Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

Companion Document 1

Deliverable 1.4

Table of contents

DICE COMPONENTS & TOOLS UPDATE **4**

1. IDE **4**

Stereotyping a UML diagram with the DICE profile to obtain a Platform- Independent Model (PIM) 4

a) Description of interactions 4

b) Sequence diagrams 5

c) Data flows 5

Analysis, simulation, verification and feedback until obtaining a deployment model 5

a) Description of interactions 6

b) Sequence diagrams 6

c) Data flows 6

2. SIMULATION TOOL **7**

a) Description of interactions 7

b) Sequence diagrams 8

c) Data flows 8

3. OPTIMIZATION **9**

a) Description of interactions 9

b) Sequence diagrams 10

c) Data flows 10

4. VERIFICATION **12**

a) Description of interactions 12

b) Sequence diagrams 12

c) Data flows 13

5. MONITORING **14**

ID: UC4.1 Title: Monitoring a Big Data framework (Scenario) [Combination of UC 4.1.1 to UC 4.1.3] 14

a) ID: UC 4.1.1. Title: Metrics Specification 14

Description 14

Data Flow 14

b) ID: UC 4.1.2 Title: Monitoring tools registration 14

Description 14

Data Flow 15

c) ID: UC 4.1.3 Title: Monitoring Data storage (Start ES and LS) 15

Description 15

Data Flow 16

d) ID: UC 4.2. Title: Data Warehouse Query 16

Description 16

Data Flow 17

6. ENHANCEMENT TOOL **18**

a) Description of interactions 18

b) Sequence diagrams 18

c) Data flows 18

7. TRACE CHECKING **19**

a) Description of interactions 19

b) Sequence diagrams 19

c) Data flows 19

d) Next scenario 20

8. ANOMALY DETECTION 21

ID: UC 4.5 Title: Anomaly Detection Model Training 21
a) Description 21
b) Data Flow 21
ID: UC 4.6. Title: Offline Anomaly Detection 21
a) Description 21
b) Data Flow 22

9. DELIVERY TOOL 23

Application deployment 23
a) Description of interactions 23
b) Sequence diagrams 23
c) Data flows 23
Continuous integration 24
a) Description of interactions 24
b) Sequence diagram 25
c) Data flows 25
Version tagging in Deployment Tool 25
a) Description of interactions 26
b) Sequence diagrams 26
c) Data flows 26

10. QUALITY TESTING 27

a) Description of interactions 27
b) Sequence diagrams 28
c) Data flows 28

11. FAULT INJECTION 29

ID R5.14.2 29
a) Description of interactions 29
b) Sequence diagrams 29
c) Data flows 29
ID R5.30 30
a) Description of interactions 30
b) Sequence diagrams 30
c) Data flows 30

12. CONFIGURATION OPTIMISATION 31

a) Description of interactions 31
b) Sequence diagram 32
c) Data flows 32

DICE Components & Tools Update

1. IDE

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

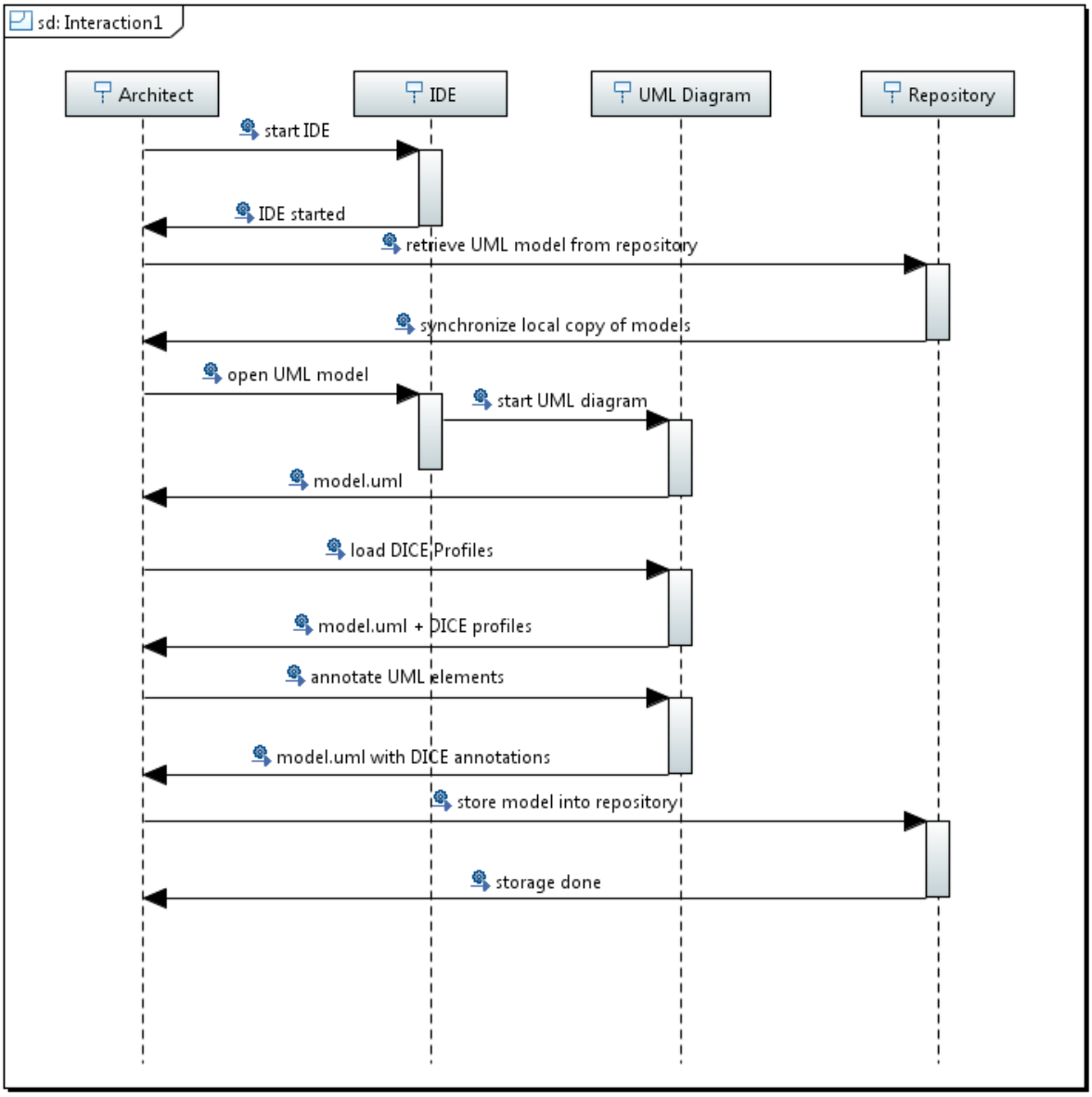
ID	UC1.1
Title	Stereotyping a UML diagram with the DICE profile to obtain a Platform-Independent Model
Priority	Required
Actors	Architect, IDE
Flow events	A technical person capable of designing and modelling a data intensive application models the Platform-Independent UML Model stereotyped with the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

a) 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.

b) Sequence diagrams



c) 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 needs 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.

Analysis, simulation, verification and feedback until obtaining a deployment model

ID	UC1.2
Title	Analysis, simulation, verification and feedback 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

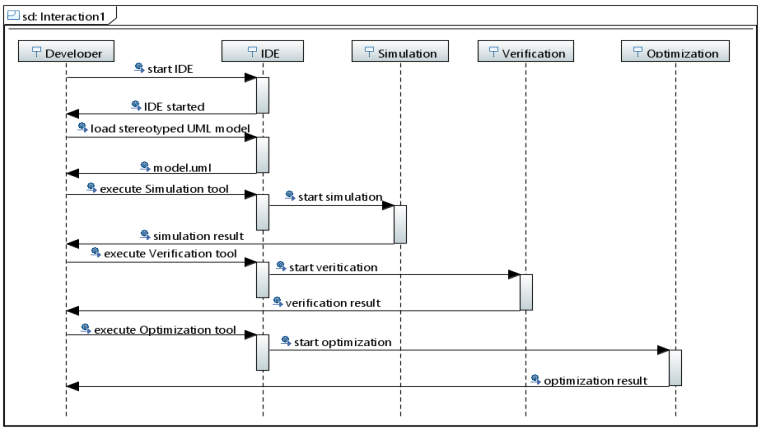
a) Description of interactions

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.

b) Sequence diagrams



c) 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.

