using the annotated UML models as input. The simulation process is configured through the	A.2

	metric values in the technological environment being studied.	IDE. The users executes the process and the results are shown to the user in the GUI.	
Optimization	Optimization is part of the development tools. While simulation offers the possibility to evaluate a given model, the optimization plugin will perform multiple invocations of the simulation tool to support the automated search of optimal solution.	The user loads the model from the repository. The optimization tool is invoked. Some performance metrics are returned to the tool and the simulation tool is invoked. A certain number of deployment models are sent to the simulation tool. The outcome is returned to the Optimization tool and reported to the user.	A.3
Verification	The verification component is part of the development tools. It offers a logic language to analyse the correctness of a system, generate counterexamples, and expose safety risks.	The user loads the model from the repository. He selects a property to be checked using templates and the tool from the analysis. The annotated DTSM model, the property and the tool are sent to the Verification plugin. The outcome of the verification and the trace (of the system that violates it) are reported to the user.	A.4
Monitoring Platform	The Monitoring platform component is part of the runtime tools. It is invoked from the IDE through RESTFul services. It monitors quality metrics in the application and in its underpinning software stack and infrastructure as the application runs.	The user loads the model from the repository. The user submits a query string to the monitoring platform through the IDE. He can specify parameters such as a time interval, the type of the output and others. An elastic search query is generated and the results are returned to the monitoring platform.	A.5
Anomaly Detection	The Anomaly detection component is part of the runtime tools. It attempts to decide if the application performance or reliability have been affected by a change based on the different version of the DICE application and models.	The user loads the model from the repository. The user will have to select a subset of features and timeframe on which anomaly detection will take place. The resulting data will be used to train and validate a predictive model. If a model has been already calculated for this subset of features, the service will check the given timeframe for anomalies. Once an anomaly is detected, a reaction (send email, notify users, etc.) will be triggered.	A.8
Trace Checking	The Trace Checking component is part of the runtime tools. It 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 user loads the model from the repository and activates trace checking. The list of logs is retrieved in the Monitoring platform and sent to the IDE. The user selects a property and a time window. The outcome of the trace-checking is reported to the IDE.	A.7
Enhancement Tool	The Enhancement tool component is part of the runtime tools. It supports the users with the task of evolving the application quality after tests on prototypes.	The user loads the model from the repository and requests to detect anti-patterns in the current design. The tool analyses the current UML models and returns an indication of possible anti-patterns to the IDE. The Enhancement tool queries monitoring data from the Monitoring Platform and uses these data to analyse the parameters of the performance models.	A.6
Quality	The Quality Testing component is	The user loads the model from the repository.	A.11

Testing	part of the runtime tools. It generates artificial workloads that are sequentially submitted to the running application according to a testing plan.	The main data entity in quality testing tool is the test plan which needs to be in a user readable XML like format and compatible with other similar tool such as JMeter. The test plan should define the notion of time unit and the level of the load that will be injected during time periods. After the test plan is defined by the user via IDE, it will be sent to the quality testing tool to run the test case.	
Configuration Optimization	The Configuration Optimization component is part of the runtime tools. It optimizes the big data application configuration within limited time using numerical optimization and machine learning models.	The user loads the model from the repository and launches the tool. The configuration template is retrieved by the tool through model repository. The appropriate configuration is then set in the template. In order to perform model fitting, tool requires to retrieve the performance data and augment new points in the repository. These performance data serve as the main ingredient for reasoning where to test next in the tool.	A.10
Fault Injection	The Fault Injection component is part of the runtime tools. It explores the application response to unexpected problems in the operational environments (e.g., faults, contention, etc.) by creating on-demand such problems.	The user starts the Fault Injection tool. He enters his input using command line options. The tool will connect to the virtual machine and begin Fault. The results are stored in an accessible log file and returned to the tool.	A.12
Repository	The repository component acts as a shared source of models and code across different versions for all the DICE tools that need to access them. It will reside externally to the IDE and will be accessed through appropriate tools (e.g. SVN).	The user loads the model from the repository. Upon completion of this work, the user submits the file (models/ configuration scripts etc.) to the repository.	A.9
Delivery Tool	The Delivery Tool component will initialize the deployment environment and deploy the application. It also connects the monitoring platform to the deployed application.	The user loads the model from the repository. The IDE tools trigger the model-to-text transformation, which produces an OASIS TOSCA document in YAML format. The Delivery Tool consumes the TOSCA document and, based on the blueprint description, deploys and configures the application in the test bed.	A.9

3.5. Integration plan

The following table presents DICE integration plan focusing on the most important activities and milestones

Table 3: DICE integration plan.

Date	DICE Framework Version	Included features
M12	-	Initial version of: • Simulation support from ZAR • Verification support from PMI • Monitoring support from IEAT

		• Delivery Tool support from XLAB
M18	1.0.0	Initial version of: • Optimization support from PMI • Anomaly detection support from IEAT • Trace Checking support from PMI • Enhancement support from IMP • Fault Injection support from FLEXI • Configuration Optimization support from IMP First release of the DICE Framework with Repository support from XLAB with all M12 initial version of the tools. This release will be available on the official GitHub repository of DICE: github.com/dice-project/ The IDE plugins will come with format of packaging and uploading to the market place. We plan to use Vagrant scripts for installing the monitoring framework in Virtual Box, which is useful for getting to know the framework. For the services related to the deployment, TOSCA blueprints will be released for Cloudify that enable bootstrapping and deploying these services with little effort.
M24	2.0.0	Initial version of: • Quality Testing support from IMP Intermediate versions of: • Simulation support from ZAR • Verification support from PMI • Delivery Tool support from XLAB Final version of: • Monitoring support from IEAT Second release of DICE Framework (Initial complete version) with all M12 and M18 initial version of the tools.
M30	3.0.0	Final version of: • Simulation support from ZAR • Verification support from PMI • Delivery Tool support from XLAB • Optimization support from PMI • Quality Testing support from IMP Final version of DICE Framework with all M24 and M30 final versions of the tools

This is a first approach of the integration plan. Every deliverable will include the related features as far as possible, depending on how much costs including it in this deliverable or not. If one feature is not included in one deliverable, will be present in the next one.

Appendix A.

A.1. IDE

A.1.1 Stereotyping a UML diagram with the DICE profile to obtain a Platform-Independent Model (PIM)

ID	UC1.1
Title	Stereotyping a UML diagram with the DICE profile to obtain a Platform-Indep. Model
Priority	Required
Actors	Architect, IDE
Flow events	A technical person capable of designing and modelling a data intensive application models the Platform-Indep. UML Model stereotyped with the DICE profile.
Pre-conditions	UML diagram of domain model
Post-conditions	Stereotyped diagram with DICE profile

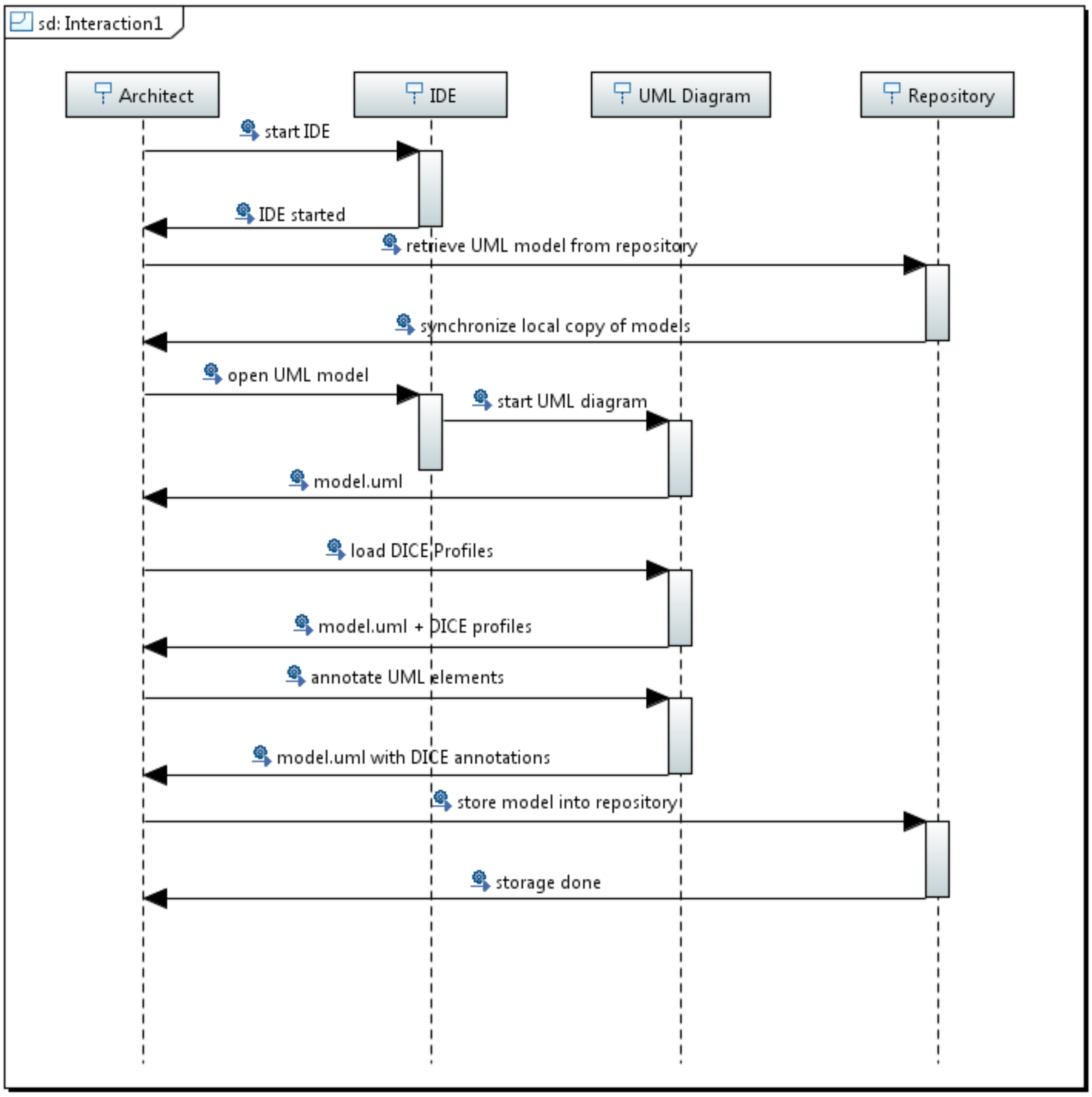
A.1.1.1. Description of interactions

The use case UC1.1 specifies that, from *an existing UML model, an architect should be able to annotate it using the DICE profile*.

To obtain such information, the following steps need to be performed:

- The Architect starts the IDE.
- The Architect open the desired model to annotate.
- The Architect loads the DICE profile model from the resources.
- The Architect selects the desired elements to annotate.
- The resulting model could be stored into a repository in order to use it in next steps.

A.1.1.2. Sequence diagrams



A.1.1.3. Data flows

The UML model should be retrieved from a repository, and the DICE profile model will be available as plugin in the IDE. The Architect will synchronize its local copy with the data in the repository. Then he need to load the DICE profile model from the IDE plugins, and start annotating the UML model. Once finished, the model should be uploaded to the repository again.

A.1.2 Analysis, simulation, verification, feedback, and transformations until obtaining a deployment model

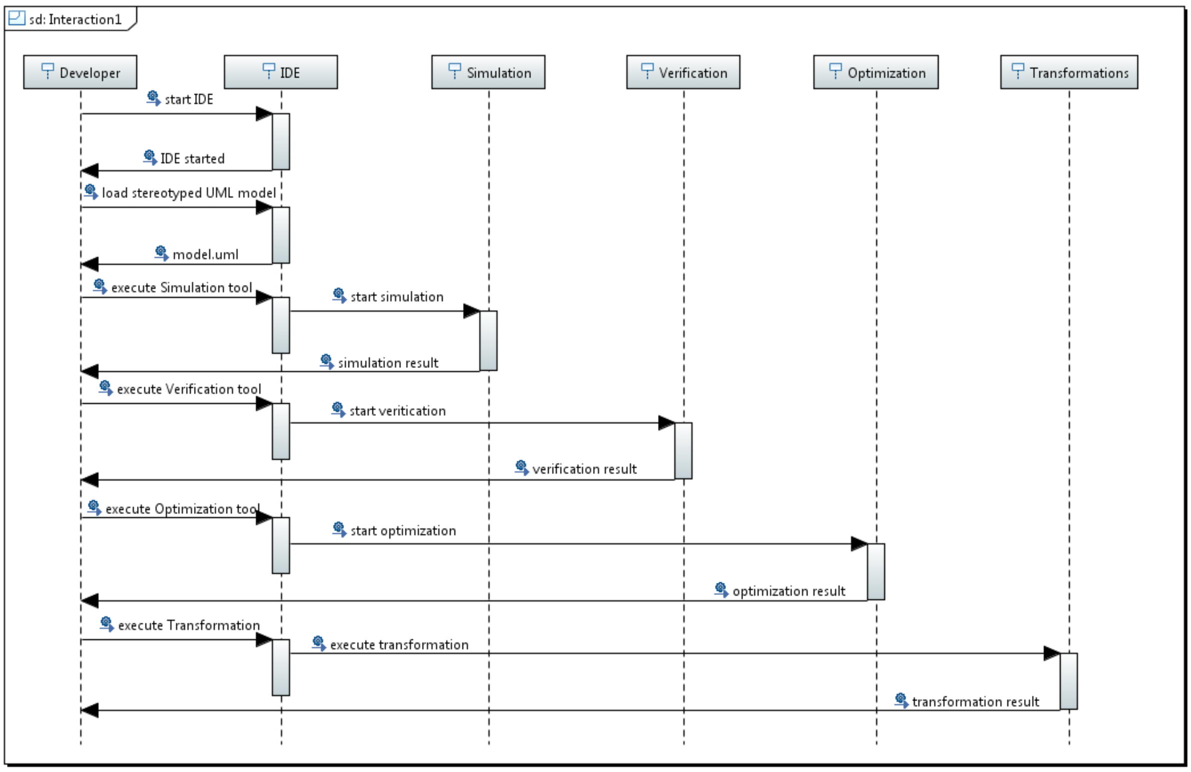
ID	UC1.2
Title	Analysis, simulation, verification, feedback, and transformations until obtaining a deployment model
Priority	Required
Actors	Developer, IDE, QA Tester
Flow events	The developer is a technical person capable of developing a data intensive application. He is guided through the DICE methodology to accelerate development and deployment of the data-intensive application with quality iteration. A Quality-Assessment expert may also run and examine the output of the QA testing tools in addition to the developer
Pre-conditions	Stereotyped diagram with DICE profile
Post-conditions	Architecture model, platform-specific model, QA models

A.1.2.1. Description of interactions

The use case UC1.1 specifies that, from *an existing stereotyped UML model, a developer should be able to execute certain operations on them*.
To obtain such information, the following steps need to be performed:

- The Architect starts the IDE.
- The Architect loads a stereotyped UML model.
- Via the contextual menu, developer will be able to start Verification, Simulation or Optimization tool over the model. Also s/he can perform transformations to other models.

