

**Developing Data-Intensive Cloud
Applications with Iterative Quality
Enhancements**



DICE delivery tools – Initial version

Deliverable 5.1

Deliverable:	D5.1
Title:	DICE delivery tools – initial version
Editor(s):	Matej Artač (XLAB)
Contributor(s):	Giuliano Casale (IMP), Pooyan Jamshidi (IMP), Tatiana Ustinova (IMP), Gabriel Iuhasz (IeAT), Matej Artač (XLAB), Tadej Borovšak (XLAB), Matic Pajnič (XLAB)
Reviewers:	Daniel Pop (IeAT), Marc Gil (PRO)
Type (R/P/DEC):	Demonstrator
Version:	1.0
Date:	31-January-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

This deliverable is a report to accompany the initial version of the DICE delivery and configuration tools. These tools handle the transition from writing code and running off-line analysis and checks to having an actual running data-intensive application in the test bed. In this way, they automate tasks, which would exceed the skills of the users who are new to the development of the data-intensive applications, and which are also too complex to be performed manually.

The suite of delivery tools consists of three components: a deployment tool, a continuous integration tool, and a configuration optimization tool. Each of the tools can work in a stand-alone mode to accomplish a specific task. However, they show their strength when used together in a larger DICE workflow. By this we mean the use of the OASIS TOSCA (Topology and Orchestration Specification for Cloud Applications) format for describing deployments from models (the DICE DeploymentSpecific Model or DDSM, to be precise), the availability of the monitoring platform for measuring performance of the application's runtime, and other services such as the enhancement tools.

The DICE Deployment Tool's purpose is to materialise an application described in a TOSCA document, also named a blueprint, in the designated test bed. We developed a thin wrapper service around the existing cloud application orchestration tool Cloudify, which already provides TOSCA blueprint parsing and deployment on common platforms. This let us to focus on supporting the building blocks (i.e., the technologies such as Hadoop, Yarn, Kafka, etc.), and as a result the users experience deploying their applications as a fully automated process. This solution saves a Deployment Tool user a lot of time required for learning and experimenting with installation of individual services. In the DICE solution, we also provided the essential functionality for deploying blueprints in the project's test bed platform of choice, the Flexiant Cloud Orchestrator (FCO).

For the DICE Continuous Integration, we again chose a solid base solution, the Jenkins Continuous Integration tool. This gives us a service for scheduling complex and long jobs while ensuring continuous building, deployment and testing of the application being developed. The DICE contribution to the process is to record and visualize the measured quality aspects of the application development history. This gives the users an insight into the non-functional properties of the application on top of the typical success/failure of unit testing.

Finally, the DICE Configuration Optimization tool acts as an expert, which examines the application's DDSM, measures iteratively its runtime, and in the given time and budget provides a recommended optimal configuration for the application. In this way it helps the users to improve their application's performance without having to understand arcane service configurations. The tool is also efficient in that it only examines a subset of all the possible configurations, which normally form a prohibitively large search space.

Glossary

DDSM	DICE Deployment Specific Model
DICE	Data-Intensive Cloud Applications with iterative quality enhancements
DPIM	DICE Platform Independent Model
DTSM	DICE Technology Specific Model
FCO	Flexiant Cloud Orchestrator
TOSCA	Topology and Orchestration Specification for Cloud Applications
IDE	Integrated Development Environment
CI	Continuous Integration