2. Simulation tool

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

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

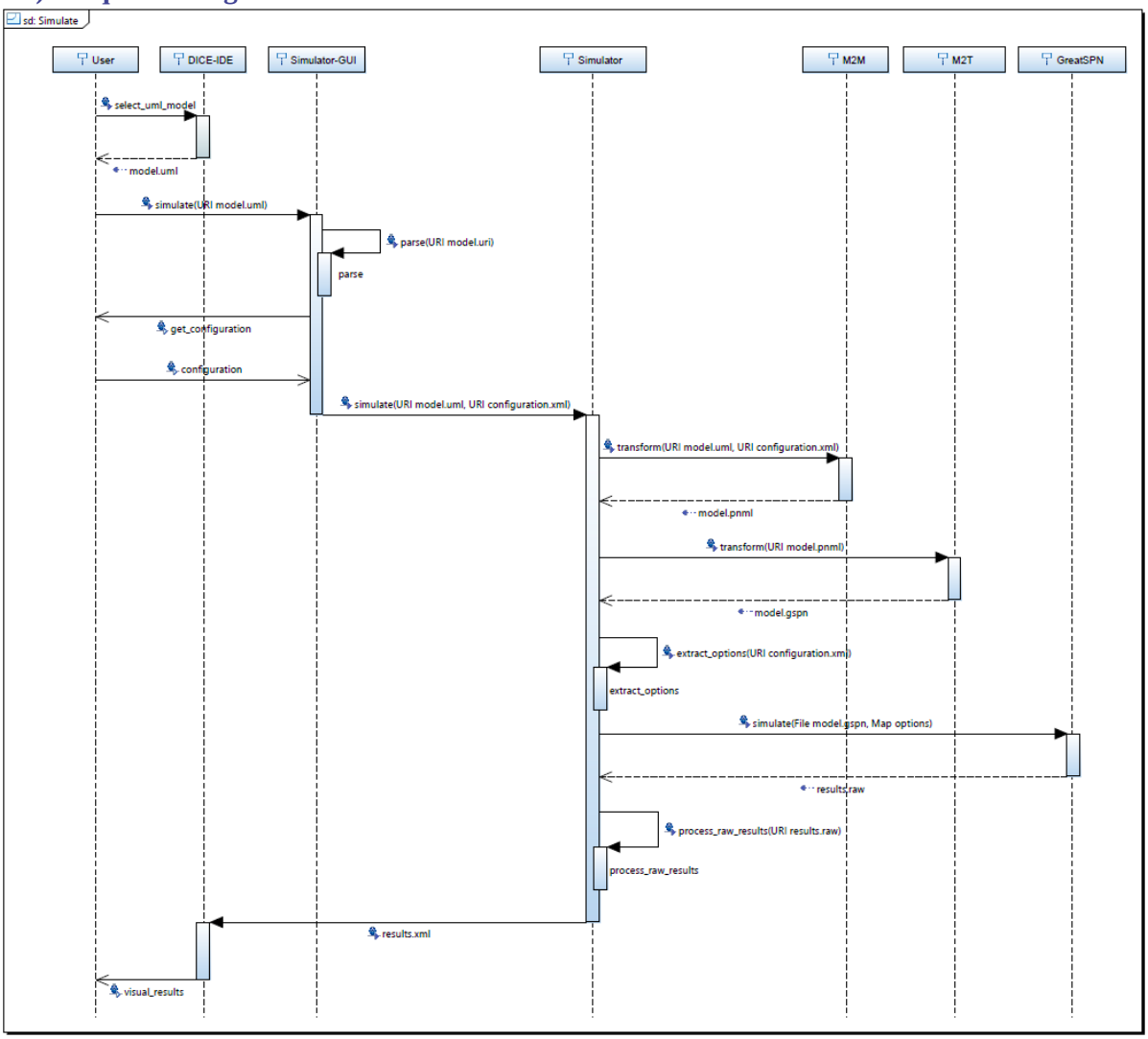
a) Description of interactions

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:

- 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).
- 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.

b) Sequence diagrams



c) 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 (e.g., information about the concrete scenario that s/he is interested in simulate among the possible multiple defined scenarios in the model, quality metrics of interest for this concrete simulation such as response time, throughput, resource utilization or reliability). 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.

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 DTSM DICE annotated UML model with performance constraints.
Priority	Required
Actors	ARCHITECT
Pre-conditions	There exists a partially specified DTSM UML annotated model including the DIA activity diagram and deployment model.
Post-conditions	The ARCHITECT gets a fully specified DDSM model minimizing the deployment cost and fulfilling QoS constraints specified as input.

a) Description of interactions

The use case UC3.3 specifies that, from an existing DTSM UML annotated model with several candidate deployments (pre-condition), the ARCHITECT gets a fully specified DDSM model minimizing the deployment cost and fulfilling QoS constraints (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 optimize it by means of the Optimization tool. The tool requires DTSM DICE activity diagram and deployment models.

The user feeds the deployment model, sporting several candidate deployments, as input into the optimization tool along with some other relevant inputs that are used to control the behavior of the tool. Some particular pieces of information as simulation accuracy, local search maximum number of iterations, 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 obtaining an initial guess via a mathematical programming model. This mixed integer nonlinear problem (MINLP) is based on approximate formulas, derived with machine learning techniques, for the prediction of DIAs execution time. Overall, the problem is meant to provide the minimum cost configuration able to meet the user defined QoS constraints, choosing among the candidate deployments. Decoupling the choice of the deployment and the cost optimization, it is possible to deduce a closed form solution for the partial minimization problem, thus allowing for an exhaustive search of all the candidate deployments that quickly yields the initial solution.

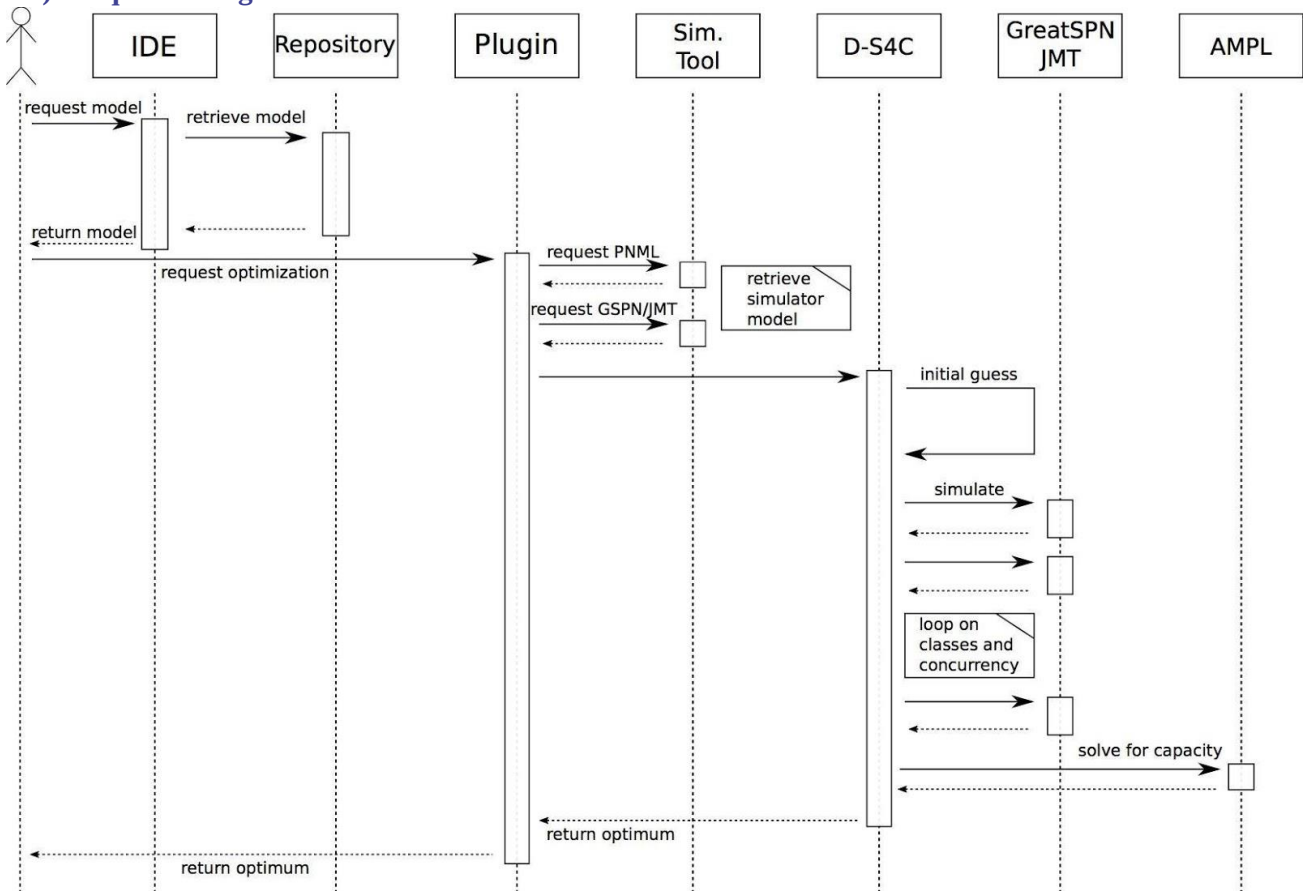
The so obtained initial solution is afterwards used within the local search based optimization process representing the core of the tool. To this end the solution undergoes a set of transformations that are applied iteratively with the aim of progressively reducing the deployment cost guaranteeing at the same time the fulfillment of the constraints. The deployment models generated during this phase are turned into Petri Nets or Queueing networks by the Simulation tool plugin, which is in charge of performing suitable model-to-model transformations. Simulations are performed by JMT or GreatSPN run directly from the optimization tool. The outcome of these simulations is a set of performance metrics 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.

In case users set up capacity constraints, possibly also enabling job rejections, the optimizer performs the search technique class-wise and for all the relevant concurrency levels, in order to feed such data into an integer linear programming (ILP) model that handles the admission control. A third party mathematical

programming solver is used to solve the problem and retrieve the configuration minimizing the sum of costs and penalties, while also meeting the capacity constraints.

At the end of this procedure the best deployment model obtained (complete DDSM) is presented to the user along with its related performance metrics.

b) Sequence diagrams



c) Data flows

We have modelled the interactions among the **Optimization** tool component as depicted in the previous figure. 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** plugin providing such a model. 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** plugin.
3. The plugin interacts with the **Simulation** tool to retrieve a representation suitable for external solvers.
4. The plugin then sends data to the **Optimization** tool to launch the procedure.
5. The **Optimization** tool obtains an initial guess of the optimal configuration by comparing possible deployments via a closed form solution to a preliminary mathematical programming (MP) problem.
6. Via external simulators, the **Optimization** tool determines the optimal amount of resources per class and per concurrency level.
7. If the overall problem has capacity constraints, the **Optimization** tool relies on AMPL to solve a proper MP problem.

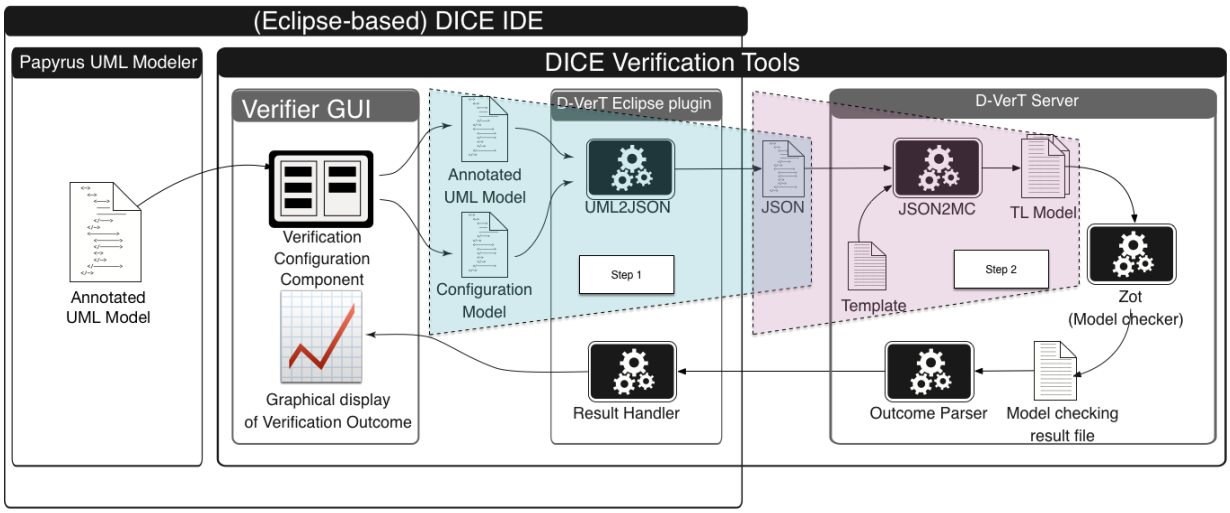
The outcome of the **Optimization** (the cheapest feasible deployment identified) and its related metrics are reported to the user.