A.1.2.2. Sequence diagrams



A.1.2.3. Data flows

As seen in the sequence diagram, the Developer always starts the request. The IDE will execute the requested action and delegate to the Tool the work. Finally, the tool will send the result to the repository.

A.2. Simulation tool

The requirements elicitation of D1.2 only considers a single use case[8] that concerns the Simulation Tools component, the UC3.1. This use case can be summarized as 2:

ID	UC3.1
Title	Verification of reliability or performance properties from a DPIM/DTSM DICE annotated UML model
Priority	Required
Actors	QA Engineer, IDE, Transformation Tools, Simulation Tools
Pre-conditions	There exists a DPIM/DTSM level UML annotated model
Post-conditions	The QA Engineer gets information about the predicted metric value in the technological environment being studied

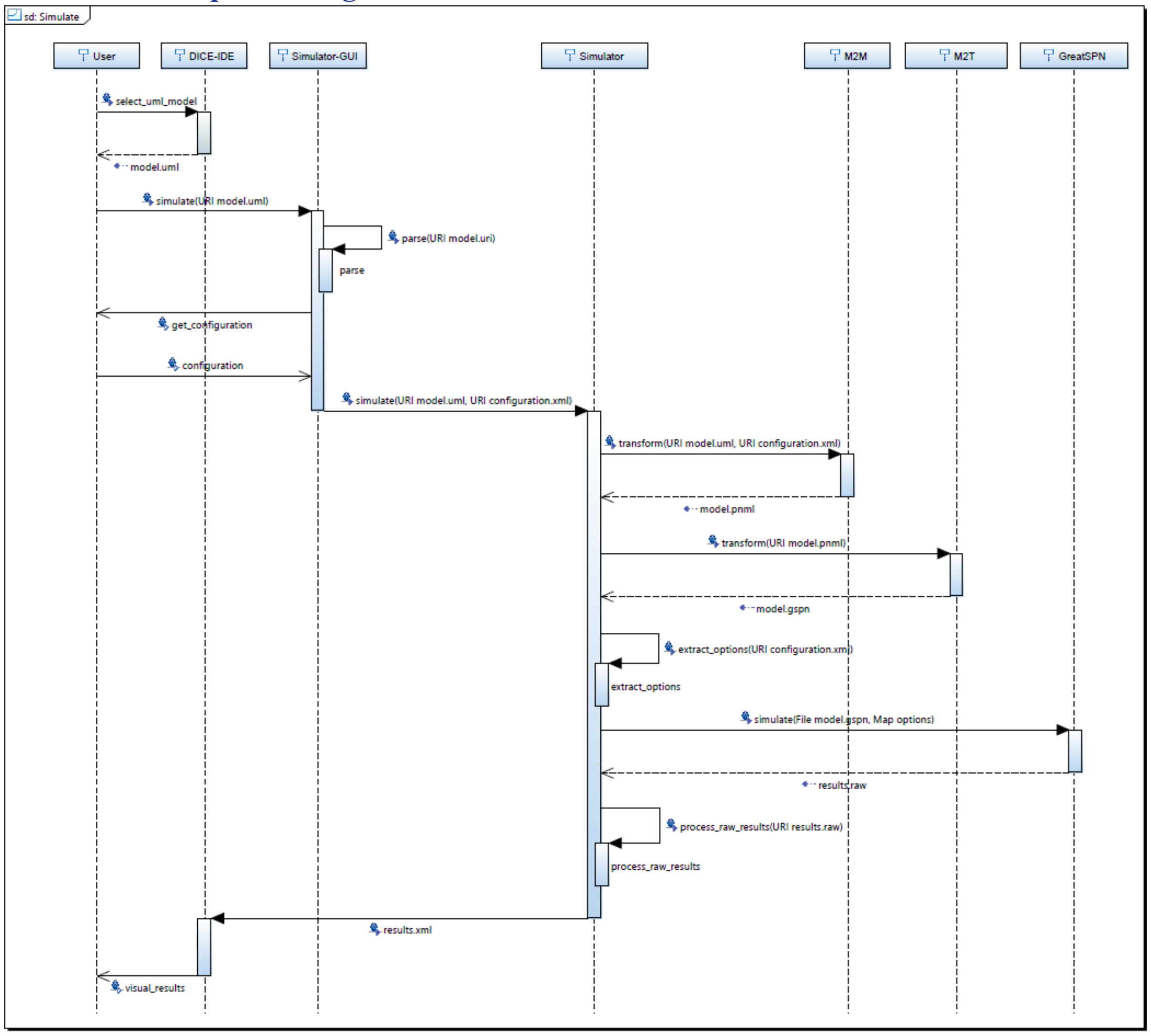
A.2.1 Description of interactions

The use case UC3.1 specifies that, from *an existing DPIM/DTSM level UML annotated model* (pre-condition), *the QA Engineer gets information about the predicted metric value in the technological environment being studied* (post-condition).
To obtain such information, the following steps need to be performed:

[8] UC3.1.1 (Verification of throughput from a DPIM DICE annotated UML model) is a specialization of UC3.1, and as such will not be considered in the present document to avoid redundancies

- The QA Engineer models a DPIM/DTSM model applying the DICE profile to a UML model using the DICE IDE.
- The QA Engineer starts a new simulation using the DICE-profiled UML models as input.
- The DICE-profiled UML models are translated within the simulation process to formal models, which can be automatically analysed, using M2M and M2T transformations.
- The simulation process is configured, specifying the kind of analysis to perform and the additional input data required to run the analysis.
- The simulation process is executed, i.e., the formal models are analysed using existing open-source evaluation tools (such as GreatSPN and JMT).
- The result produced by the evaluation tool is processed to generate a tool-independent report, conformant to a report model, with the assessment of performance and reliability metrics.
- The tool-independent report is fed into the DICE IDE and it is shown to the user in the GUI.

A.2.2 Sequence diagrams



A.2.3 Data flows

We have modelled the interactions among the *Simulation Tool components* as depicted in sequence diagram. For the sake of maintainability, the Simulator component has been split up in UI and non-UI components, i.e., Simulator-GUI and Simulator respectively.

Specifically, the sequence diagram depicted in sequence diagram describes the specific steps to simulate a DICE-profiled UML diagram using as an example the GreatSPN tool as the underlying evaluation tool, but others may be used.
As it can be seen in the figure, the modeling step is outside the scope of the Simulation phase, and the model to be analysed is supposed to pre-exist and is managed by the DICE IDE. When the user wants to simulate a model, s/he invokes the Simulator-GUI, which parses the model and asks the user any additional required information. When this information is obtained, the Simulator-GUI calls the Simulator that will handle the simulation in background.
The Simulator will then orchestrate the interaction among all the different modules. First, the M2M transformation module will create a PNML representation of the DICE-profiled model. Second, the PNML file will be transformed to a GreatSPN-specific Petri net description file. Third, the Simulator will start the analysis of the Petri net using GreatSPN. Finally, when the analysis ends, the raw results produced by GreatSPN will be converted into a formatted results file. This formatted results will be then sent to the DICE IDE that will show them to the user in a visual form.

A.3. Optimization

The requirements elicitation of D1.2 only considers a single use case that concerns the **Optimization** tool component (UC3.3). This use case can be summarized as:

ID	UC3.3
Title	Optimization of the deployment from a DDSM DICE annotated UML model with reliability and performance constraints.
Priority	Required
Actors	ARCHITECT
Pre-conditions	There exists a partially specified DDSM UML annotated model (in particular the number and type of storage and compute nodes are missing).
Post-conditions	The ARCHITECT gets a fully specified DDSM model minimizing the deployment cost and fulfilling QoS constraints specified in the input DTSM model.

A.3.1 Description of interactions

The use case UC3.3 specifies that, from *an existing DDSM UML annotated model* (pre-condition), *the ARCHITECT gets* a fully specified DDSM model minimizing the *deployment cost and fulfilling QoS constraints specified in the input DDSM model (post-condition).*

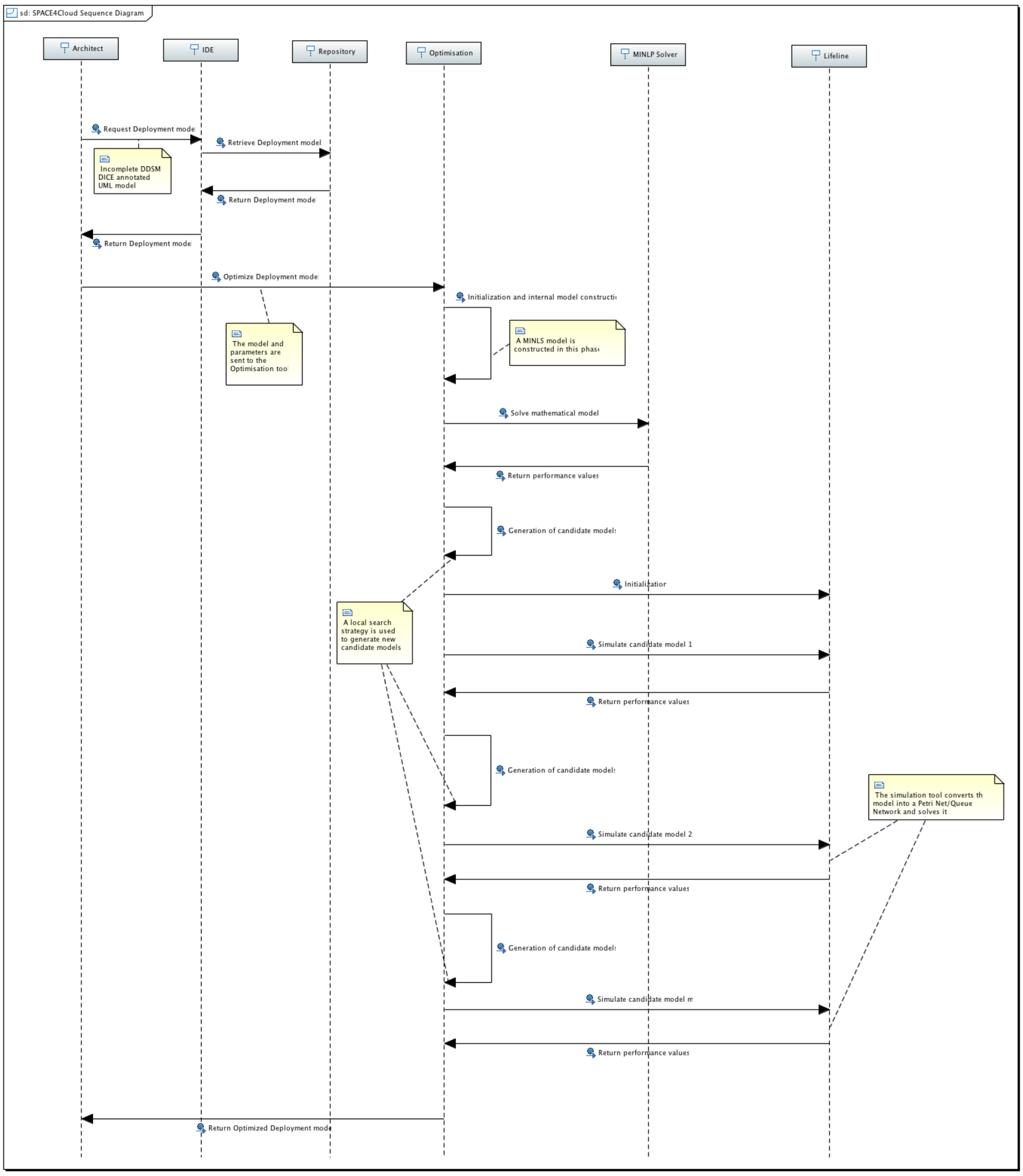
In this scenario the *ARCHITECT* is required to interact with the IDE in order to retrieve the deployment model, which is stored and versioned within the Repository component and optimized it by means of the Optimization tool. The deployment model is a DDSM DICE annotated UML model that may be incomplete, meaning that some quantitative information (namely the number and type of VMs to be used at runtime) are missing.