Table of contents

EXECUTIVE SUMMARY	3
GLOSSARY	4
TABLE OF CONTENTS	5
LIST OF FIGURES	6
1. INTRODUCTION.....	8
1.1. Motivation for the tools	9
1.1.1. DICE Deployment Tool.....	9
1.1.2. DICE Continuous Integration Tool	10
1.1.3. DICE Configuration Optimization	10
2. ARCHITECTURE	15
2.1. High level architecture	15
2.2. Stakeholders and use cases	16
3. TOOLS.....	17
3.1. DICE Deployment Tool.....	17
3.1.1. Main components	17
3.1.2. TOSCA documents for describing applications	18
3.1.3. Tools usage	21
3.1.4. Deployment tool REST API.....	23
3.1.5. Obtaining the Deployment Tool	23
3.2. DICE Continuous Integration Tool.....	24
3.2.1. Main components	24
3.2.2. Tool usage.....	24
3.2.3. Obtaining Continuous Integration Tool.....	26
3.3. Configuration Optimization Tools	26
3.3.1. Main components of Configuration Optimization Tool	26
3.3.2. Behaviour of the Configuration Optimization Tool	27
3.3.3. Configuration Optimization Tool usage	28
3.3.4. Results	29
3.3.5. Obtaining Configuration Optimization Tool	30
4. CONCLUSION.....	31
REFERENCES.....	34

List of Figures

Figure 1: DICE delivery and configuration tools architecture. Delivery and configuration tools are represented in blue boxes, while external components are in grey boxes.	15
Figure 2: Service components making up the DICE deployment tools.	18
Figure 3: Contents of a TOSCA blueprint package.	19
Figure 4: An example TOSCA blueprint for a Storm topology.	20
Figure 5: A graphical representation of the example TOSCA blueprint for a Storm topology.	21
Figure 6: Basic Deployment Tool service use cases. Each row shows the state of the service (middle) and the test bed (right) at each point in the usage sequence.	22
Figure 7: DICE plug-in for Jenkins displays a history of the measured performance metrics (e.g., latency) as a chart, and the history of the builds as a table	25
Figure 8: Options available when configuring a continuous integration job for the DICE Continuous Integration plugin.	25
Figure 9: Configuration optimization architecture.	27
Figure 10: Internal behaviour of configuration optimization tool.	27
Figure 11: Storm architecture	28
Figure 12: WordCount topology.....	29
Figure 13: Distance to optimum configuration, error is in milliseconds.....	30

List of Tables

Table 1: Level of compliance of the initial version of the DICE delivery tools with the initial set of requirements.....	32
--	----

1. Introduction

The DICE toolset aims to provide a complete support for developers of data-intensive applications covering all stages of the development process from application design to the deployed fully working application in the pre-production environment. DICE also offers a number of tools in order to monitor the operational environment and refactor the architectural designs [23] based on performance feedback. In other words, DICE enables performance-oriented DevOps [24][24] for data intensive applications. The role of the delivery and configuration tools in the development workflow is to substantially reduce the manual effort needed for installing and configuring the application in the test bed or a pre-production environment by automating the process. It also helps the agile software development teams to take advantage of the automation of the builds and deployments, speeding up the design-release cycles. There are a number of tools on the market that can help the development teams in this respect.

The goal of the DICE tools is two-fold. First, it aims to simplify the development and deployment of the data-intensive applications so that it becomes more accessible to the new and less experienced programmers. Second, the emphasis is on the tools' ability to provide recommendations on how the vital application components should be configured in order to achieve efficient and reliable application runtime. Existing tools require users' expertise and an extensive set of tweaking the configuration to find the optimal solution. Finally, the DICE tools aim to provide feedback on the quality of the application being developed.

In the first year of the DICE project we have defined and outlined the basic capabilities the tools should possess in order to achieve the stated goal. They include:

- Adoption of the Cloudify [4] tool as a core of the deployment tool. This solution provides the tool with support for deploying applications described in TOSCA (Topology and Orchestration Specification for Cloud Applications) [7] formatted models.
- Creation of a thin wrapper service for a simplified management of the deployments provided by Cloudify.
- Basic support for the Flexiant Cloud Orchestrator (FCO) [9] platform for the Cloudify.
- Implementation of a tool for actively testing and optimizing the configurations of the applications.
- Implementation of a prototype of the Configuration Optimization tool.
- Support for the initial technologies: Storm [10], Spark [13], Zookeeper [11], Kafka [12], Yarn [14].