4. Verification

a) Description of interactions

The user performs verification on the current model loaded in the IDE. The user might also select an already defined model from the repository; in the last case, the model is first loaded and then showed in the IDE.

The following figure outlines the architecture of the verification tool and shows the workflow elaborating the UML model of the application undergoing verification.



The verification is performed on annotated DTSM models which already contain all the information required to perform the analysis. DTSM models are defined by the user in the DICE IDE as Papyrus diagrams. In the IDE, the user provides all the values of the parameters needed to carry out verification.

The annotated model can then be verified with the D-VerT, the verification tool implemented in the DICE framework. D-VerT is a client-server application. The client resides in the DICE IDE, being implemented as an Eclipse plugin; the server is a REST service running on a server.

To carry out verification, the user selects a (safety) property to be checked in the verifier GUI, possibly using templates. The property is compliant with the class of properties specified at DPIM level.

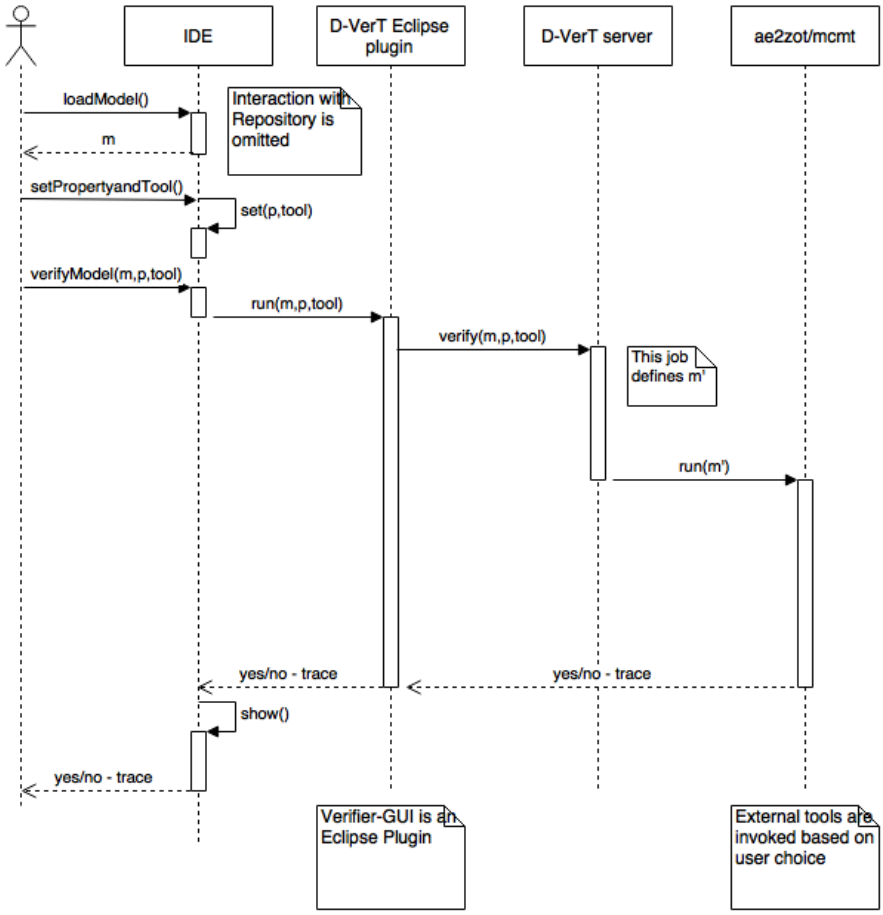
Once the property is selected, the D-VerT Eclipse plugin submits the verification request to the D-VerT server (i.e., the Verification engine) which runs the verification task. The request is sent by means of a REST API call and consists of a JSON file which includes:

1. a description of the DTSM model that is used by the verification engine to instantiate a temporal logic formula modeling the application and
2. a set of parameters that specify the configuration values (such as, for instance, the plugin to use to carry out verification) characterizing the problem instance that is then solved by the model checker (the Zot tool).

The Verification engine is a service available in an external server. Based on specific templates, JSON2MC converts the JSON descriptor into a formal model that is suitable for verification. i.e., a temporal logic formula which is processed by the model checker (the Zot tool). A template, in this case, is a “schema” of formula that is instantiated with the parameters specified in the JSON descriptor.

The outcome of the solver is sent to D-VerT Eclipse plugin by means of a request to the server that the user performs in the IDE to “pull” the result. If the result is available, the D-VerT Eclipse plugin elaborates it in order to show a graphical representation of the anomalies in the Verifier GUI. If the property is violated, the IDE presents the trace of the system that violates it.

b) Sequence diagrams



c) 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.

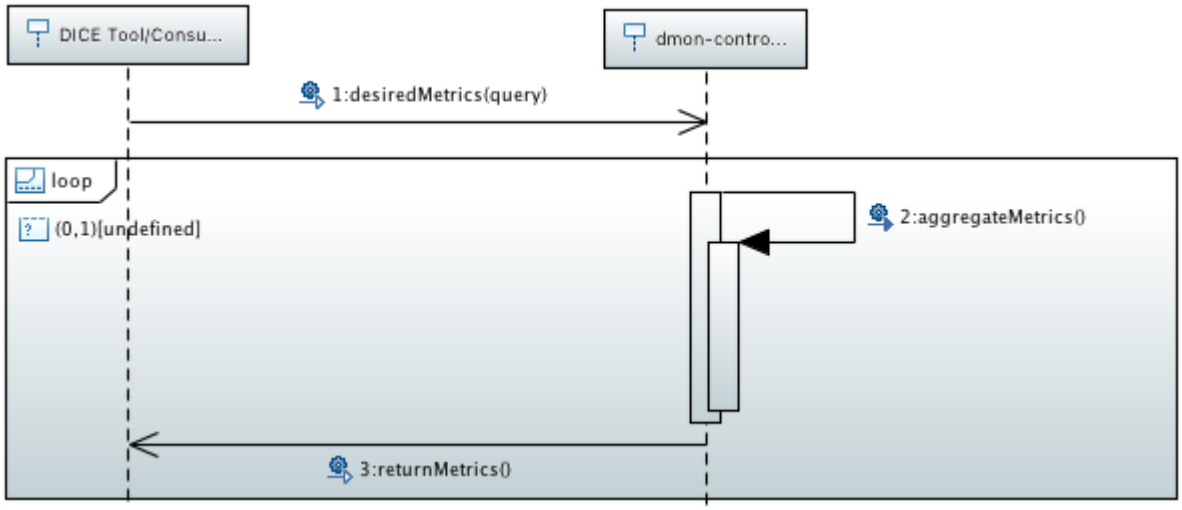
5. Monitoring

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

a) ID: UC 4.1.1. Title: Metrics Specification

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 important to note that the query should also contain the unique application tag. 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.



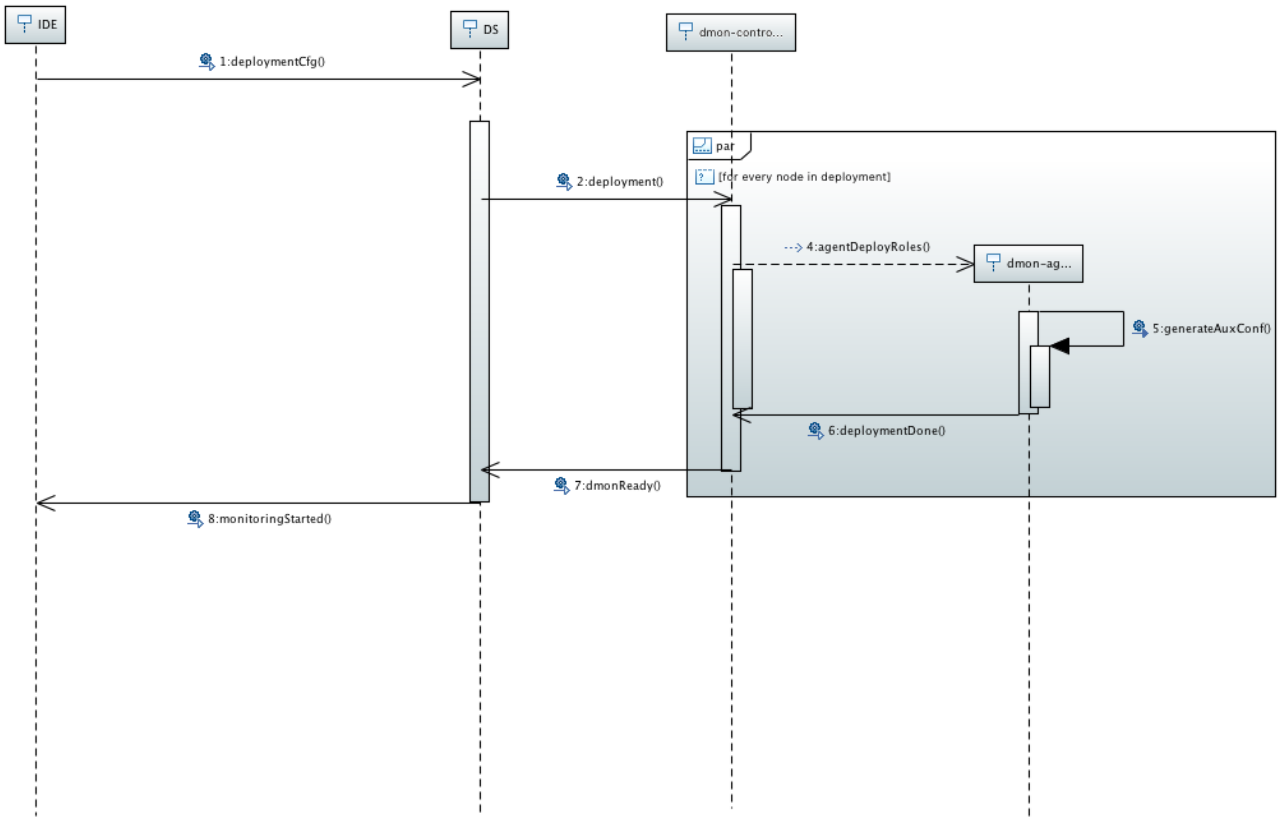
Data Flow

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.

b) ID: UC 4.1.2 Title: Monitoring tools registration

Description

The current deployment specification is sent from the IDE 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.