The user feeds the deployment model as input into the optimization tool along with some other relevant inputs that are used to control the behavior of the tool. In particular pieces of information as simulation accuracy, local search number of iteration, and QoS constraints to be applied to the deployment can be set by the user in this phase. Once the model, the constraints, and the properties are correctly loaded, the tool starts its execution by firstly generating a Mixed Integer Non-Linear Problem (MINLP) that is based on approximated formulae to evaluate DIA jobs execution time and is meant for a quick identification of a potentially high-quality initial solution.

A specific MINLP solver running as a service is invoked to solve the internal mathematical model. The so-obtained initial solution is afterwards used within the local search based optimization process representing the core of the tool. To this end the solution undergoes a set of transformations (also known as **moves**) that are applied iteratively with the aim of progressively reducing the deployment cost guaranteeing at the same time the fulfillment of the constraints. The deployment models generated during this phase are turned into Petri Nets or Queue networks by the Simulation tools, which is in charge of performing suitable model-to-model transformations and solve the resulting performance models. The outcome of performance evaluation process is a set of performance metrics that are returned to the optimization tool. Such pieces of information are used to drive the next steps of the search process towards less and less costly deployments. According to the level of parallelism granted by the **Simulation** tool more than one candidate solution can be generated and evaluated in parallel.

At the end of this scenario the best deployment model obtained (complete DDSM) is presented to the user along with its related performance metrics. If, for whatever reason, such an outcome does not fit the user’s expectation, s/he will perform the changes s/he deems appropriate and re-execute the optimization process from the new deployment.

A.3.2 Sequence diagrams


sd: SPACE4Cloud Sequence Diagram
Architect
IDE
Repository
Optimisation
MINLP Solver
Lifeline
Request Deployment mode
Incomplete DDSM DICE annotated UML model
Retrieve Deployment model
Return Deployment mode
Return Deployment mode
Optimize Deployment mode
The model and parameters are sent to the Optimisation too
Initialization and internal model constructi
A MINLS model is constructed in this phas
Solve mathematical model
Return performance values
Generation of candidate model:
Initialization
A local search strategy is used to generate new candidate models
Simulate candidate model 1
Return performance values
Generation of candidate model:
The simulation tool converts th model into a Petri Net/Queue Network and solves it
Simulate candidate model 2
Return performance values
Generation of candidate model:
Simulate candidate model n
Return performance values
Return Optimized Deployment mode

A.3.3 Data flows

We have modelled the interactions among the **Optimization** tool component as depicted in Fig. 1. The modelling step is outside the scope of the optimization phase, and the model to be analysed is supposed to pre-exist and is managed by the DICE IDE. When the user wants to optimize a model, s/he invokes the **Optimization** tool, which loads the model and asks the user any additional required information. More in details, the optimization is performed through the following steps:

1. The model is chosen from the Repository. Repository sends the model to IDE and the IDE to the user.
2. The possibly incomplete DICE DDSM annotated model, the QoS constraints, and properties to use for the analysis are sent to the **Optimization** tool.
3. A first-approximation mathematical optimization model is generated and sent to a suitable MINLP solver. Some performance metrics are returned to the **Optimization** tool.

4. The **Simulation** tool is initialized and set up.
5. A certain number of fully defined deployments model are generated and sent to the **Simulation** tool to perform the required analysis. The outcome is a set of performance indicator that are returned to the **Optimization** tool

The outcome of the **Optimization** (the cheapest feasible deployment identified) and its related metrics are reported to the user.

A.4. Verification

A.4.1 Description of interactions

The user performs verification on the current model loaded in the IDE or selects the model from the repository; in the last case, the model is first loaded and then showed in the IDE.

The verification is performed on annotated DTSM models which already contain all the information required to perform the analysis.

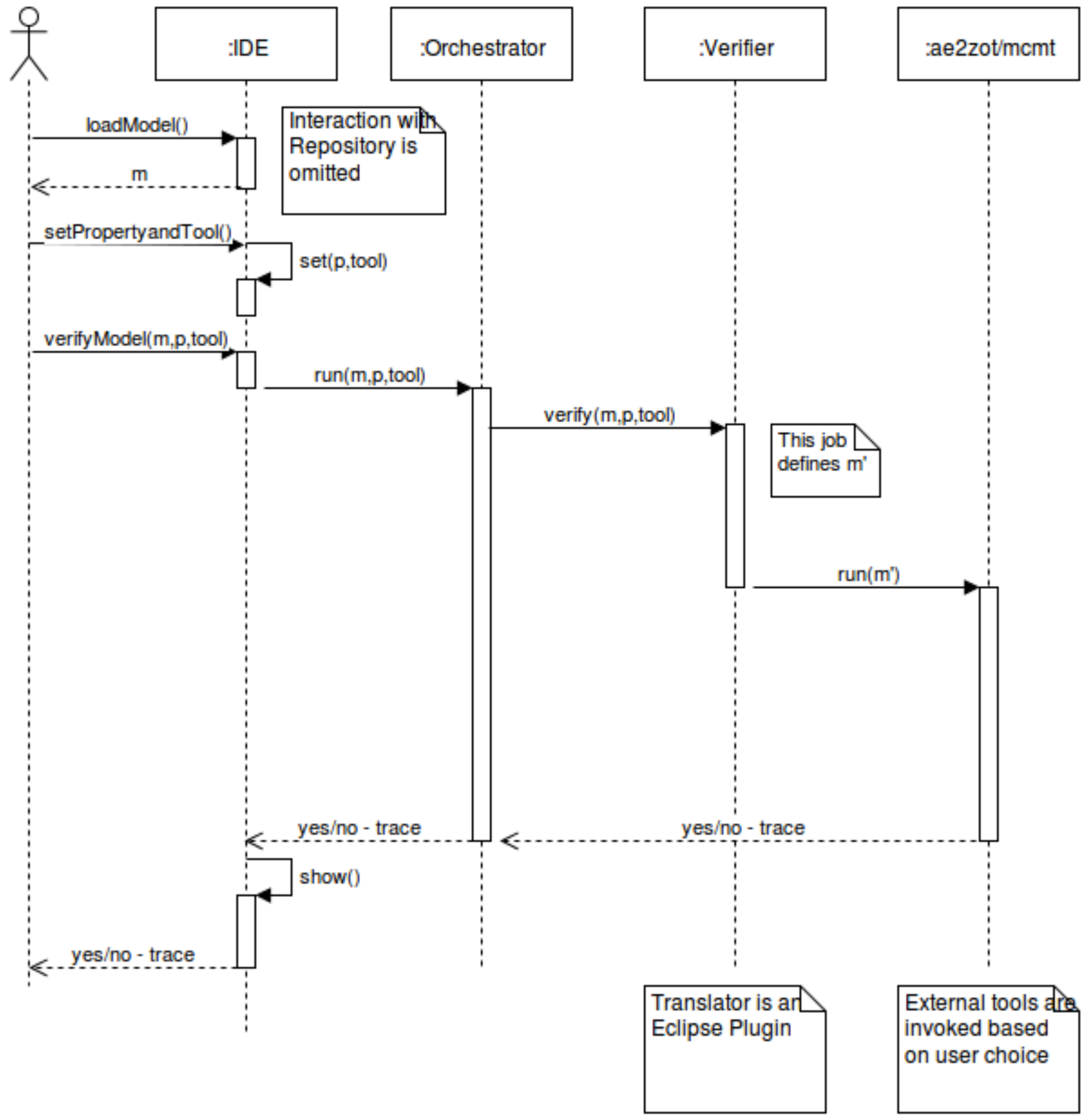
The user selects a (safety/privacy) property to be checked possibly using templates (which are compliant with the definition of the class of desired properties for the application, specified in the design phase at DPIM level) and the tool for the analysis.

After the definition of the property of interest, the orchestrator submits the verification request to the ***Verification plugin***. The Verification plugin converts DICE UML model and the property to be verified into a formal model that is suitable for verification (e.g., a temporal logic model).

Based on the class of property to verify or on the approach the user applies, the plugin selects the appropriate solver which analyzes the formal model against the property and determines whether the property holds for the system or not.

The outcome is sent to orchestrator and then to the IDE which presents the result. It shows whether the property is fulfilled or not; and, if the property is violated, the IDE presents the trace of the system that violates it.

A.4.2 Sequence diagrams



A.4.3 Data flows

1. The model is chosen from the Repository. Repository sends the model to IDE.
2. The annotated DTSM model, the property and the tool to use for the analysis are sent to the Verification plugin.
3. The outcome of the verification (yes/no) and the trace (if any) is sent to the orchestrator component which then reports the results in the IDE.

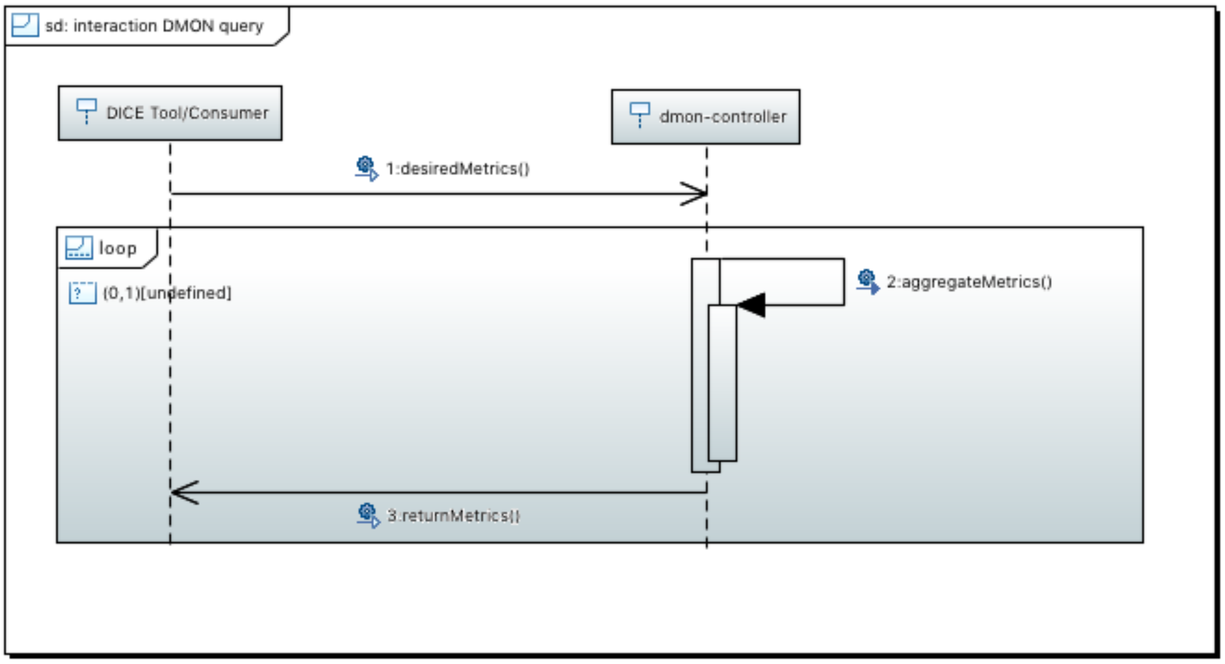
A.5. Monitoring

ID: UC4.1 Title: Monitoring a Big Data framework (Scenario)
[Combination of UC 4.1.1 to UC 4.1.3]

A.5.1 ID: UC 4.1.1. Title: Metrics Specification

A.5.1.1. Description

Any user or DICE tool can query the monitoring platform. The query request needs to contain a query string similar to the one used in Kibana, a time interval (or time math representation of interval). It is also possible to specify the type of output (csv, json, rdf+xml, plain). The dmon-controller then receives this request and generates the elasticsearch query that is executed and then returned in the specified output format.