This document describes the tools, their usage and the benefits that the tool brings to the users. It provides both non-technical and technical aspects of their implementation and use. The rest of the section discusses the motivation for individual tool in the DICE delivery toolset. The Section 2 summarizes the requirements for the DICE delivery tools, extracted from the D1.2 [2]. In the Section 3, we present the top-level architecture of the delivery and configuration tools. In Section 4 we present each tool in a deeper technical level, also presenting their usage and preliminary results. Finally, in Section 5 we present the conclusions.

1.1. Motivation for the tools

1.1.1. DICE Deployment Tool

For beginners in the world of data-intensive applications and their support services, one of the major obstacles is preparation of the environment to support the runtime of the application. Modern data storage and processing services require a number of time-consuming steps to install, configure and connect the services into an inter-dependent mesh of a functional application. With every application, the topology of services and components is different to address the specific data flows and workflows particular to each application.

A well-established rule in industry is that manual installation and deployment of any application services is not recommended for anything beyond learning and simple testing of the technologies. A much better approach to the problem is to treat both infrastructure and the components deployed on top of the infrastructure as software [6]. This has a great benefit of being able to version the changes, trace any changes, and provide repeatable and predictable results with each deploy.

To achieve this goal, a number of tools exist that automate configuration of individual nodes (e.g., virtual machines) at the lowest level, including Chef [5], Puppet [15], Ansible [16], etc. But deploying and running applications takes coordination at a higher level to ensure that interdependencies are set up in a proper order and configured to properly discover each other. For instance, a web application needs a web server to be configured first, and a database set up and running on another node. This is done by the orchestrator tools, representatives of which include Ubuntu Juju [17], Apache Brooklyn [18], Flexiant Cloud Orchestrator [9] and Cloudify [4].

The listed tools themselves are not enough, because the configuration tools need cookbooks, recipes or other configuration definitions to be able to install and configure useful services. The orchestration tools then need application blueprints on top of the cookbooks and recipes. Many cookbooks are available from the open source community, but their usage requires at least some expertise on the technologies, they do not always work out of the box, and even if they do, they could be incompatible with each other.

In DICE, we aim at building tools that are transparent and as easy to use as possible. They need to fit into the model-driven development approach, where the most detailed level of the model – the DDSM – is represented according to the OASIS [19] TOSCA standard [7]. The deployment tool also needs to be simple to use, granting it a good usability as a stand-alone tool. Most of the users will not use it directly, however, therefore we also designed it to be easy to integrate with the other tools.

The models involving the DICE supported building blocks also need to work when instructed to deploy. That is why we provide cookbooks and predefined TOSCA types that they are tested to be compatible for arbitrary (but sensible) topologies that the users might model.

With the DICE Deployment Tool, we believe that the users will gain a well-rounded and complete solution. It will be easy and quick to set up, yet powerful enough to autonomously install almost any application, which uses one or more technologies from a growing library of the DICE building blocks. The users will not have to handle the configuration recipes, but instead invest the time into developing their own applications.

1.1.2. DICE Continuous Integration Tool

Software development teams decide for agile development in order to speed up the process of designing, developing, testing and deploying their products. An important part of this speed-up is the ability of the whole development toolset to build, test and deploy the application, on a schedule or whenever the developers make an update of their code. Doing this automatically releases the developers from performing complex, possibly long-lasting tasks.

Continuous Integration is a practice [20][23][24], which has two aspects. First, the developers, who normally develop their updates and new features in isolation from the currently stable main development branch, have to be disciplined and merge their changes with the main development branch on a daily basis. This assures that the incremental changes that require merging are smaller and less complex. This way any conflicts are caught early and can be easily resolved.

The second aspect is that a supporting toolset exists, which tests and evaluates the changes, assesses the stability of the main branch, packages and deploys the changes. The glue to the processes provided by these tools is a common tool, which is both a portal for managing continuous integration jobs and a central record place for the current and past integration executions.

In DICE, the Continuous Integration tool takes over the complex and repetitive processes, which are an integral part of the quality-driven development methodology. It also provides a visual representation of the history of the application's performance, helping the developers assess the progress of the application development.