Data Flow

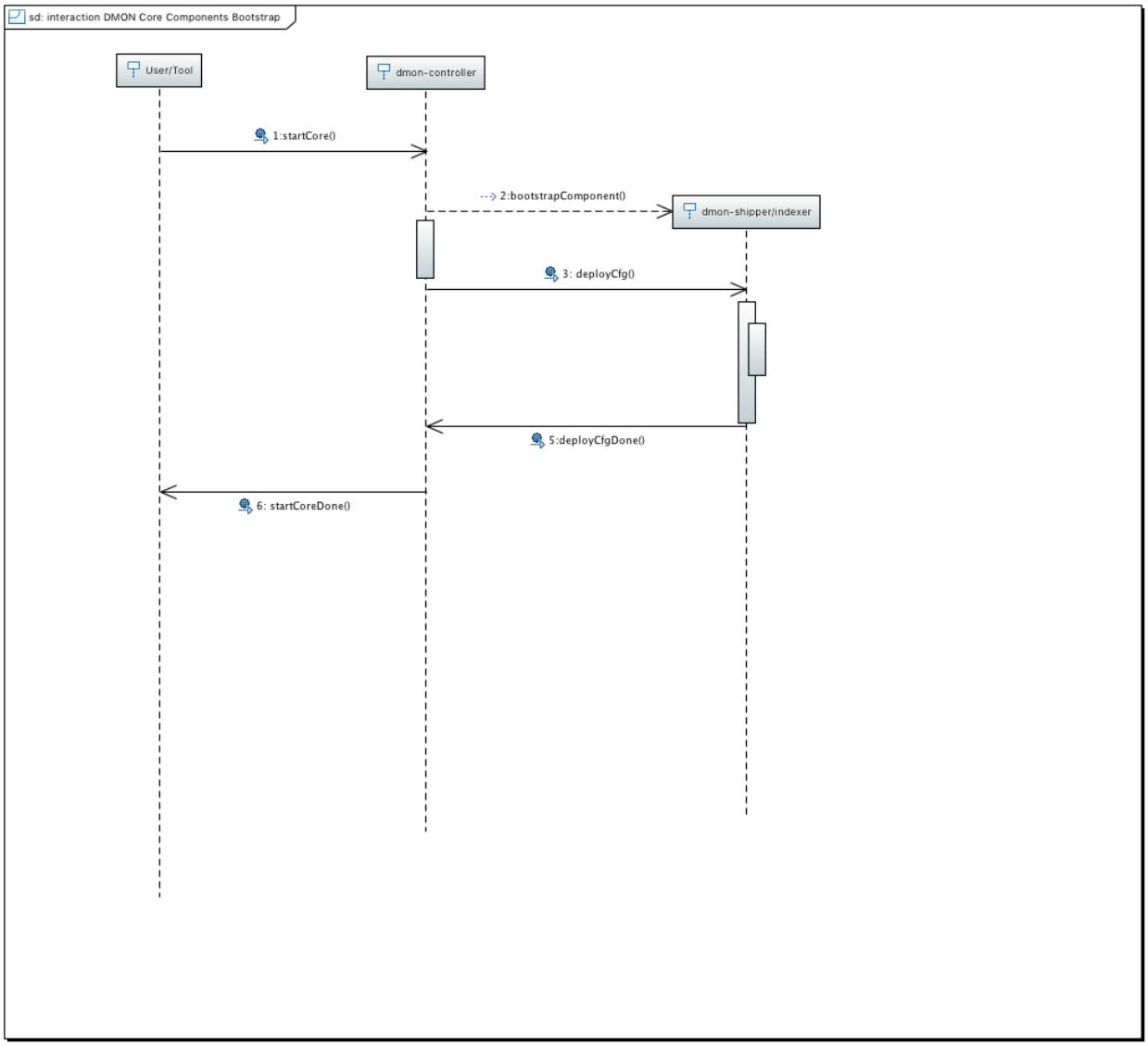
Data is represented only by the request payloads (json).

c) ID: UC 4.1.3 Title: Monitoring Data storage (Start ES and LS)

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 elastic search. 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.



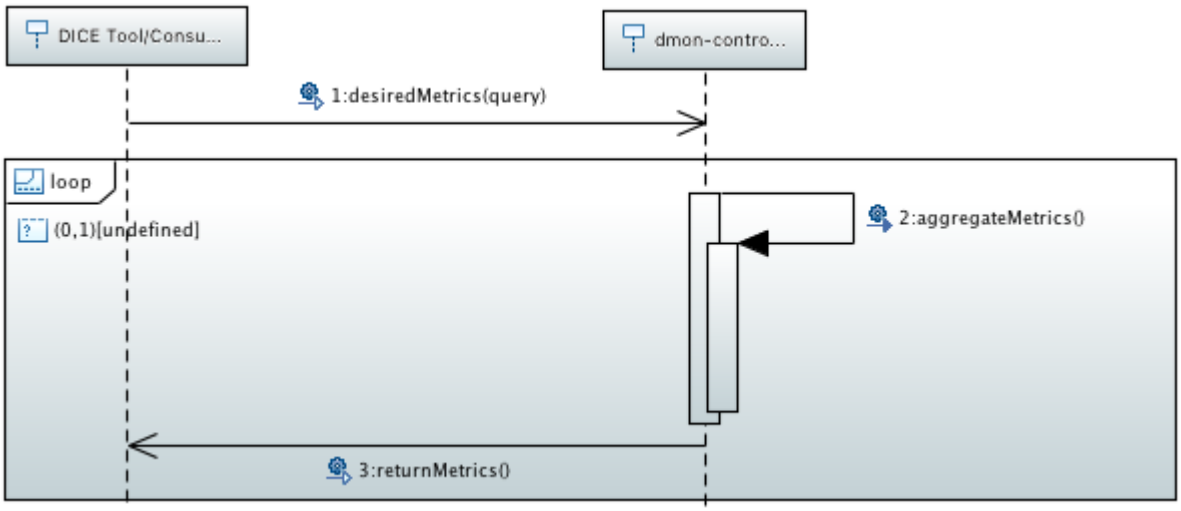
Data Flow

The only data is the request payloads (json).

d) ID: UC 4.2. Title: Data Warehouse Query

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, the query also must contain the application tag. 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.



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.

6. Enhancement Tool

a) Description of interactions

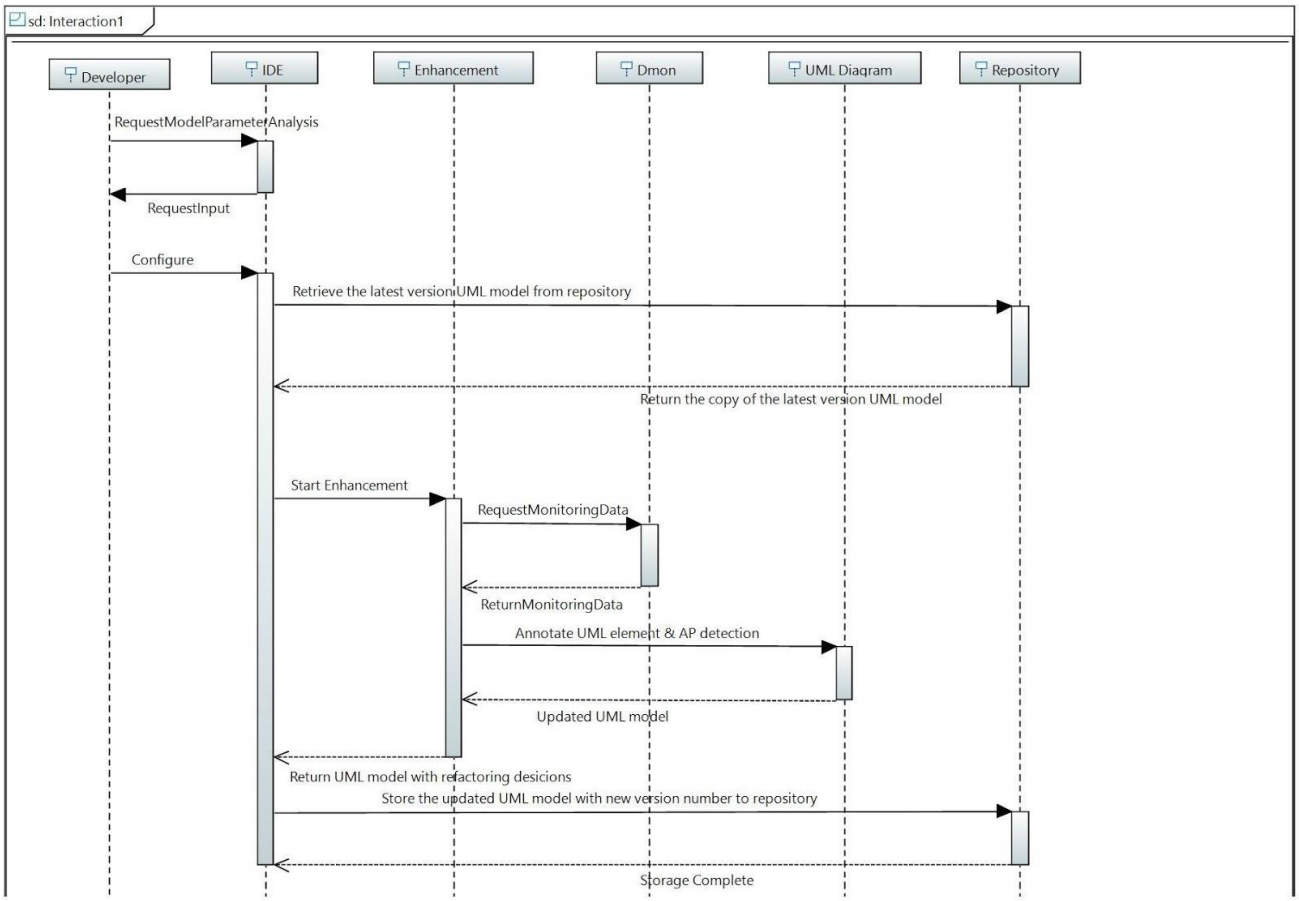
The user activates the enhancement analysis for model parameter analysis and inputs the parameters for the Enhancement tool through IDE. Then the IDE triggers the start of the Enhancement tool.