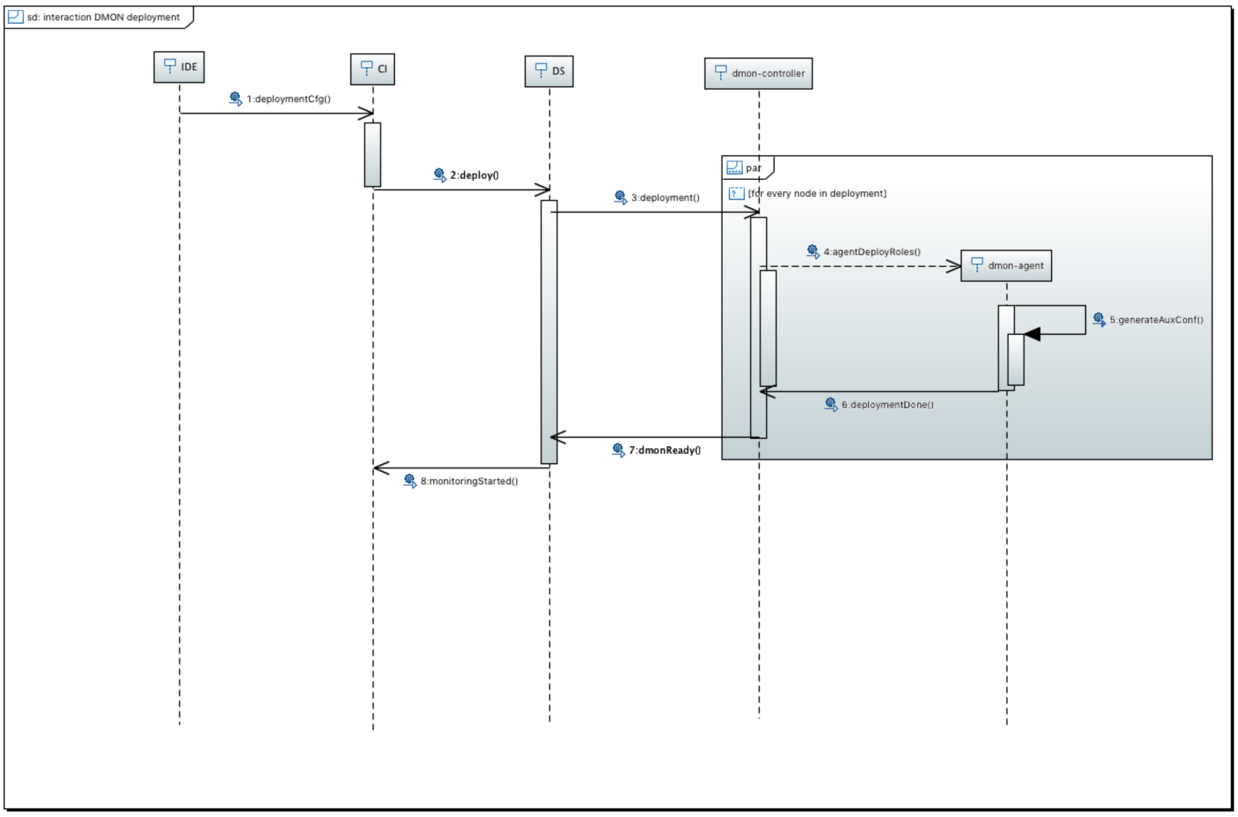
A.5.1.2. Data Flow

The request and its payload are sent to the dmon-controller. The data from ElasticSearch is sent to dmon-controller where it is further processed (if it is required) and then sent to its final destination.

A.5.2 ID: UC 4.1.2 Title: Monitoring tools registration

A.5.2.1. Description

The current deployment specification is sent to the Continuous Integration (CI) tool which then sends the platform specific deployment to the Deployment service (DS) which enacts this. The DS send a request that contains the FQDN, credentials and roles of each node from the deployment to the dmon-controller which in turn deploys in parallel all dmon-agent instances on these nodes. Based on the roles assigned to each node the monitoring auxiliary components are installed and configured. When everything is done a response is given to the DS which sends that to the CI.



A.5.2.2. Data Flow

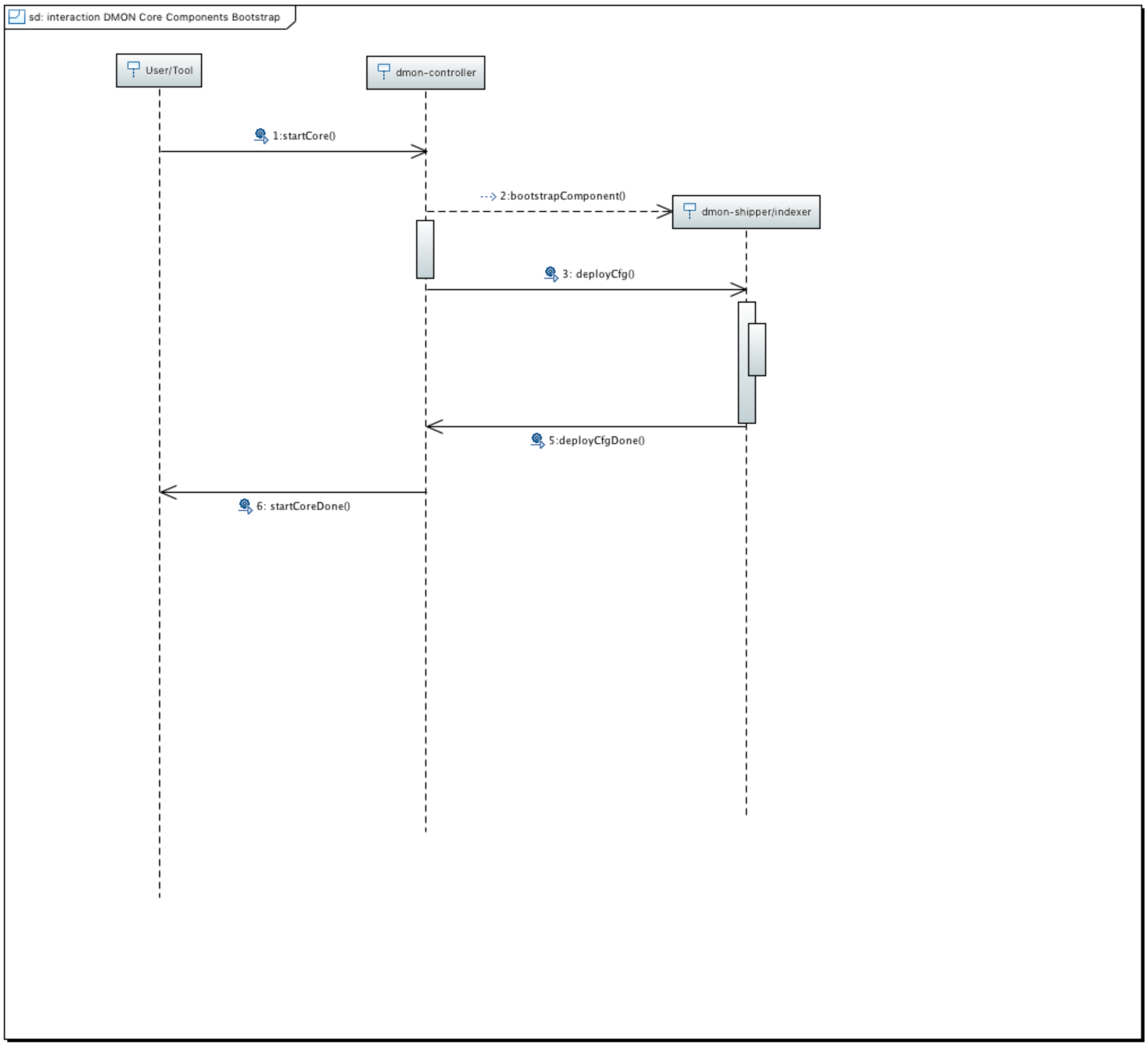
Data is represented only by the request payloads (json).

A.5.3 ID: UC 4.1.3 Title: Monitoring Data storage (Start ES and LS)

A.5.3.1. Description

Any user or tool that has access to the dmon-controller Management API can bootstrap additional monitoring platform core components. The dmon-shipper controls an instance of logstash server while dmon-indexer controls an instance of elasticsearch. It is also possible to start/stop and reconfigure each of these components. The only prerequisite is that there exist a registered newly provisioned VM.

This diagram represents both first deployment and possible scaling scenarios. For both scenarios a prerequisite is that there exist provisioned VMs on which the services and components can be bootstrapped.



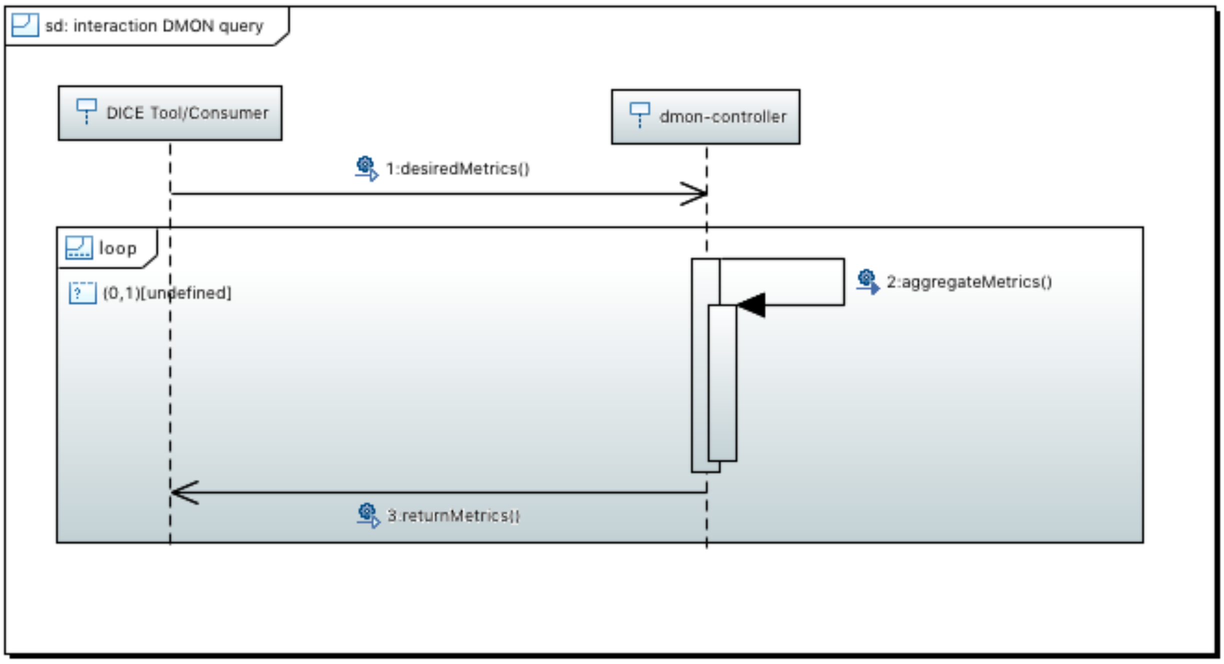
A.5.3.2. Data Flow

The only data is the request payloads (json).

A.5.4 ID: UC 4.2. Title: Data Warehouse Query

A.5.4.1. Description

In the DICE solution the data warehouse is represented by the instance (or cluster) of ElasticSearch. Because of this querying the data warehouse is done the same way as in UC4.1. It is possible to export both the data and the indexes from any ElasticSearch instances. This can be, at a later time, imported into the monitoring platform and again queried the same way as before. It is even possible to import this data together with its index into a completely separate ElasticSearch instance. By removing older unused indexes from the monitoring platform we can limit the amount of computational resources needed by it and store potentially valuable data for later use.



A.5.4.2. Data Flow

The request and its payload are sent to the dmon-controller. The data from elasticsearch is sent to dmon-controller where it is further processed (if it is required) and then sent to its final destination.

A.6. Enhancement

A.6.1 Description of interactions

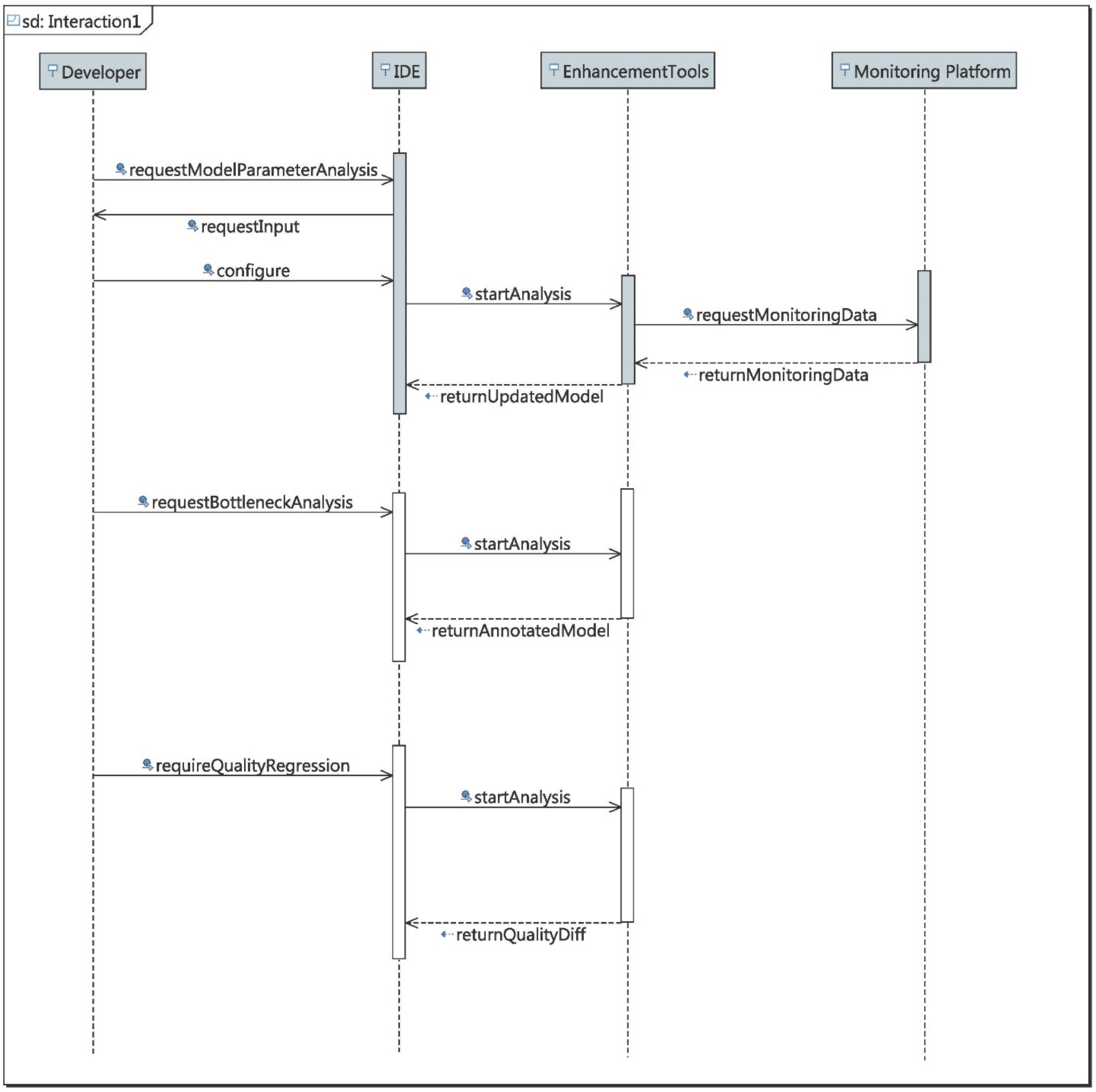
The user activates the enhancement analysis for model parameter analysis and inputs the parameters for the Enhancement tool through IDE. Then the IDE triggers the start of the Enhancement tool.