1.1.3. DICE Configuration Optimization

The main objective of the configuration optimization tool is to find the optimum configuration for big data applications within limited budget. This limited budget can be specified by either limited time or pre-determined experimental time.

Configuration parameters internal to the services used by the data-intensive application provide a way to significantly change the performance of the application. Finding appropriate configuration options is a difficult task specifically for big data systems. These can employ several frameworks such as Apache Storm [10], Kafka [12], Spark [13] and many more in order to develop a robust application that is able to process large amount of data. Each of these frameworks has hundreds of configuration parameters and tuning them to get the best performance is not a trivial task. The difference between a tuned framework versus an un-tuned framework is several orders of magnitude in terms of performance. However, the problem of finding the optimum configuration is time consuming and requires an experimental test bed. For example, if we consider 10 configuration parameters, each of which have 4 configuration options, we require to perform $4^{10}=1\text{M}$ tests and if for each test we consider 10 minutes, in total we require to spend ≈ 19 years to find the best configuration if we would follow a naive full factorial design for testing. Notice that 10 out of hundreds is only a small subset of the available configuration parameters in such systems.

The DICE Configuration Optimization tool therefore takes over the task of an automated expert, which can experimentally arrive at a good set of configuration parameters. This helps move the application away from the default parameters, which may not always work optimally. It also improves any configuration that the users have arrived at through manual testing and tweaking. As

Deliverable 1.1. State of the art analysis.

a result, DICE provides deployments, which are tuned to the specific user's' requirements, environment and application.

2. Requirements

In Deliverable D1.2 [2], we presented the requirement analysis for the DICE project. This section summarises the requirements related to the DICE Delivery tools. The actors involved include CI_TOOLS, which represent the DICE Continuous Integration tools, the DEPLOYMENT_TOOLS, which represent the DICE DICE deployment tool

ID	R5.3
Title	Continuous integration tools deployment
Priority	Must have
Description	The ADMINISTRATOR MUST manually install and configure CI_TOOLS MUST upon installation of the CI_TOOLS and can be updated later on. The configuration MUST enable CI_TOOLS to access the TESTBED.

ID	R5.4
Title	Translation of TOSCA models
Priority	Must have
Description	The DEPLOYMENT_TOOLS MUST be able to translate TOSCA models from WP2 into the supported target configuration manager's DSL for orchestration

ID	R5.4.1
Title	Deployment plan support
Priority	Must have
Description	The DEPLOYMENT_TOOLS MUST be able to deploy all the DICE supported core building blocks.

ID	R5.4.2
Title	Translation tools autonomy
Priority	Must have
Description	The DEPLOYMENT_TOOLS MUST take all of its input from the TOSCA model and therefore MUST NOT require any additional user's input.

ID	R5.4.7
Title	Deployment of the application in a test environment

Priority	Must have
Description	The DEPLOYMENT_TOOLS MUST provision the resources required by the application

ID	R5.4.8
Title	Starting the monitoring tools
Priority	Must have
Description	The DEPLOYMENT_TOOLS MUST start the MONITORING_TOOLS for the application.

ID	R5.5
Title	User-provided initial data retrieval
Priority	Must have
Description	CI_TOOLS MUST retrieve from the artifact repository or use input from the code versioning system any user-provided initial data

ID	R5.7
Title	Data loading support
Priority	Must have
Description	DEPLOYMENT_TOOLS and QTESTING_TOOLS MUST support bulk loading and bulk unloading of the data for the core building blocks.

ID	R5.16
Title	Provide monitoring of the quality aspect of the development evolution (quality regression)
Priority	Must have
Description	The CI_TOOLS MUST record the results of each test and map them to the momentary project's (model, code etc.) version.

ID	R5.19
Title	Deployment configuration review
Priority	Must have

Description	The CI_TOOLS MUST enable that ADMINISTRATOR assigns one or more users (including self) for reviewing the deployment configuration
--------------------	---

ID	R5.20
Title	Build acceptance
Priority	Must have
Description	The CI_TOOLS MUST NOT run the deployment of the application to pre-production if the quality test fail or the reviewers have not provided a positive score.