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.

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.

b) Sequence diagrams



c) Data flows

The major data exchange happens when the enhancement tool updates the models. The DICE-FG will query monitoring data from the Monitoring Platform and uses these data to analyse the parameters of the performance models. Then, the updated model will be used as input of DICE-APR. DICE-APR will refactoring the model if anti-patterns are detected.

7. Trace checking

a) Description of interactions

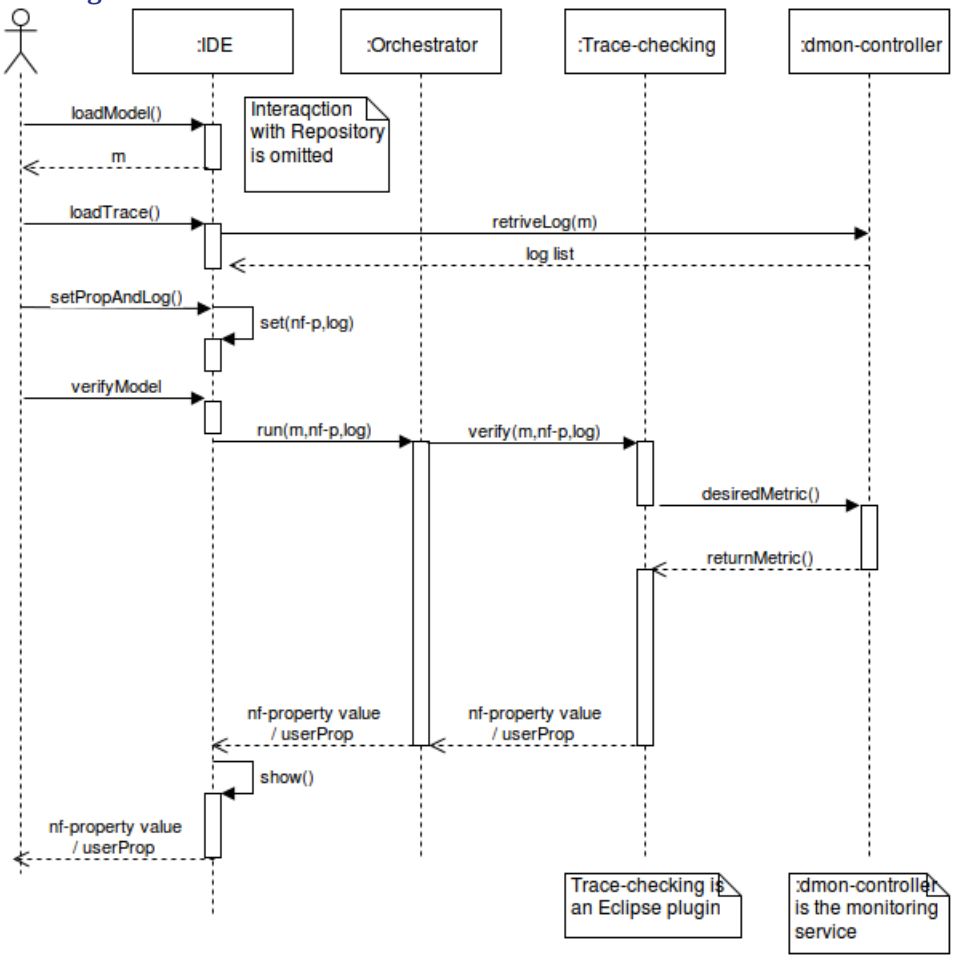
The user activates trace checking on the current model loaded in the IDE or selects the model from the repository; in the last case, the model is first loaded and then showed in the IDE.

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.

b) Sequence diagrams



c) 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.

d) Next scenario

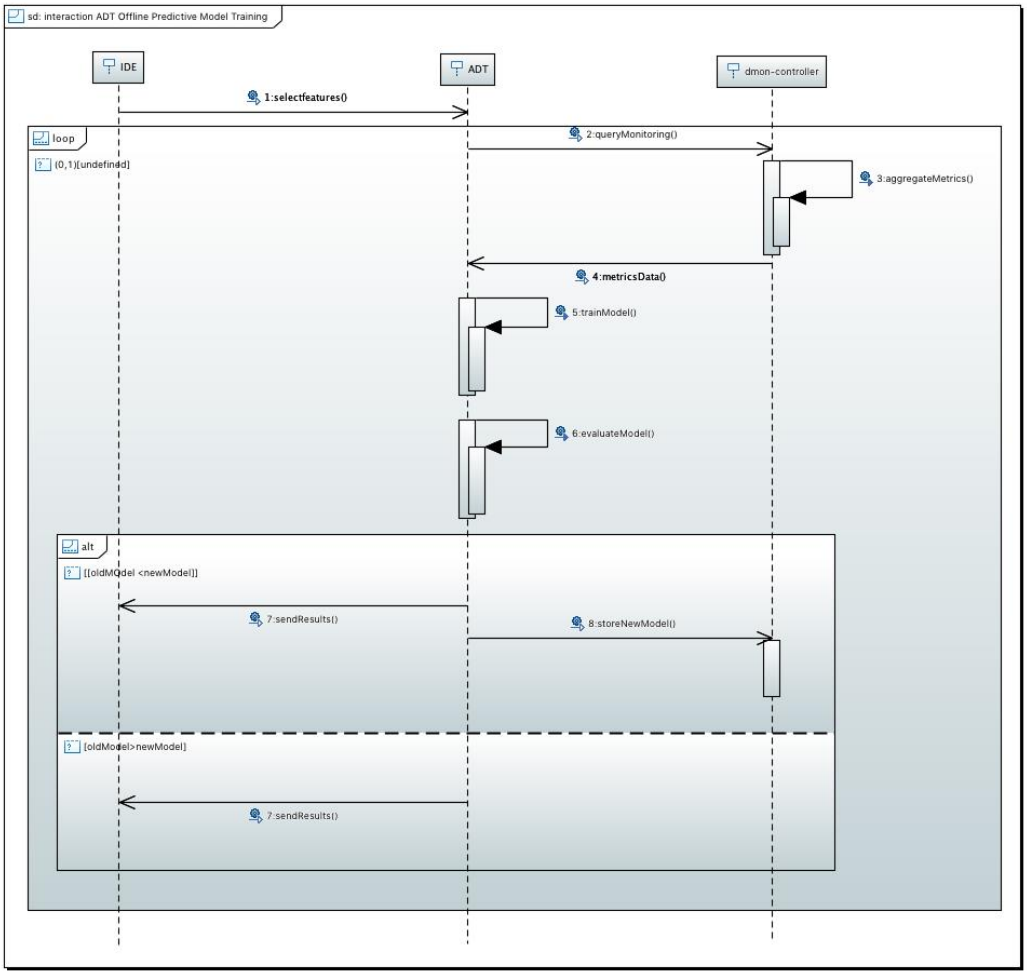
If the non-functional properties have specific relation (order) with respect to the values established at design time then the tool reports the properties, defined at DPIM level that might be violated.

8. Anomaly detection

ID: UC 4.5 Title: Anomaly Detection Model Training

a) 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. This task is under development.



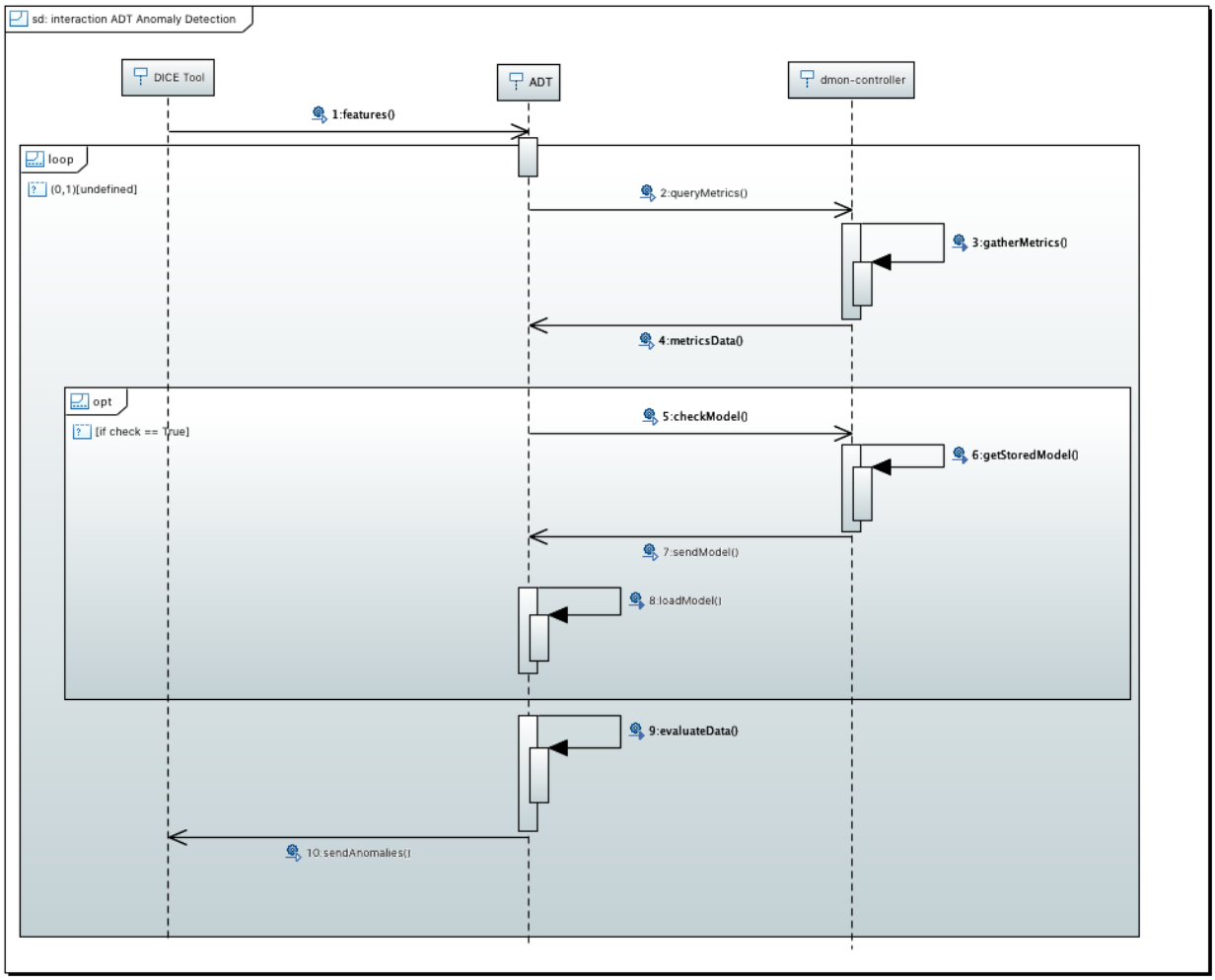
b) Data Flow

Data is consumed from the DICE Monitoring platform.