If the user requires to update the model parameters, then according to the parameters set by the user the Enhancement tool queries the specific monitoring data, such as CPU utilization, response time and throughput, used for the analysis from the monitoring platform. Statistical analysis is performed based on the runtime monitoring measurements and the tool generates the new parameters for the model. Finally with the new parameters, the tool updates the model directly and return the updated model back to the IDE along with a report generated based on the performance of the application.

If the user requires for bottleneck identification for the current design from the IDE, then the Enhancement tool analyzes the current UML model and highlights software or hardware bottlenecks based on testing results and return the result back to the IDE.

The user may also request to examine quality regressions in two versions of the application. Then Enhancement tool analyses quality differences between versions by operating directly on the monitoring data and return the result back to the IDE.

A.6.2 Sequence diagrams



A.6.3 Data flows

The major data exchange happens when the enhancement tool updates the models. The tool will query monitoring data from the Monitoring Platform and uses these data to analyse the parameters of the performance models.

A.7. Trace checking

A.7.1 Description of interactions

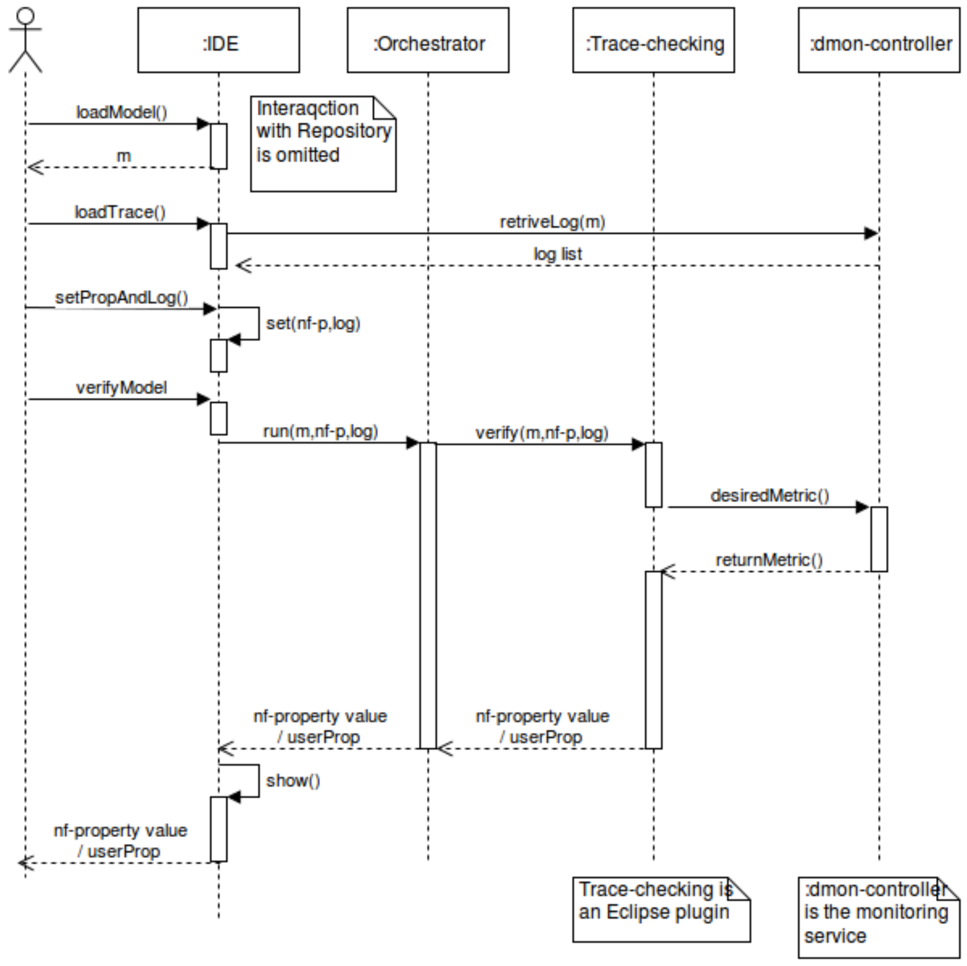
The user activates trace checking on the current model loaded in the IDE or selects the model from the repository; in the last case, the model is first loaded and then showed in the IDE.

The user selects a non-functional property (metric) to be checked from a list (compliant with the model/properties that are supported in the framework), a time window, and a log from the Monitoring platform to be checked over the specified time window.

The orchestrator submits the trace-checking request to the ***Trace-checking plugin***. The trace-checking plugin then activates a trace-checking job on the selected trace.

The outcome is sent to orchestrator and then to the IDE which presents the result. It shows the values of the non-functional properties that are extracted from the trace compared with the values of the same properties defined at design time.

A.7.2 Sequence diagrams


:IDE
:Orchestrator
:Trace-checking
:dmon-controller
loadModel()
m
Interaqction with Repository is omitted
loadTrace()
retriveLog(m)
log list
setPropAndLog()
set(nf-p,log)
verifyModel
run(m,nf-p,log)
verify(m,nf-p,log)
desiredMetric()
returnMetric()
nf-property value / userProp
nf-property value / userProp
show()
nf-property value / userProp
Trace-checking is an Eclipse plugin
:dmon-controller is the monitoring service

A.7.3 Data flows

1. The model is chosen from the Repository. Repository sends the model to IDE.
2. The list of the logs is retrieved in the Monitoring platform; it sends the list to the IDE.
3. The non-functional property chosen by the user (based on the DPIM annotation) is sent to the Trace-checking plugin along with the time-window.
4. The outcome of the trace-checking is sent by the Trace-checking plugin to the orchestrator component which then reports the results in the IDE.

Next scenario

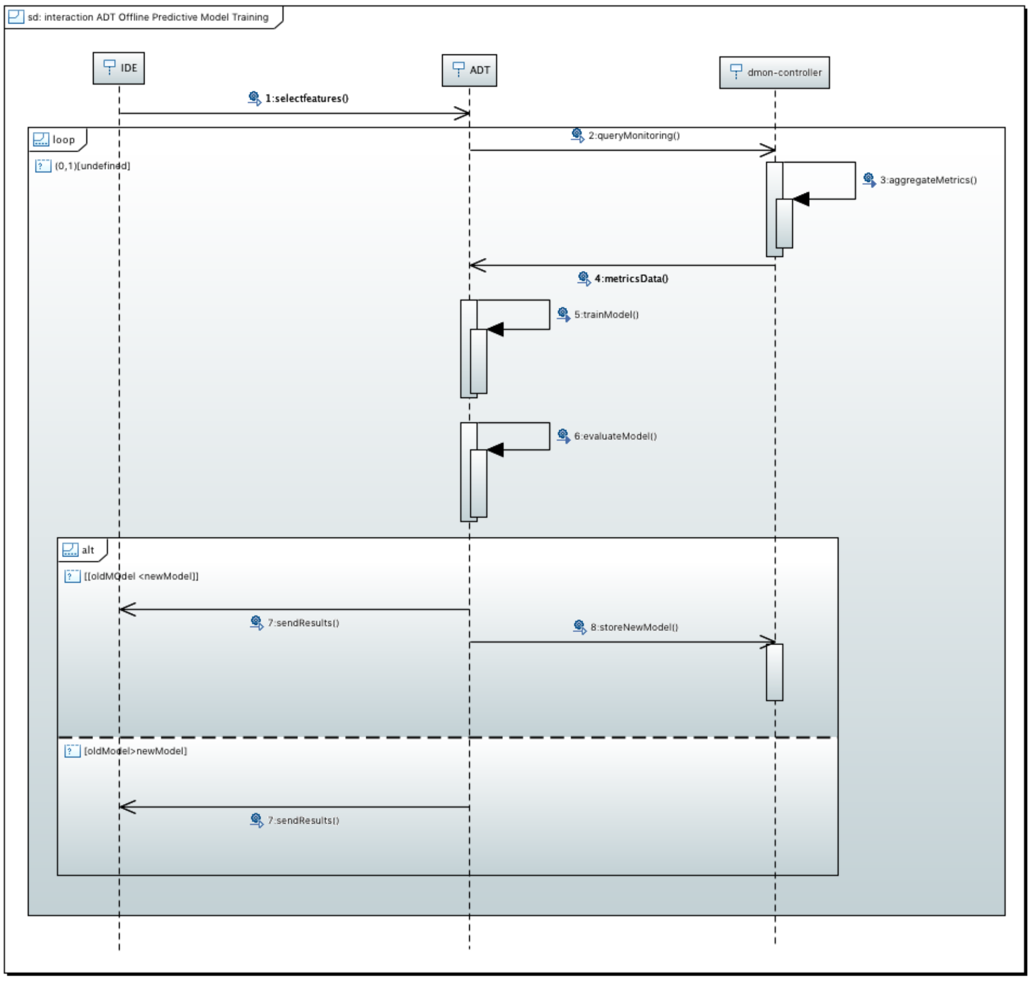
If the non-functional properties have specific relation (order) with respect to the values established at design time then the tool reports the properties, defined at DPIM level that might be violated.

A.8. Anomaly detection

A.8.1 ID: UC 4.5 Title: Anomaly Detection Model Training

A.8.1.1. Description

In order to create viable predictive models that are able to detect not only point anomalies but also contextual anomalies we need a robust training methodology. In the case of DICE a user will have to select a subset of features that are stored in the Monitoring Platform. This is then used to query the controller and a dataset is created. The resulting data is then used to train and subsequently validate a predictive model. If the trained model has a good performance it is stored, if not then it is discarded. The type of anomaly detection algorithm is not yet defined. It is scheduled for the second year of the project.



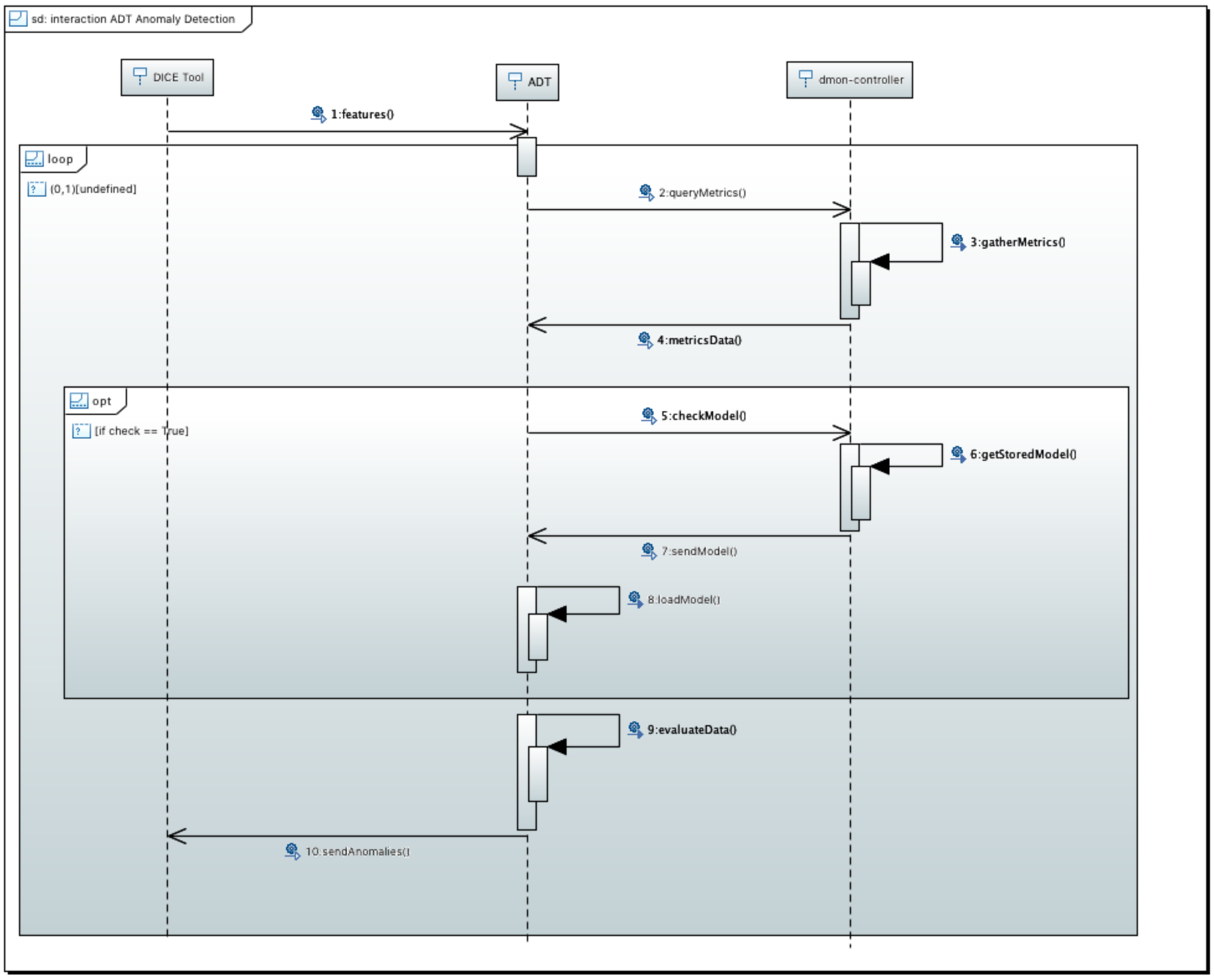
A.8.1.2. Data Flow:

Data is consumed from the DICE Monitoring platform.