ID	R5.27
Title	Recommender Engine and Optimization
Priority	Must have
Description	DEPLOYMENT_TOOLS (recommender engine) MUST retrieve from the OPTIMIZATION_TOOLS initial deployment parameters and recommend the vaules of the parameters that have not yet been set.

ID	R5.27.2
Title	Recommender Engine API
Priority	Must have
Description	DEPLOYMENT_TOOLS MUST provide APIs to access recommender system (push data, get recommendations, etc)

3. Architecture

3.1. High level architecture

The DICE delivery and configuration tools are an important part of the DICE tools workflow. They receive the code and design (the model) of the application after the developer has used off-line tools (IDE, formal analysis tools, optimization). Their goal is to bring the application to life as a deployment in a test bed. Effectively, they represent the Ops part of the DevOps. The full DICE architecture is available in [3], and the tools described here are represented as: Configuration Optimization and Delivery tools.

In this document, we focus on the architecture and interactions between these two top-level components and the components they depend on. The Figure 1 illustrates this in more detail. Each component can work stand-alone, but they were created to interoperate in an integrated toolset.

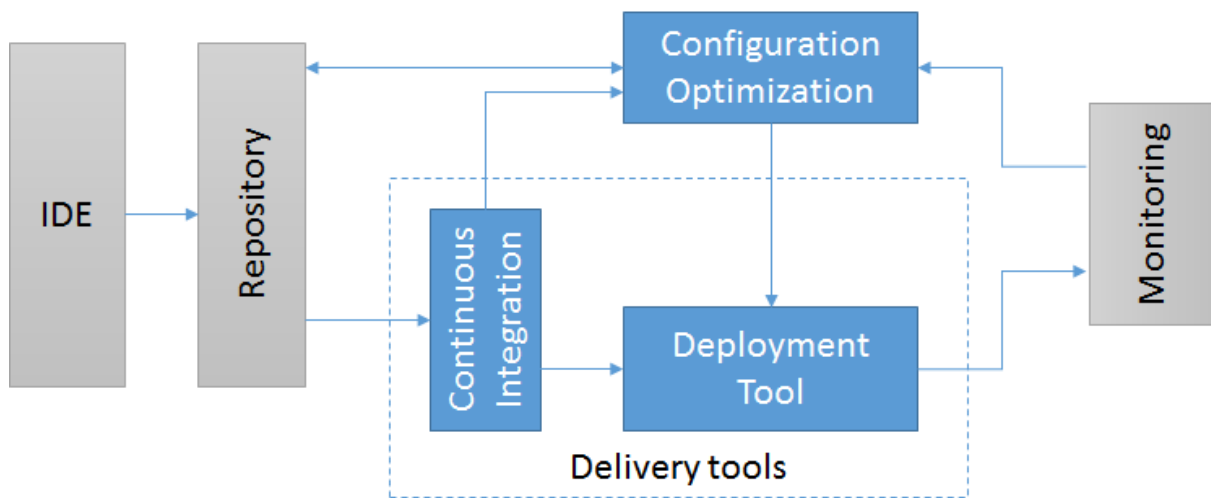


Figure 1: DICE delivery and configuration tools architecture. Delivery and configuration tools are represented in blue boxes, while external components are in grey boxes

In the architecture, we consider the **IDE** as an external component, where the developers author most of the content and the input to the other tools. The **Repository** is another external component, which keeps the code, models and artifacts created by the developers in a versioned form and available for retrieval and updates.

We represent the Delivery tools as two components: the Continuous Integration Tool and the Deployment Tool. The **Continuous Integration** depends on the Repository to provide the information about the project's updates. It also provides the code and models themselves. The Continuous Integration service then triggers Deployment Tool or Configuration Optimization when needed.