ID: UC 4.6. Title: Offline Anomaly Detection

a) 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.



b) 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.

9. Delivery tool

Delivery tool consists of the DICE deployment service and the DICE Continuous Integration. Its aim is to enable one-time deployment of user's application (the deployment service) or to perform deployments continuously, normally followed by running another tool such as Quality Testing Tool or Configuration Optimization (the Continuous Integration).

Application deployment

This scenario enables that a user or another client that has a blueprint of the application achieves that the application described in the blueprint becomes live in the testbed.

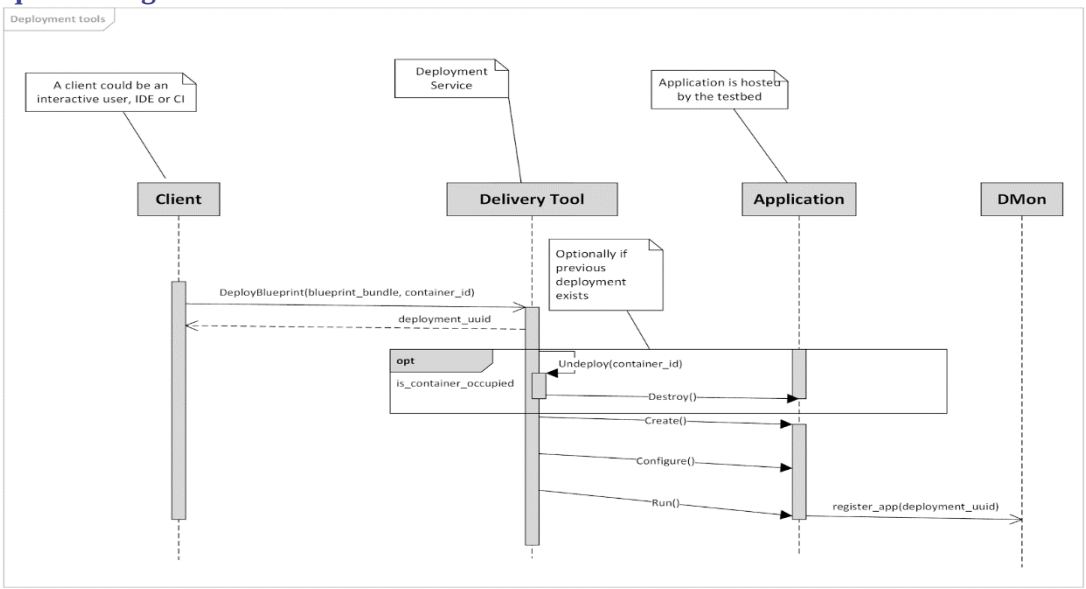
a) Description of interactions

The DICE deployment service can be used stand-alone without the Continuous Integration by using IDE, where the user can deploy applications using a TOSCA blueprint derived from the DDSM. The Client in this sequence can also be another DICE tool such as the Configuration Optimization.

The actors (normally programmers) use the IDE to model the application at the 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 in their IDE. Then, they can either commit the project code (model, application's coder) in the Repository (SVN, GIT, etc.)., or they deploy the application directly. The Delivery Tool receives a bundle containing the TOSCA blueprint of the application and any other artifacts needed in the deployment (blueprint_bundle). The user needs to specify a logical deployment container's ID (container_id). The Delivery Tool returns the deployment's unique identification, deployment_uuid.

The deployment service part of the Delivery Tool initiates the deployment and configuration phase. It first destroys any existing deployment of the application in the logical deployment container. Then it proceeds by first creating the environment in the test bed, which provisions 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.

b) Sequence diagrams



c) Data flows

In this scenario, the user produces the application code. The user employs DICER to perform the model-to-text transformation, which produces an OASIS TOSCA blueprint document in YAML format. The IDE bundles the blueprint with any supplemental files that are needed in the deployment (e.g., jar files, custom scripts, etc.).

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. It returns a deployment identificator, and results in a deployed application, which is also registered with the DMon.

Continuous integration

This scenario addresses the Continuous integration sequence, which is an iterative process, which loops from the programmer's model and code updates, across deployed platform services and the application in the test bed, and performing quality (non-functional) tests runs. The outcomes of each iteration are available in the Continuous Integration component of the DICE Delivery Tool, i.e., accessible through the Jenkins user interface and its web links..

a) Description of interactions

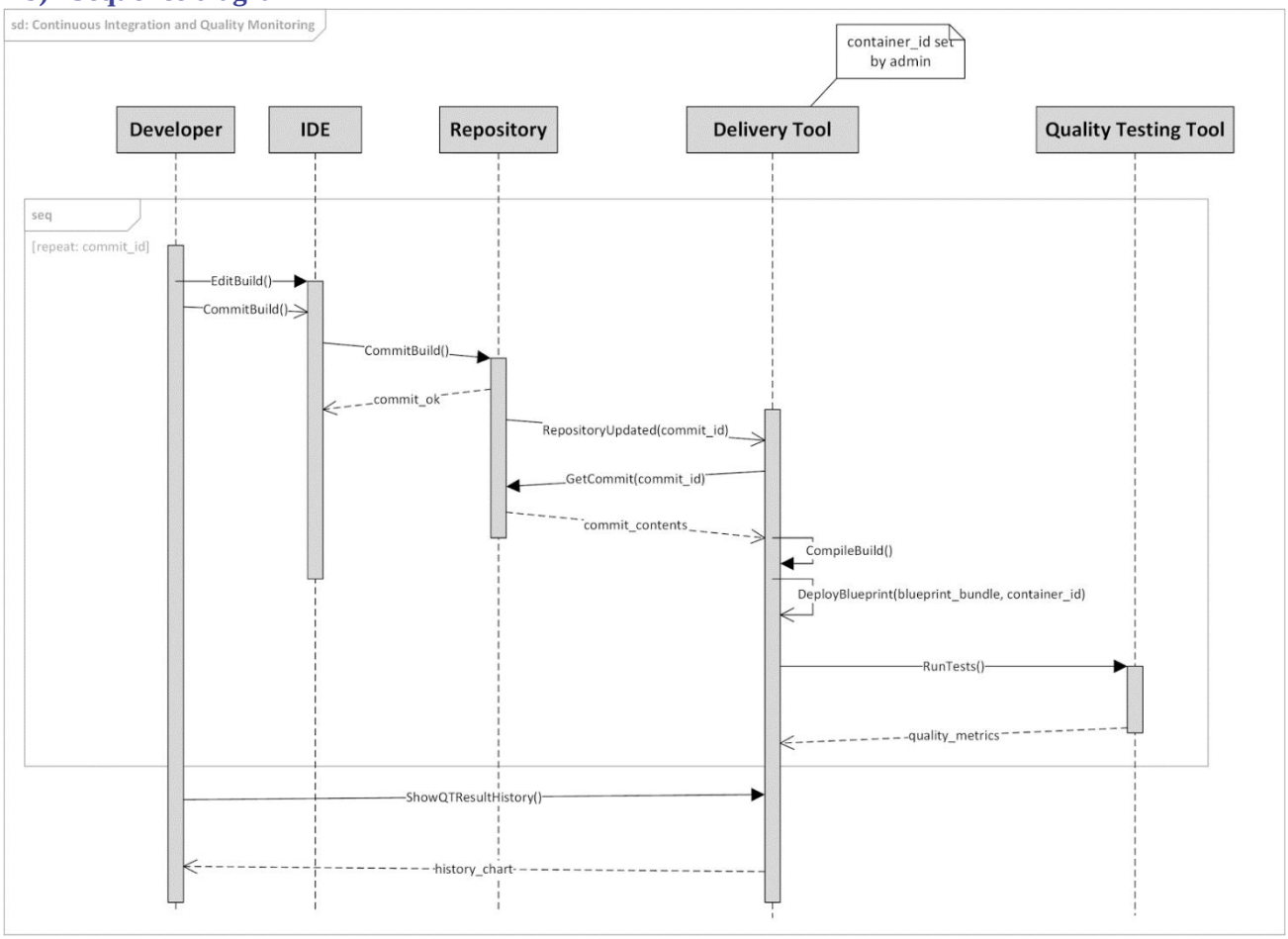
This sequence diagram shows the process at a larger context than the previous one. The developer, who edits the current build as the code and models, requests of the IDE to commit the build. IDE pushes the build to the Repository. The Continuous Integration part of the Delivery Tool receives a notification 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. This step is often followed by unit testing of the code.

The Delivery Tool then deploys the blueprint from the build using the logical deployment container that was assigned by the Administrator. 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 deploys any previous builds in the same logical deployment container.

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 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.

b) Sequence diagram



c) 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 only addition here is that the project needs to be pushed to the repository (SVN, GIT, etc.). The Delivery Tool consumes the TOSCA document and, based on the blueprint description, deploys and configures the application in the test bed.

The Delivery Tool is configured to run quality tests to a certain extent (frequent short tests, occasional longer tests). It runs 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.