A.8.2 ID: UC 4.6. Title: Offline Anomaly Detection

A.8.2.1. Description

Any tool or user can issue a request to the anomaly detection tool. This request has to define a set of features and timeframe on which the anomaly detection will take place. If anomaly detection tool (ADT) model training has been done/initialized for this subset of features the service will check the given timeframe for anomalies. This requires the querying of the monitoring solution and fetching the pre-trained predictive model. The best performing predictive model is then fetched and instantiated. If a better performing model for the given dataset is detected than the one already instantiated the new model will be loaded. This check happens before a new batch of data is loaded.



A.8.2.2. Data Flow

Data is consumed from the DICE Monitoring platform. This is true both for the required metrics as well as the trained predictive models.

A.9. Delivery tool

A.9.1 Description of interactions

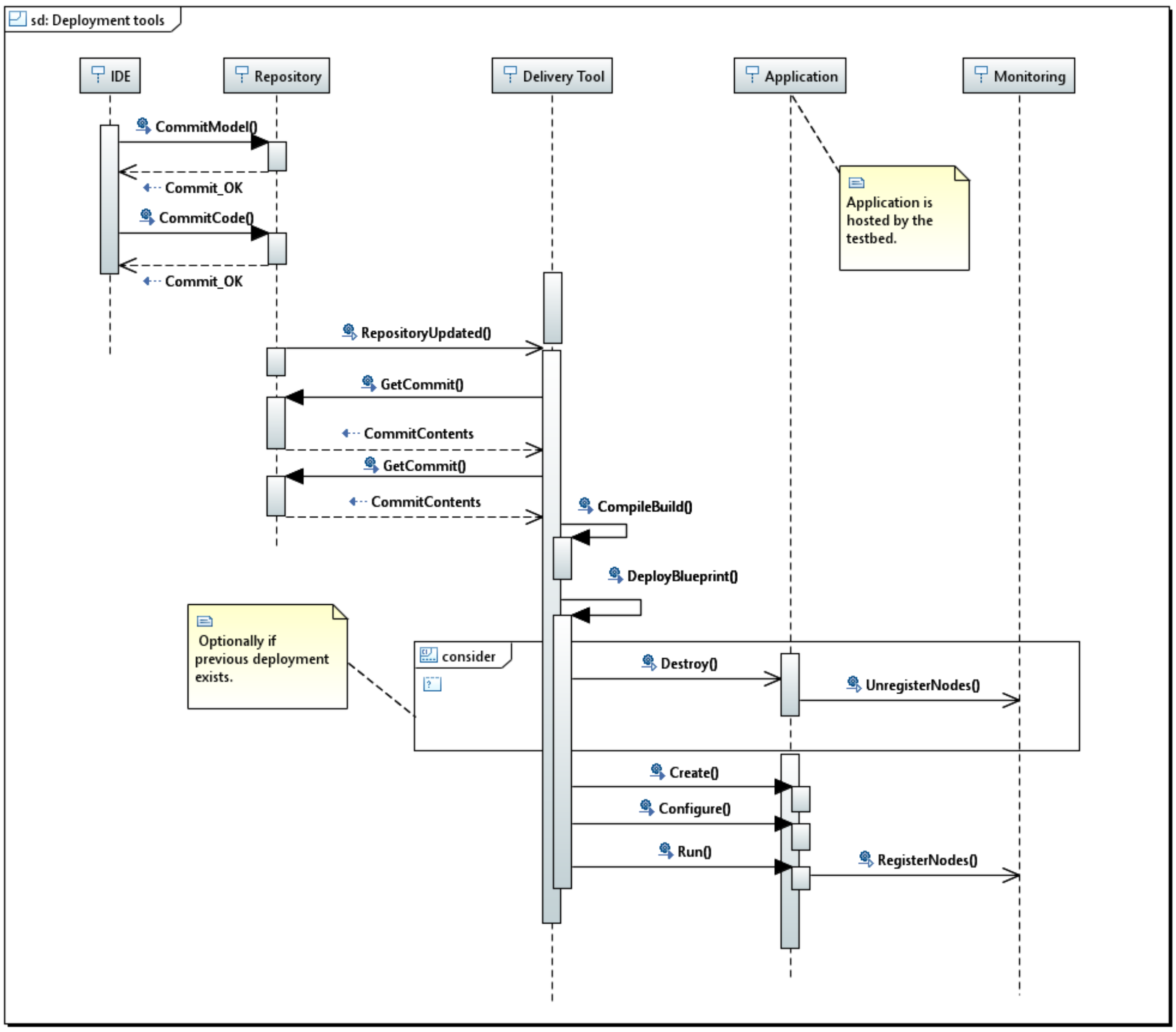
The Continuous integration sequence starts with the programmer's code and models, and results in a deployed platform services and the application in the test bed, quality (non-functional) tests run, and a result of the non-functional tests available in the Continuous Integration component of the DICE Delivery Tool. The part 1 of the use case describes the feature up to the point of the application deployed and configured in the test bed.

The actors (normally programmers) use the IDE to model the application at the DPIM and DDSM level. They also edit the application code. At some point during their development (but normally at least once a day) they decide that the application is ready to be deployed and, possibly, tested in the test bed. They verify that the code compiles and checks in their IDE. The next step for them is to commit the model of the application in the Repository. Also, they commit their code changes in the Repository.

The Continuous Integration part of the Delivery Tool receives a notification about the Repository update (conversely, it polls the status of the Repository periodically until it learns of a change) of the project. It then fetches the updates from the Repository - both the code and the model. Then it performs the compilation and assembly of any user-written code.

Finally, the Delivery Tools initiate the deployment and configuration phase. In the current implementation, it first destroys any existing deployment of the application along with all the platform services needed for the application. This effectively cleans up the environment and frees the resources. Then it proceeds by first creating the environment in the test bed, provisioning any virtual resources (computation, storage, and networking) required according to the application model. Then it configures the services and the application, and finally it runs the services and application components. This last step also includes the step of registering the nodes running services to the Monitoring in order for the application components to initiate the streams of runtime metrics.

A.9.2 Sequence diagrams



A.9.3 Data flows

In this scenario, the user produces the application code. We assume that the IDE tools also trigger the model-to-text transformation, which produces an OASIS TOSCA document in YAML format. This document contains a blueprint, describing the application to be deployed as well as the configuration for each service in the blueprint.

The Delivery Tool consumes the TOSCA document and, based on the blueprint description, deploys and configures the application in the test bed.

A.9.4 Continuous integration

A.9.4.1. Description of interactions

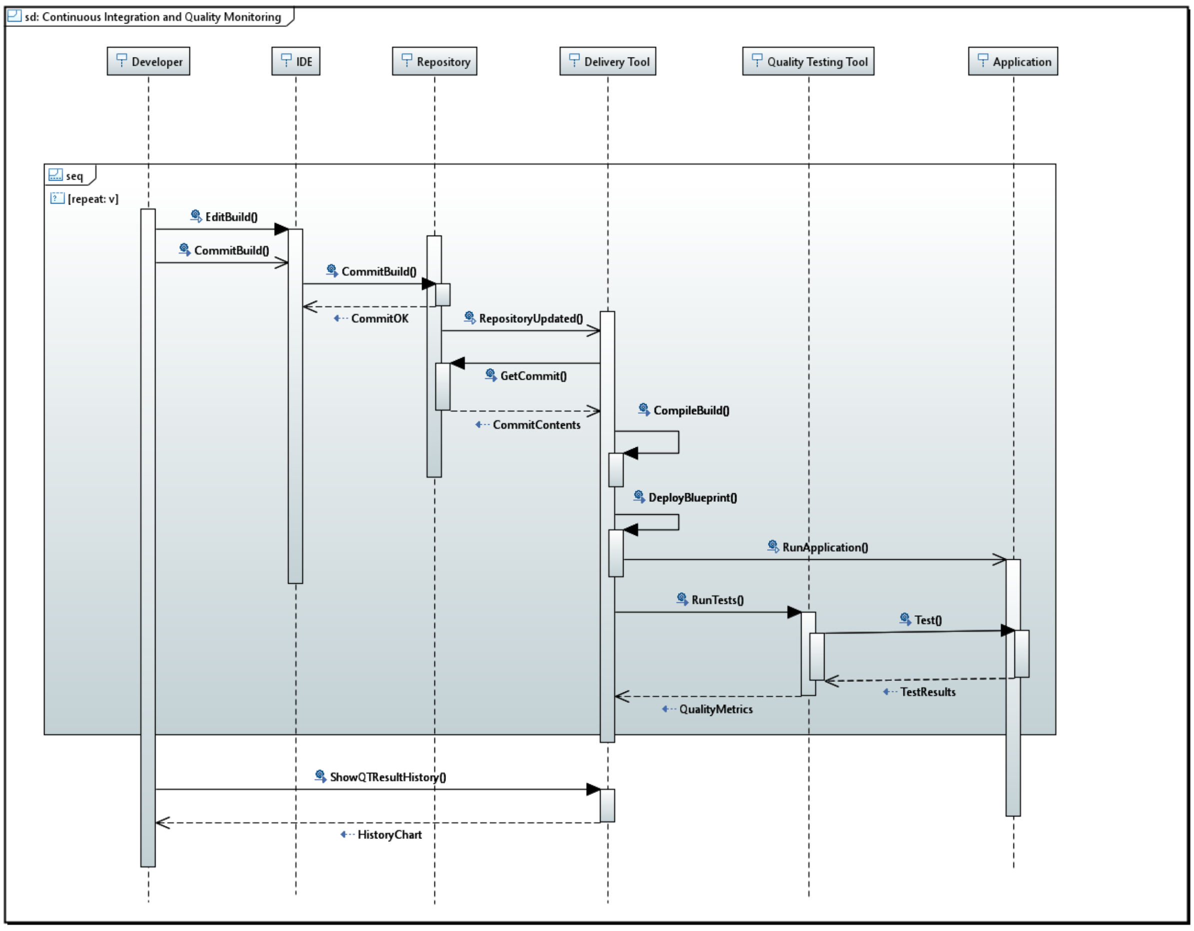
This sequence diagram emphasizes the continuous nature of the integration, which includes getting a feedback using the quality testing tools to assess the validity of the current build of the application.

The previous sequence diagram shows in further details what this sequence diagram shows at a more abstract level. The developer, who edits the current build as the code and models, requests of the IDE to commit the build. IDE pushes the build to the Repository, which, in turn, notifies the Delivery Tool about the update. The Delivery Tool fetches the build's commit contents, compiles them and, if all goes well, deploys the blueprint from the built into the testbed. The result of this interaction is an application, representing the current build, which runs in the test bed. Like in the previous sequence diagram, the deployment action replaces the deploys of any previous builds and their supporting platform services in the testbed.

Then the Delivery Tool invokes the Quality Testing tool, which exposes the application to a test workload. With the help of the Monitoring Tools (omitted from the diagram for clarity), it produces the quality metrics describing the build's non-functional properties.

The whole process repeats for each new build, which represents a part of the application's version. The Delivery Tool stores the history of this information. The developer can then at any time request of the Delivery Tool to show the Quality Test results history, and as a result should obtain a chart (or some other time series representation) showing the build performance through time.