The **Deployment Tool** uses the external **Monitoring** to properly set up the individual nodes' monitoring agents, enabling the stream of the monitoring data to flow to the monitoring platform.

Finally, the **Configuration Optimization** uses the Repository to receive the current model, and to deposit the updated model with the recommended optimal configuration after it is done running. It also depends on the Monitoring to measure and provide the runtime performance metrics.

3.2. Stakeholders and use cases

Stakeholders are the actors who interact with the components and tools. They either require the features that the components and tools provide, or are involved in the workflow mandated by the components and tools. We have so far identified the following stakeholders:

- System Administrator: involved only for a short time when the delivery and configuration tools need to be installed or reconfigured.
- Developer: this is the main stakeholder, who uses the majority of the tools' features. Developers write code of the application and design the application models. They continuously update the application, requiring constant updates to the deployment of the application in the test bed. Developers occasionally require the optimal configuration of their applications' topologies.
- Architect: similar actor to developer, except that they interact with the tools less frequently and normally only focus on the topology and optimal configuration of the application's design.
- Quality assurance tester: they rely on the Continuous Integration tool to run the functional tests that they prepare as well as the non-functional tests. They also take advantage of the applications' deployment in the test bed, where they can, for example, perform A/B testing.

4. Tools

4.1. DICE Deployment Tool

4.1.1. Main components

The main goal of the DICE deployment tool is to accept a TOSCA document, and deploy the application in the target environment. The **core component** would therefore be a cloud application orchestration tool, which needs to:

- Be capable of translating the topology described in the TOSCA document into a deployment plan of the application.
- Be as compliant as possible with the TOSCA YAML standard.
- Support or use Chef recipes to handle the actual configuration management of the individual nodes.
- Run as a multi-tenant service.
- Offer a management and usability interface by exposing RESTful or SOAP services, or respond to topics of a messaging bus.
- Support IaaS managers such as OpenStack, EC2 and Eucalyptus, and the ability to implement support for additional environments.
- Be open source.

According to our survey [1], a number of existing solutions cover some of the listed requirements, but we found that only Cloudify [4] fulfilled all of them.

With the TOSCA standard gaining in popularity, we can expect that other tools will emerge, providing an improved set of features that users of the DICE deployment tool could benefit from (e.g., a simple ability to reuse parts of an existing deploy when updating the application with small changes in configuration). Hence, we plan to re-evaluate our decision of using Cloudify at later project milestones.

In case we do decide to replace our core component with one from another vendor, we still want to preserve the way other DICE tools access the deployment. We also want to provide an access to the deployment tool that is simple for the users and for integration with other services, focusing only on the frequently used workflows and functionalities. To implement this simplification, the DICE tools consist also of a thin **front-end service**. In this way it will also be a relatively simple matter to implement client authentication and access restrictions.

Another essential component of the DICE deployment tool is the **Chef server**. The component comes from a third party [5], but it serves both as the supported technologies' cookbook repository and the manager of the Chef cookbook workflow executions at the target nodes. DICE also provides the content side in the shape of the **Chef cookbook library**, which provide recipes for all the supported technologies. We tailor made the cookbooks so that their recipes take advantage of the parameters supplied by the Cloudify's Chef plug-in. Conversely, we added the recipes, which support crucial events in the application orchestration workflow.