Version tagging in Deployment Tool

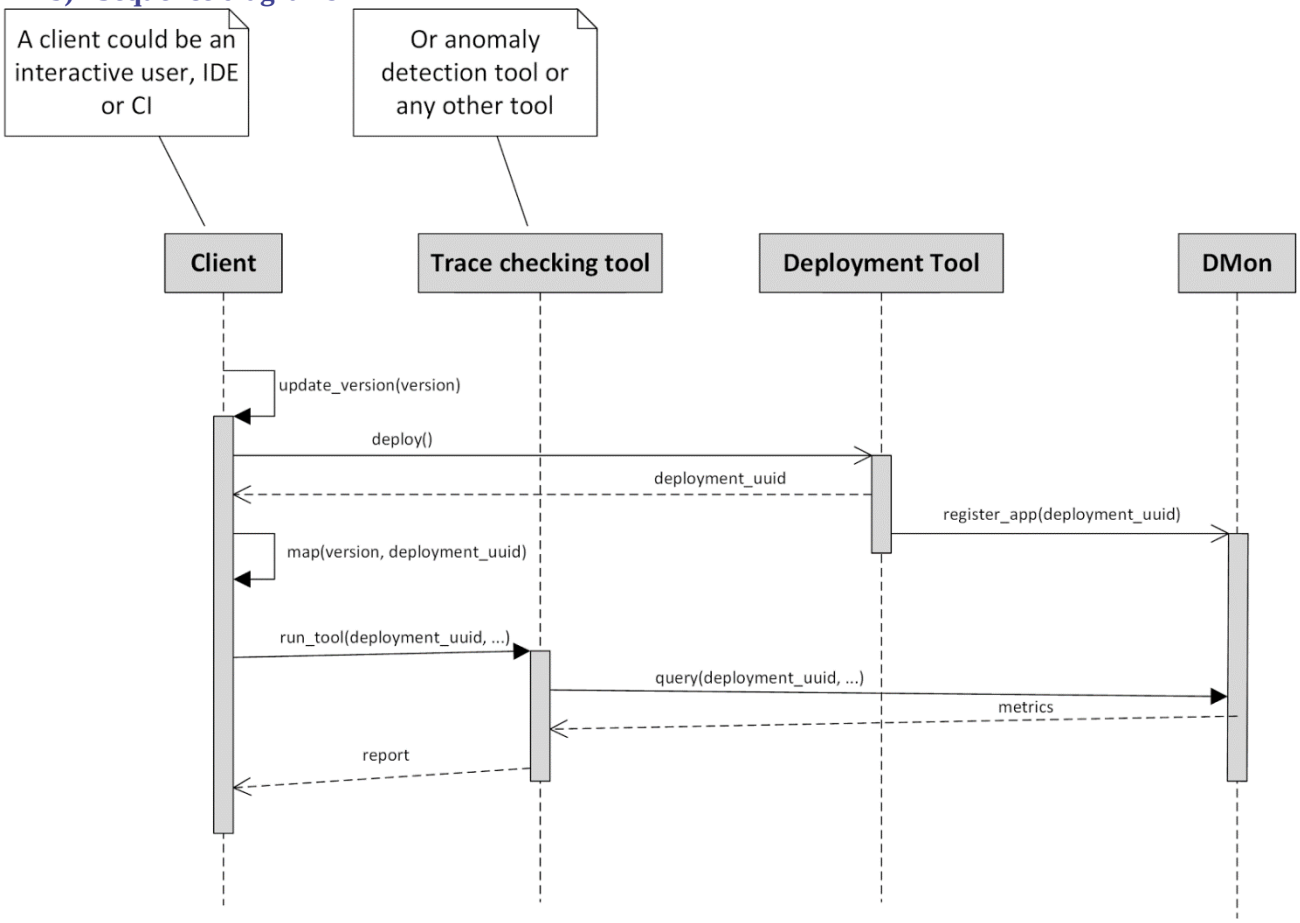
In this scenario, we need to be able to follow a specific version of an application through the DICE methodology. The basic problem we want to solve is this: a UML diagram has a certain version. When the user uses this diagram to deploy the application and do some enhancement-related analysis (filling the gaps, trace checking, anomaly detection, etc.) the tool used needs to be able to distinguish between monitored data from different versions. In these scenarios we simplified the situation such that the version of a diagram is bound to the version of the whole project (e.g., version number is complemented with a non-decreasing test execution or build number).

a) Description of interactions

The first sequence diagram starts when a developer updates the version of the build (set to variable version). It then deploys the application and gets a deployment_uuid from the deployment tool. The deployment tool uses this UUID to register the application being deployed with the DMon. The client then needs to save the mapping between version and deployment_uuid.

When the user then wants to run analysis, he/she needs to supply the deployment_uuid to the tool doing the analysis. This tool, in turn, needs to use the deployment_uuid when querying DMon in order to figure out which metrics are related to the version of the application being tested.

b) Sequence diagrams



c) Data flows

The dataflow for the scenario involves a client that internally manages the version of the user's application. It sends to the Deployment Tool the blueprint bundle and receives the deployment identifier, which it must store internally. The same client then uses this deployment identifier when running queries with the DMon.

10.Quality testing

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

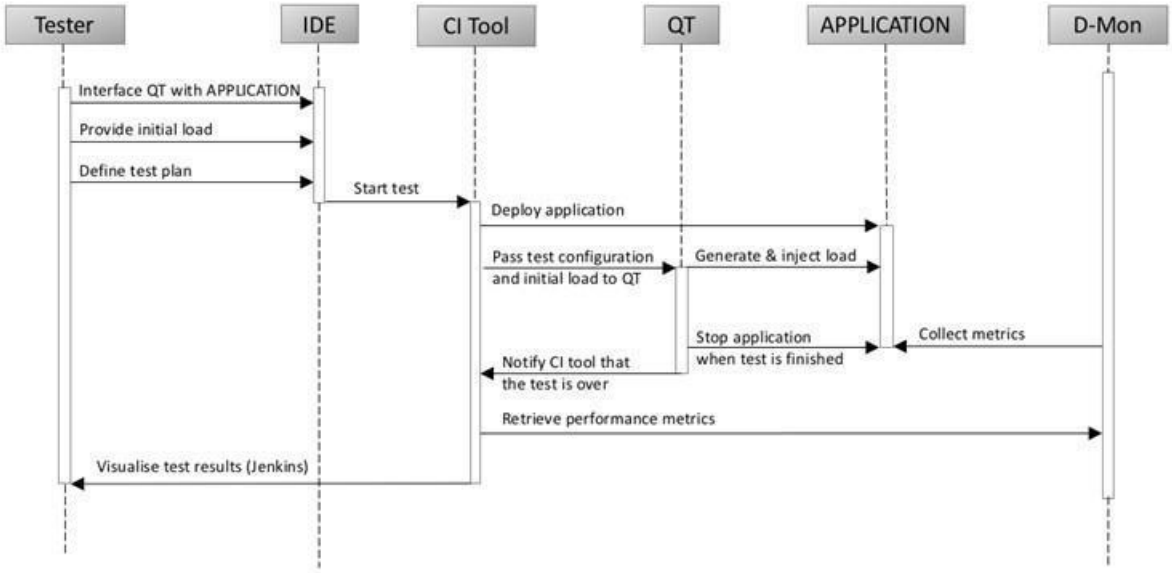
ID:	U5.10
Title:	Performing the quality testing
Task:	T5.3

Priority:	REQUIRED
Actor 1:	QTESTING_TOOLS
Actor 2:	MONITORING_TOOLS
Actor 3:	TESTBED
Actor 4:	CONTINUOUS INTEGRATION (CI) TOOL
Flow of Events:	QTESTING_TOOL starts injecting load in the APPLICATION after CI TOOL deploys APPLICATION in the test mode (QTESTING_TOOL will be bundled and deployed with the APPLICATION). The test plan is executed with parameters (test scenario) provided by the developer in DICE IDE. If requested, QTESTING_TOOLS may access TESTBED APIs to perform the test
Pre-conditions:	A quality test has been requested in some scenario Test resources and test configuration/specification have been provisioned
Post-conditions:	1. Test data has been collected by MONITORING_TOOLS
Exceptions:	N/A
Data Exchanges:	N/A

a) 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 initial load to the load generator.
2. The tester then initiates the test using the IDE.
3. The load will be injected using appropriate injection driver 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.

b) Sequence diagrams



c) 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.