A.9.4.2. Sequence diagram



A.9.4.3. Data flows

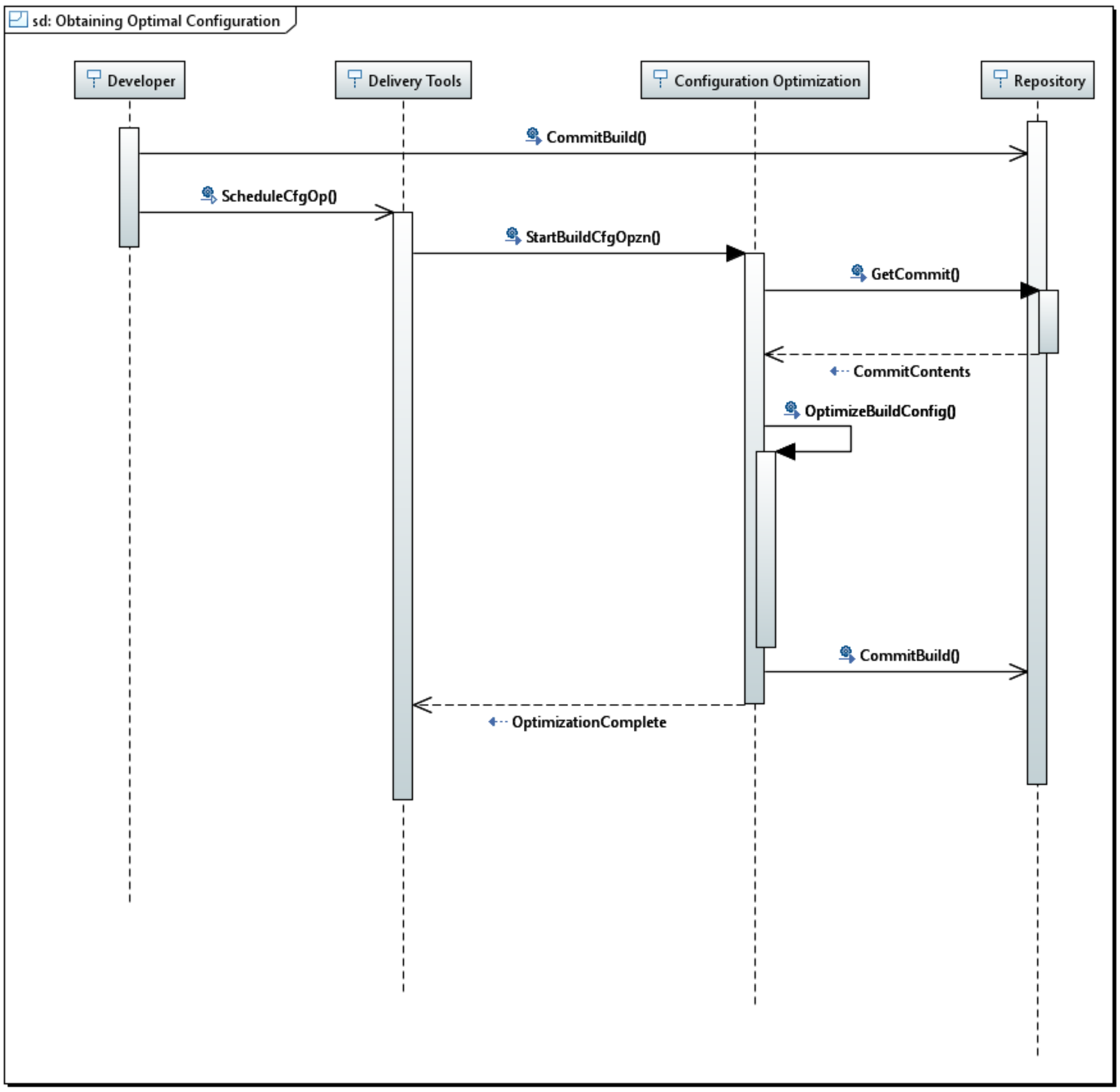
The first part of this scenario's data flows repeat the ones described in the previous scenario: the user must first produce the application code and have an OASIS TOSCA YAML document blueprint built. The Delivery Tool consumes the TOSCA document and, based on the blueprint description, deploys and configures the application in the test bed.
The Delivery Tools are configured to run quality tests to a certain extent (frequent short tests, occasional longer tests). They run the tests, passing any configuration needed to the Quality Testing tool. The outcome of the tests are a scalar or an array of scalars to be stored in the Delivery Tool's database.
The developers then request the history of a project or an application, possibly specifying the time range of the query and the type of metric to inspect. As a response they receive a graphical or tabular representation of the metric history.

A.9.5 Obtaining configuration recommendation

A.9.5.1. Description of interactions

The configuration recommendation is a result of a sophisticated process, which is carried out by the Configuration Optimization tool. This sequence illustrates how an actor (typically a developer) puts the process in motion.
As with the previous workflows, the developer first works on the code and the model of the application, and at some point commits both to the Repository. Considering that the configuration optimization is still a relatively lengthy non-interactive process, which also needs to use the resources of the testbed, we preferably move it away from the IDE. Therefore, the developer needs to schedule the configuration optimization, and this is possible through Delivery Tools.
The Delivery Tools invoke the Configuration Optimization tool, which in turn needs to first obtain the TOSCA application topology from the Repository as a part of the build's commit. It then starts its iteration towards an optimal optimization.
When finished, it posts the result back at the Repository.
This approach of the scenario is a slightly different take from the one described in Section [Configuration Optimization]. Here we emphasize the use of the Configuration Optimization tool in an asynchronous (background processing) approach, where the user can set the optimization to run and then forget it until it finishes.

A.9.5.2. Sequence diagram



A.9.5.3. Data flows

The main data item to flow through this scenario consists of the configuration, i.e., the specific values of various services' and application parameters. They start with the Developer, who either sets some initial values from defaults, manual guesses or any previous runs of the Configuration Optimization. The configuration gets stored in the Repository with the current build's version.
The Configuration Optimization updates the configuration values as it iterates through its algorithm, and the last set values are said to be optimal.
The recommended (optimal) configuration is then available in the Repository for the developers to use with the subsequent deploys and further application development.

A.10. Quality testing

The requirements elicitation of D1.2 considers the following scenarios (U5.10, U5.11) that concerns the quality testing component.

ID:	U5.10
Title:	Performing the quality testing
Task:	T5.3

Priority:	REQUIRED
Actor 1:	QTESTING_TOOLS
Actor 2:	MONITORING_TOOLS
Actor 3:	TESTBED
Actor 4:	N/A
Flow of Events:	1. QTESTING_TOOLS starts injecting load in the APPLICATION after signalling to MONITORING_TOOLS the start of a test 2. The test plan is executed taking into consideration the dimensions to be explored (e.g. performance, reliability, etc.) 3. If requested, QTESTING_TOOLS may access TESTBED APIs to perform the test
Pre-conditions:	1. A quality test has been requested in some scenario 2. Test resources have been provisioned
Post-conditions:	1. Test data has been collected by MONITORING_TOOLS
Exceptions:	N/A
Data Exchanges:	N/A

A.10.1 Description of interactions

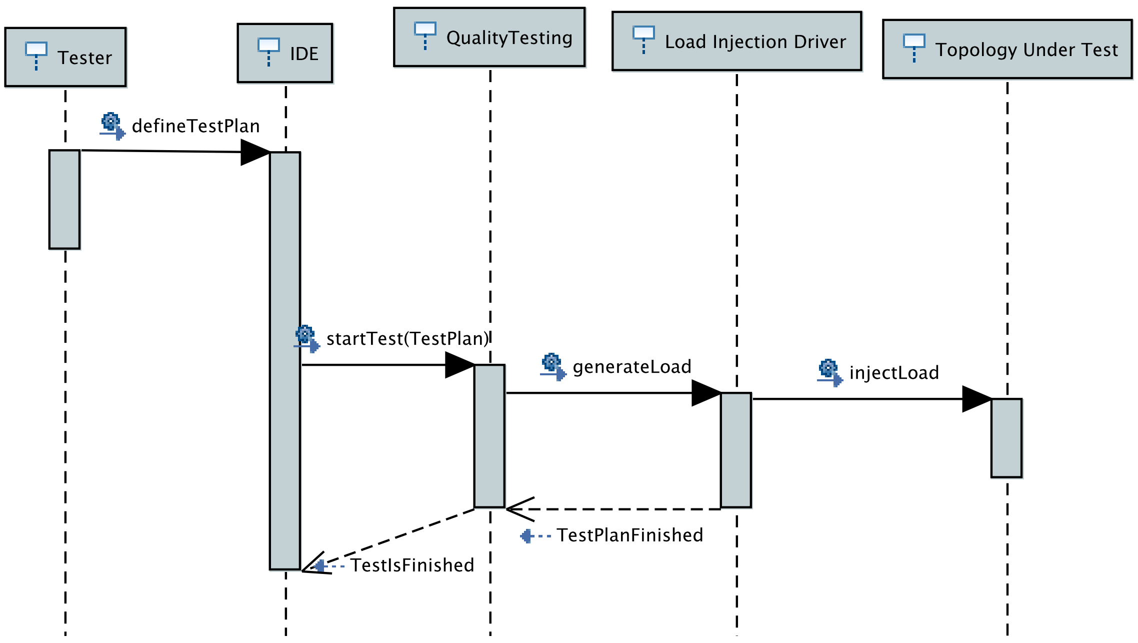
1- The tester defines a test plan using the interface provided by the IDE. A test plan consists of a test scenario comprising of a workload over time and the data source that needs to generate the real load.

2- The tester then initiates the test using the IDE.

3- The load will be injected using appropriate injection driver including the Fault Injection Tool for different technologies involved in the system under test.

4- When the test plan has been finished, the quality testing tool will inform the user about the outcome of the test generation.

A.10.2 Sequence diagrams



A.10.3 Data flows

The main data entity in quality testing tool is the test plan which needs to be in a user readable XML like format and compatible with other similar tool such as JMeter. The test plan should define the notion of time unit and the level of the load that will be injected during time periods.

A.11. Fault injection

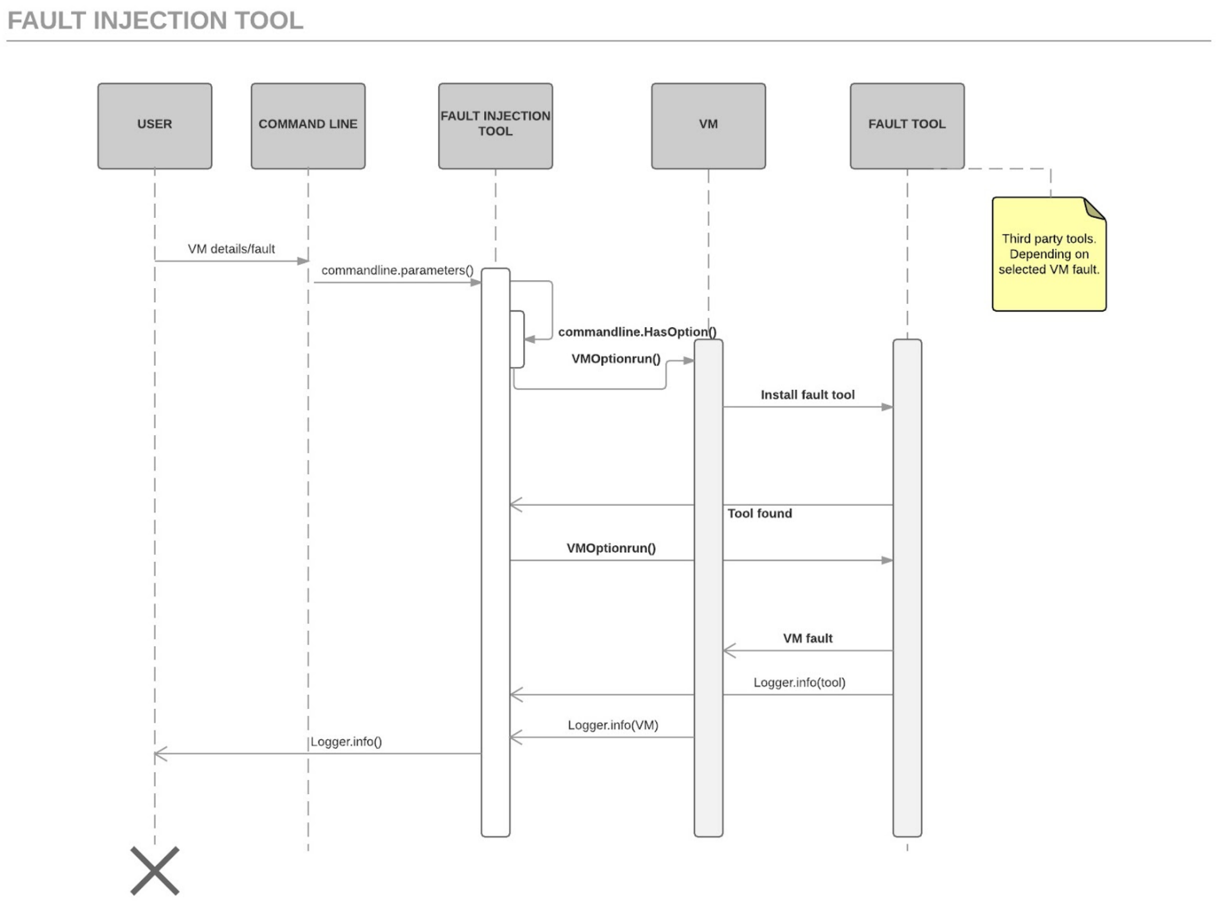
A.11.1 ID R5.14.2

ID	R5.14.2
Title	Trigger deliberate outages and problems to assess the application's behavior under faults
Priority	Required
Actors	User, VM

A.11.1.1. Description of interactions

- The User starts the Fault Injection tool.
- Using command line options they pass fault and required details.
- Fault Injection Tool Connects to VM and begins Fault.
- Fault information and status is pass to Fault Injection tool which stores this within a log file.