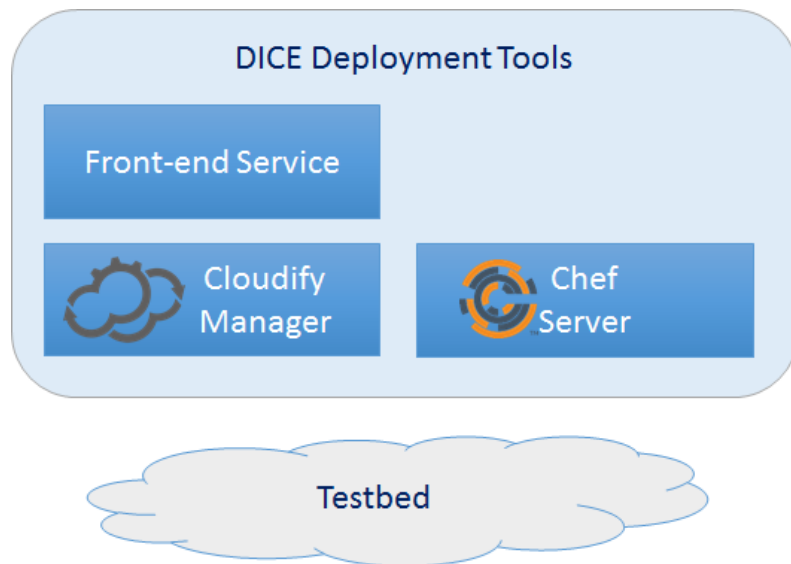


Figure 2: Service components making up the DICE deployment tool

In the Figure 2, we can see a diagram of the deployed services, which make up the DICE deployment tool. The clients are either users, which use a command-line interface to access the front-end service, or other DICE tools such as the DICE Continuous Integration tool or the DICE Configuration Optimization tool, which directly use the RESTful interface of the Front-end Service. The Front-end service interacts with the Cloudify Manager, which in turn accesses the test bed's IaaS API to manage the lifetime of the virtual machines. From each of the managed nodes (i.e., virtual machines), Cloudify's Chef plug-in retrieves the cookbooks specified in the TOSCA blueprint for the node, and runs its recipes to install, configure, connect or clean up the application components.

4.1.2. TOSCA documents for describing applications

The components described in the previous section represent the infrastructural framework of the DICE Deployment Tool. In principle, the users could interact with this tool by submitting fully-functional and valid TOSCA blueprints in order to deploy their applications. However, this would require recipe and blueprint preparation and enough knowledge of the TOSCA, recipes and services. With DICE, we want to simplify the blueprint preparation process as much as possible.

To do this, we provide the content aspect of the DICE Deployment Tool, which takes form of a set of prepared **TOSCA node types** corresponding to the classes in the DDSM. These come as a set of importable YAML files, covering all of the generic and reusable constructs that the users' applications will need. In particular, the following modules are available:

- DICE supported technology types. These define the nodes, which correspond to the DICE supported building blocks. They inherit from a generic Chef node, provide definitions of the parameters for configuring each nodes along with reasonable default values. They also define the cookbook recipes needed to be run when Cloudify creates and configures the node, then connects it to the other nodes based on their dependences, and starts the node. They also define the steps needed when destroying the node.
- Target IaaS vendor DICE definitions. To provide an abstraction from the underlying hosting environment (e.g., OpenStack, FCO, EC2...), we provide the node types, which correspond to virtual machines of various flavours and sizes.

- A template for the blueprint inputs. These define operational parameters of the DICE deployment tool, and are thus a responsibility of the administrator, who sets up the deployment tool.

In practice, the user should then only provide the blueprint in a form of a set of node templates and relationships, each of which instantiates one of the prepared types. In this way, the details of the implementation (such as the scripts needed to run to establish relationships between nodes) are kept to a minimum in the end-user's blueprint.

The Figure 3 illustrates the package that the deployment service needs to receive in order for the application to be successfully deployed. The `blueprint.yaml` is the only document that the user needs to provide, while all the others are effectively a fixed part of the DICE Deployment Tool. Note that in the integrated environment of the DICE tools, even the blueprint itself will be an outcome of a model-to-text transformation, so effectively the user should not have to edit it in the first place.

The diagram shows a file explorer view of a TOSCA blueprint package. Three callout boxes provide context: 'Installed globally by Admin' points to the package root, 'Created manually or generated from DDSM' points to the `blueprint.yaml` file, and 'Shipped with DICE' points to the `scripts` and `types` folders.