11.Fault injection

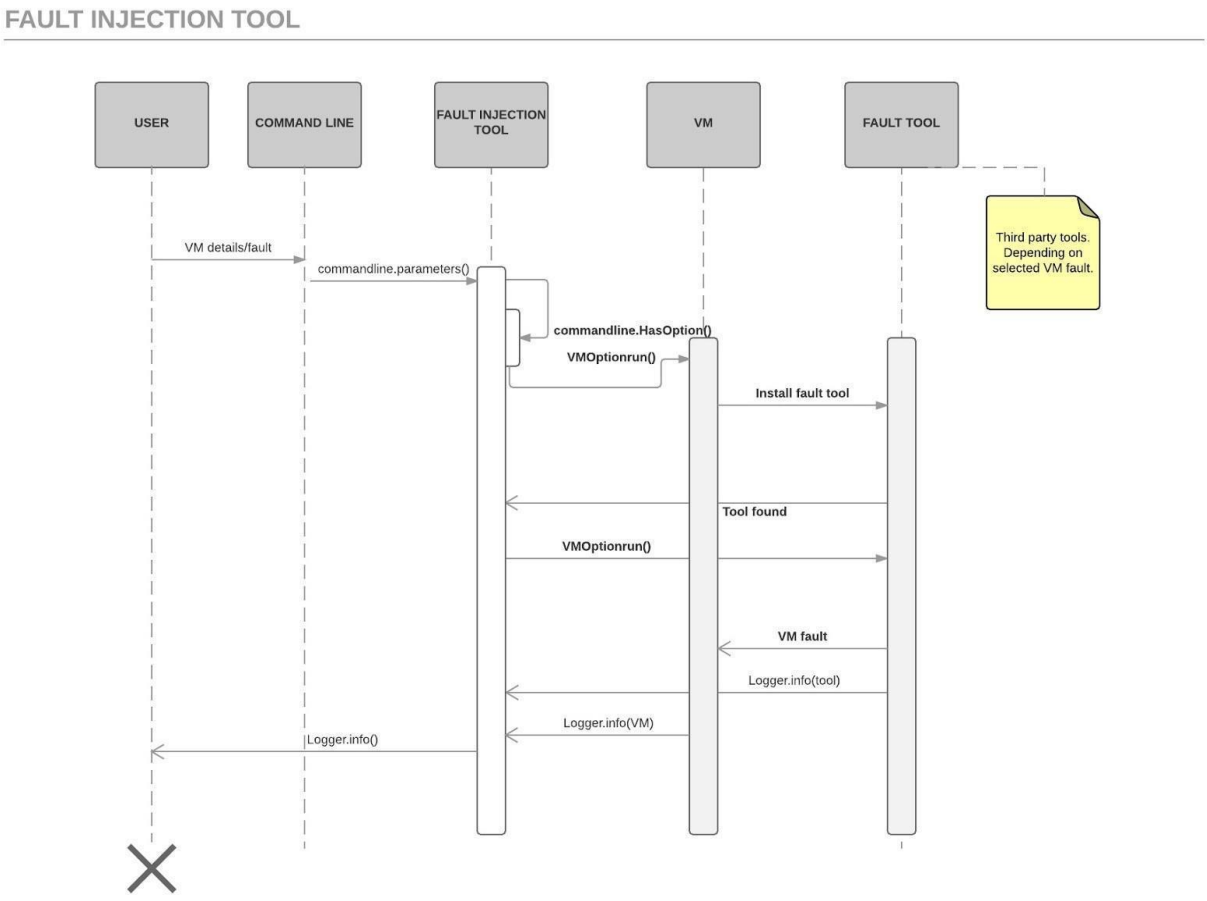
ID R5.14.2

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

a) 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.

b) Sequence diagrams



c) 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.

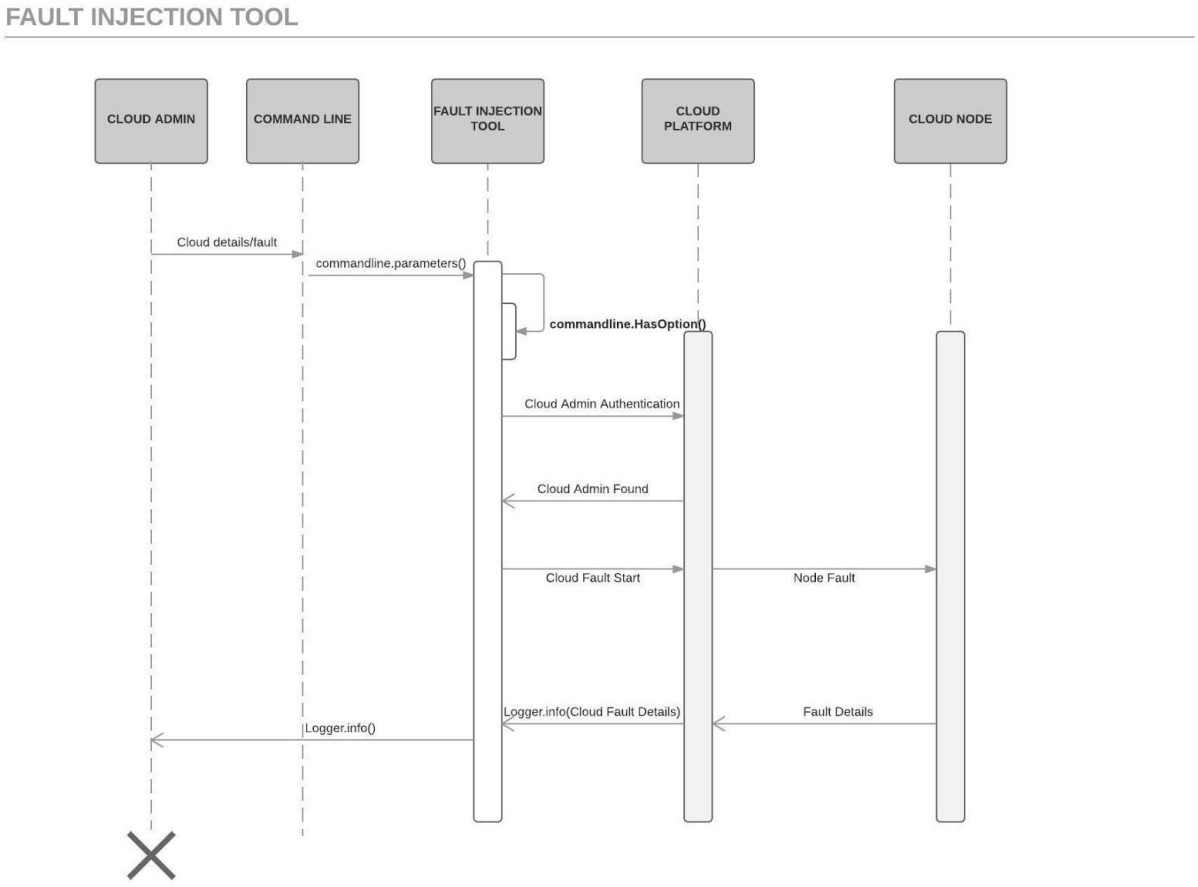
ID R5.30

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

a) 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.

b) Sequence diagrams



c) 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.

12.Configuration Optimisation

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:	DEVELOPER provides the model, fixed parameters and free parameters as an input to DEPLOYMENT_TOOLS DEPLOYMENT_TOOLS provide recommended values for the free parameters, optionally quantified with the quality criteria (reliability, efficiency, safety) DEVELOPER selects from the recommended values to fix all of the parameters
Pre-conditions:	Model of the application (WP2) Free/fixed parameters in the model (WP2) 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)

a) Description of interactions

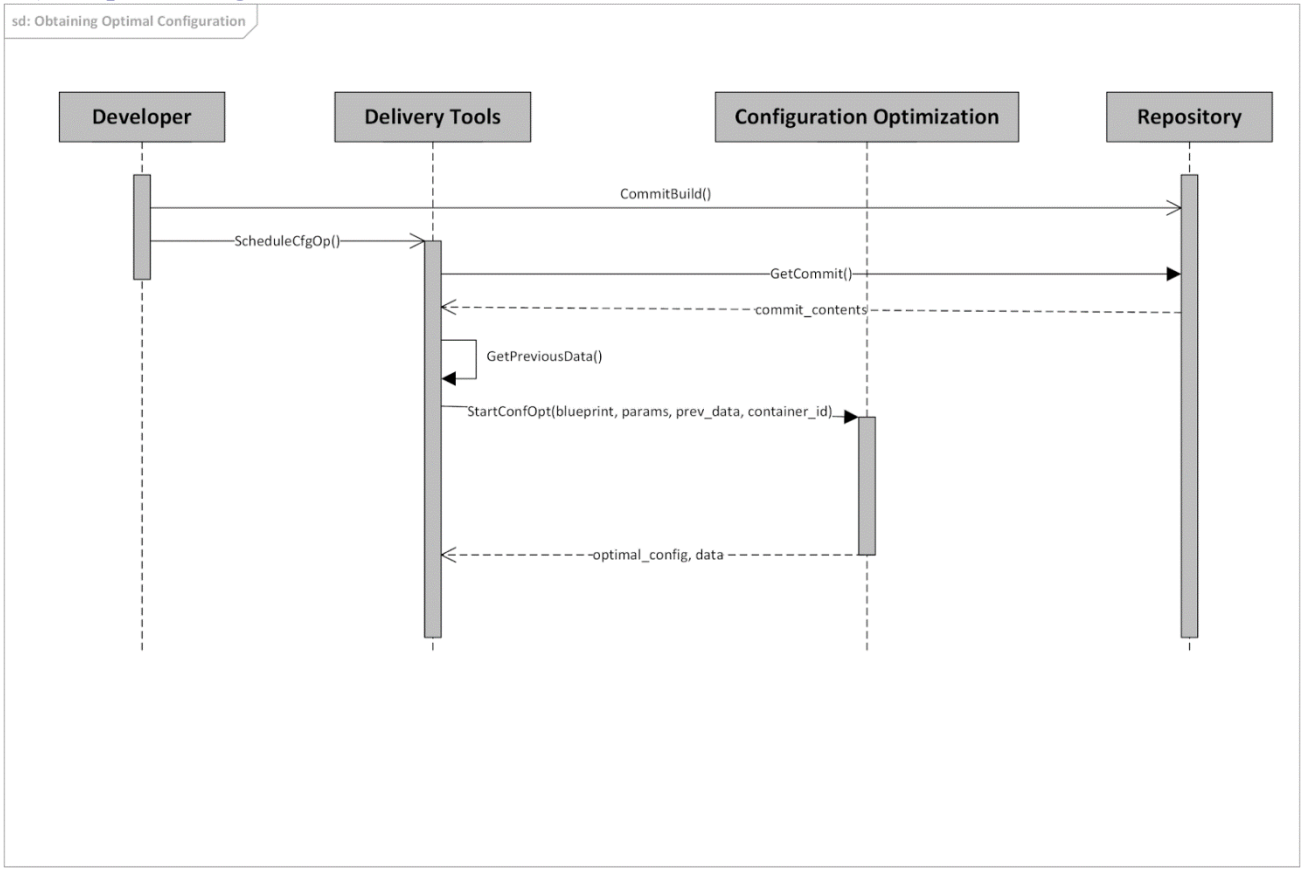
In year 2 we have refined the interaction model of the Configuration Optimization (CO) tool with an asynchronous (background processing) approach, where the user can set the optimization to run and then forget it until it finishes. The configuration recommendation is a result of a sophisticated process, which is carried out by the CO 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 CO is a relatively lengthy non-interactive process, which also needs to use the resources of the testbed, we preferably move it away from the IDE, as it would be typically executed in batch mode overnight.

Therefore, the developer needs to schedule the configuration optimization, and this is possible through Delivery Tools. The Delivery Tool schedules the invocation of the Configuration Optimization tool, supplying it with the previous data and the TOSCA application topology in a blueprint bundle. It then starts its iteration towards an optimal optimization. When it finishes, the Delivery Tool collects the new data and

the optimal configuration, both returned by the Configuration Optimization tool, and stores it in the Continuous Integration's job result.

b) Sequence diagram



c) Data flows

- (for diagram 1) 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 Delivery Tools for the developers to use with the subsequent deploys and further application development.

- (for diagram 2) 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.