A.11.1.2.Sequence diagrams



A.11.1.3.Data flows

As seen in the sequence diagram, the user begins the request to the Fault Injection tool. The Fault Injection tool then connects to the VM and stores the output of the Fault in a log file for user access.

A.11.2 ID R5.30

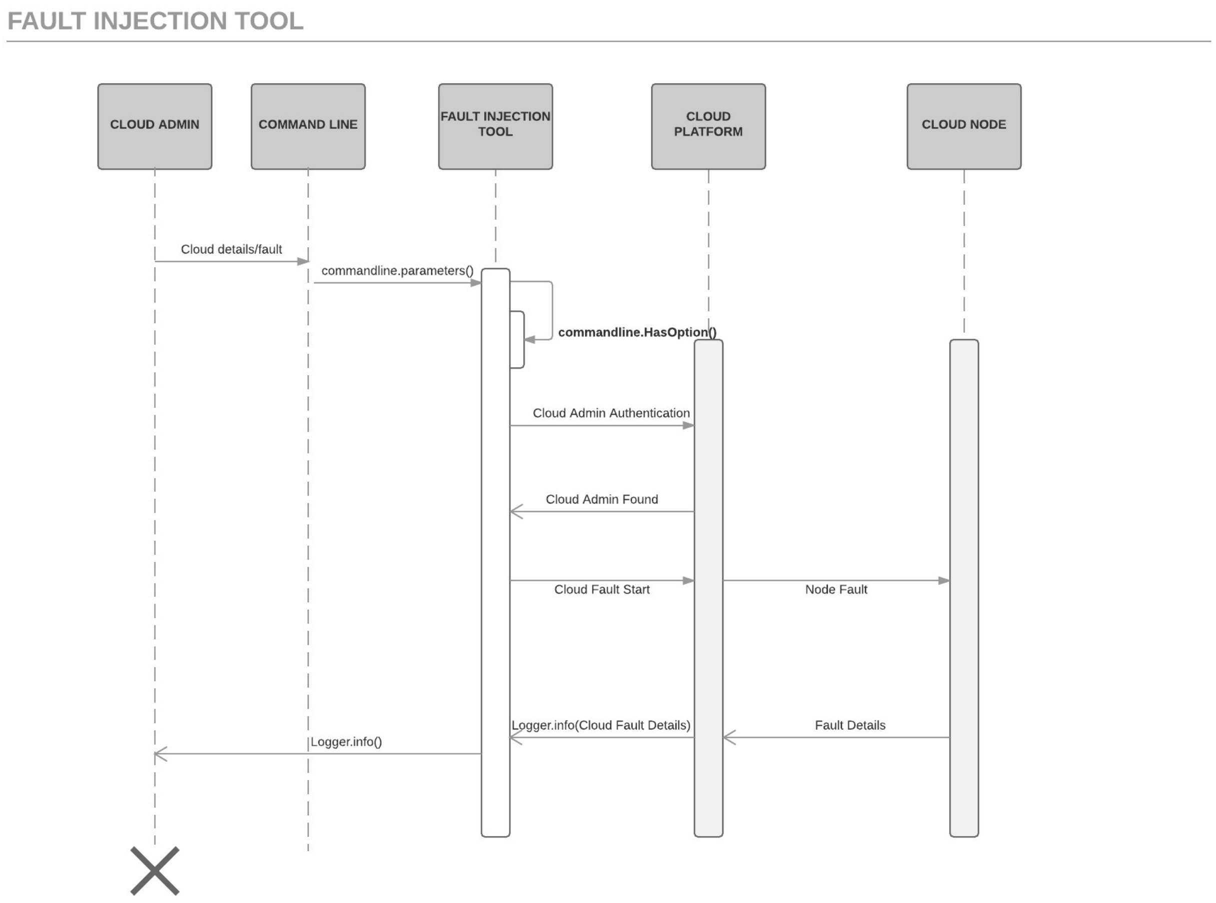
ID	R5.30
Title	Induced faults in the guest environment
Priority	Required
Actors	Cloud Admin, Fault Injection Tool, Cloud Platform

A.11.2.1.Description of interactions

- The Cloud Admin starts the Fault Injection tool.
- Using command line options they pass the required Cloud Level fault and required authentication details.
- Fault Injection Tool Connects to Cloud Platform and authenticates to begin the fault.
- The Fault is then started on the required physical node.

- Fault information and status is pass to Fault Injection tool which stores this within a log file.

A.11.2.2.Sequence diagrams



A.11.2.3.Data flows

As seen in the sequence diagram, the Cloud Admin starts the request. The Fault Injection tool will execute the requested action, and before returning the result to the Cloud Admin via storing the details within an accessible Log file.

A.12. Configuration optimization

The requirements elicitation of D1.2 considers U5.5 as the main scenario that concerns the Configuration Optimization component.

ID:	U5.5
Title:	Obtaining configuration recommendation
Task:	T5.1
Priority:	REQUIRED
Actor 1:	DEVELOPER
Actor 2:	DEPLOYMENT_TOOLS

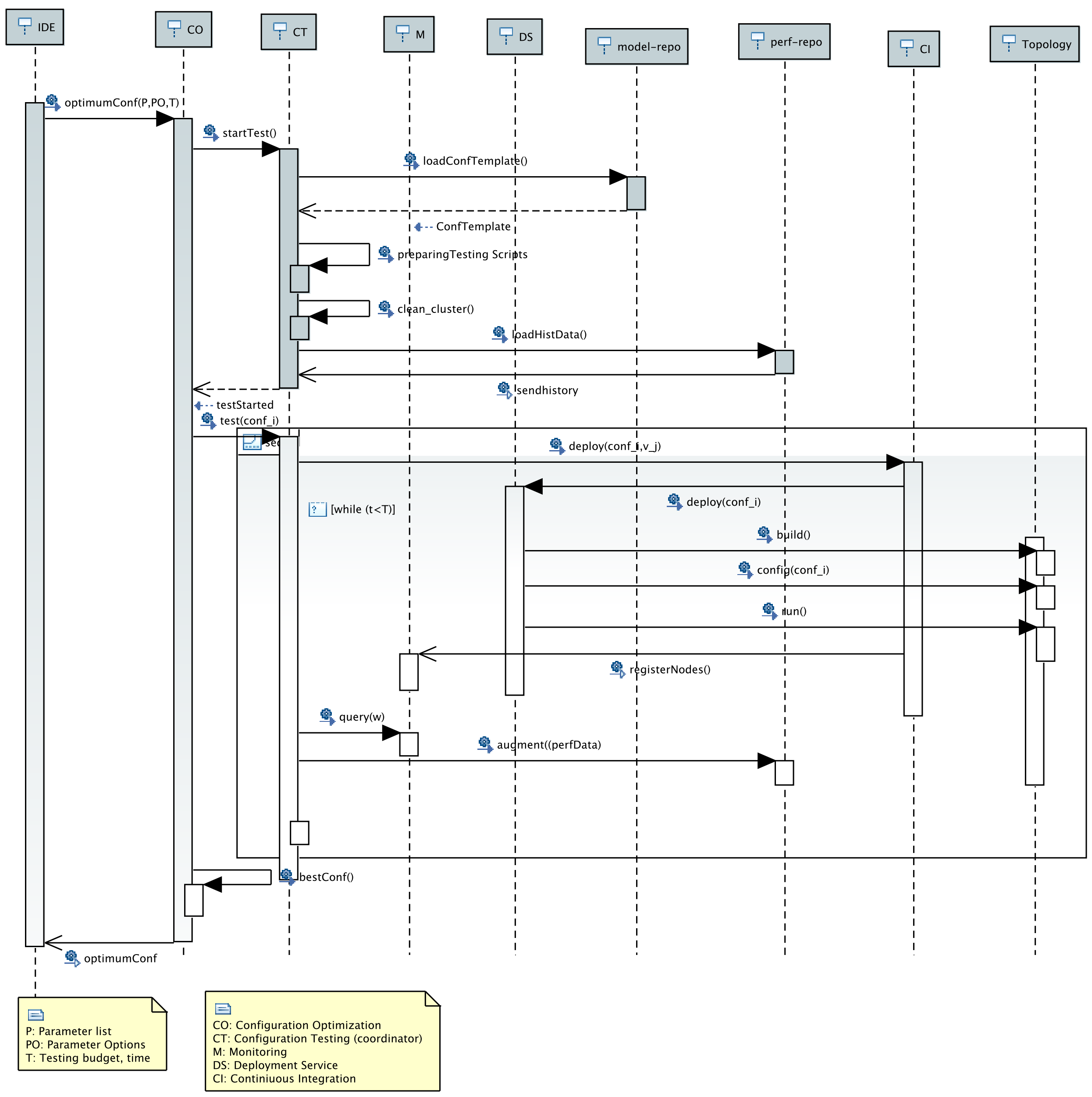
Actor 3:	N/A
Actor 4:	N/A
Flow of Events:	1. DEVELOPER provides the model, fixed parameters and free parameters as an input to DEPLOYMENT_TOOLS 2. DEPLOYMENT_TOOLS provide recommended values for the free parameters, optionally quantified with the quality criteria (reliability, efficiency, safety) 3. DEVELOPER selects from the recommended values to fix all of the parameters
Pre-conditions:	1. Model of the application (WP2) 2. Free/fixed parameters in the model (WP2) 3. Output of OPTIMIZATION_TOOLS proposing additional fixed parameters (WP3)
Post-conditions:	1. Deployment configuration with parameters set to optimal and recommended values
Exceptions:	OPTIMIZATION_TOOLS and DEPLOYMENT_TOOLS help assign a complimentary set of parameter values (e.g., number of Hadoop mappers and reducers)
Data Exchanges:	

A.12.1 Description of interactions

1- The tester provides a list of configuration parameters and potential (exhaustive) set of configuration options for each parameter. The tester also provides the maximum number of experiments for which she has the budget for.

2- The tester then starts the tool via IDE.

3- The tool then starts the test by retrieving the configuration template from model repository
4- Prepares the testing scripts and the testbed and load the historical data from data repository.
5- The configuration optimization tool also performs the initial experiments by doing a small DoE design to provide some initial data for initial model fitting.
6- The tool then sequentially performs the experiments and after the budget is finished it gives the optimum configuration as well as the internal machine learning model for performance predictions in use cases for example, A/B testing or other scenarios as mentioned above. For doing so it performs the following steps:
a. The configuration optimization tool deploys the configuration by specific parameter settings by using the DS tool.
b. The tool builds and runs the topology on the testbed.
c. The tool queries the monitoring and augments the experimental data to the data repository.
d. The tool performs the model refitting on the updated historical data and reason where to test next using the model prediction of the good locations (good location means low estimates of latency or high estimates of throughput using the internal machine learning model).



A.12.1 Data flows

As it is shown in the sequence diagram, two major entities are involved in quality testing tool: (i) the configuration file, (ii) the performance data. The configuration template is retrieved by the tool through model repository in a YAML file. The appropriate configuration is then set in the template by appropriate values. In order to perform model fitting, tool requires to retrieve the performance data and augment new points in the repository. These performance data serve as the main ingredient for reasoning where to test next in the tool. Also further internal entities are used in the model for storing the historical configurations and also storing the machine learning model and its estimates.

References

[1] Papyrus modeling environment tool. https://eclipse.org/papyrus/

[2] ECore EAnnotation http://wiki.eclipse.org/STEM_Model_Generator/EAnnotations

[3] Modelling Software KIT (MOSKitt) https://www.prodevelop.es/en/products/MOSKitt

[4] Jenkins https://jenkins-ci.org/

[5] Eclipse Modeling Framework (EMF) https://eclipse.org/modeling/emf/

[6] Elasticsearch https://www.elastic.co/products/elasticsearch

[7] Kibana https://www.elastic.co/products/kibana

[8] Kazman, R.; Klein, M. & Clements, P. (2000), 'ATAM: Method for Architecture Evaluation' (CMU/SEI-2000-TR-004) , Technical report, Carnegie Mellon University, Software Engineering Institute (SEI).

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

[10] Balalaie A, Heydarnoori A, Jamshidi P., "Microservices Migration Patterns", Technical Report, Sharif University of Technology, Technical Report No. 1, TRSUT-CE-ASE-2015-01 2015. [Available Online at http://arminb. me/microservices/report.pdf]

[11] Balalaie, A., Heydarnoori, A. and Jamshidi, P., "Migrating to cloud-native architectures using microservices: An experience report," in In Proceedings of the 1st International Workshop on Cloud Adoption and Migration, September 2015.

[12] Brunnert, A., et al., "Performance-oriented DevOps: A Research Agenda", SPEC Research Group --- DevOps Performance Working Group, Standard Performance Evaluation Corporation (SPEC), SPEC-RG-2015-01 (2015).