Name	Date Modified	Size
<code>blueprint.yaml</code>	Today 10:32	6 KB
<code>inputs-fco.yaml</code>	Today 10:32	3 KB
<code>scripts</code>	Today 10:32	--
<code>configure_hosts.sh</code>	Today 10:32	645 bytes
<code>configure_zookeeper.sh</code>	Today 10:32	207 bytes
<code>connect_storm_to_nimbus.sh</code>	Today 10:32	390 bytes
<code>connect_storm_to_zookeeper.sh</code>	Today 10:32	418 bytes
<code>connect_zookeeper_servers.sh</code>	Today 10:32	557 bytes
<code>start_zookeeper_service.sh</code>	Today 10:32	99 bytes
<code>types</code>	Today 10:32	--
<code>dice-flexiant.yaml</code>	Today 10:32	1 KB
<code>dice-openstack.yaml</code>	Today 10:32	909 bytes
<code>dice.yaml</code>	Today 10:32	4 KB

Figure 3: Contents of a TOSCA blueprint package

Nevertheless, the DICE Deployment Tool can be used as a stand-alone tool, so the users are able to manually prepare their application's blueprints. The Figure 4 shows an example YAML document, which describes a topology for deploying Storm. The same topology is represented graphically in the Cloudify's GUI shown in the Figure 5.

```

tosca_definitions_version: cloudify_dsl_1_1

imports:
  # common imports
  - http://www.getcloudify.org/.../types.yaml
  - http://www.getcloudify.org/.../chef.yaml
  - types/dice.yaml

  # OpenStack imports
  - http://www.getcloudify.org/.../openstack.yaml
  - types/dice-openstack.yaml

# The inputs are fixed.
inputs:

  # omitted for brevity

node_templates:

  # one medium-sized instance host for zookeeper
  zookeeper_host:
    type: dice.medium_host
    instances:
      deploy: 1

  # 1 med-sized instance hosting the Storm Nimbus
  storm_nimbus_host:
    type: dice.medium_host
    instances:
      deploy: 1
    relationships:
      - type: [...]server_connected_to_floating_ip
        target: storm_floating_ip
      - type: [...]connected_to_security_group
        target: storm_security_group

  # 3 med-sized instances for hosting Storm
  storm_host:
    type: dice.medium_host
    instances:
      deploy: 3

  # The node templates defining actual services.
  # Their definition is fixed except for the
  # properties some of the services might have.
  zookeeper:
    type: dice.zookeeper
    # zookeeper service properties, which
    # can be set explicitly; if omitted their
    # default values will be used implicitly
    properties:
      tickTime: 1500
      initLimit: 10
      syncLimit: 5
    relationships:
      - type: cloudify.relationships.contained_in
        target: zookeeper_host
        source_interfaces:
          # omitted for brevity
    interfaces:
      # implementation detail, which can be reused
      cloudify.interfaces.lifecycle:

    create: scripts/configure_hosts.sh

    # this node template defines the Storm
    # Nimbus service
    storm_nimbus:
      type: dice.storm_nimbus
      relationships:
        - type: cloudify.relationships.contained_in
          target: storm_nimbus_host
        - type: cloudify.relationships.connected_to
          target: zookeeper

  # node template to define regular Storm service
  storm:
    type: dice.storm
    relationships:
      - type: cloudify.relationships.contained_in
        target: storm_host
      - type: cloudify.relationships.connected_to
        target: zookeeper
      - type: cloudify.relationships.connected_to
        target: storm_nimbus

  # platform-specific node templates
  storm_floating_ip:
    type: cloudify.openstack.nodes.FloatingIP

  storm_security_group:
    type: cloudify.openstack.nodes.SecurityGroup
    properties:
      security_group:
        name: ma_cloudify_storm
    rules:
      - remote_ip_prefix: 0.0.0.0/0
        port: 8080
      - remote_ip_prefix: 0.0.0.0/0
        port: 22

outputs:
  storm_nimbus_gui:
    description: URL of the Storm nimbus gui
    value: { concat: [ 'http://', { get_attribute:
[storm_floating_ip, floating_ip_address] },
':8080' ] }

```

Figure 4: An example TOSCA blueprint for a Storm topology

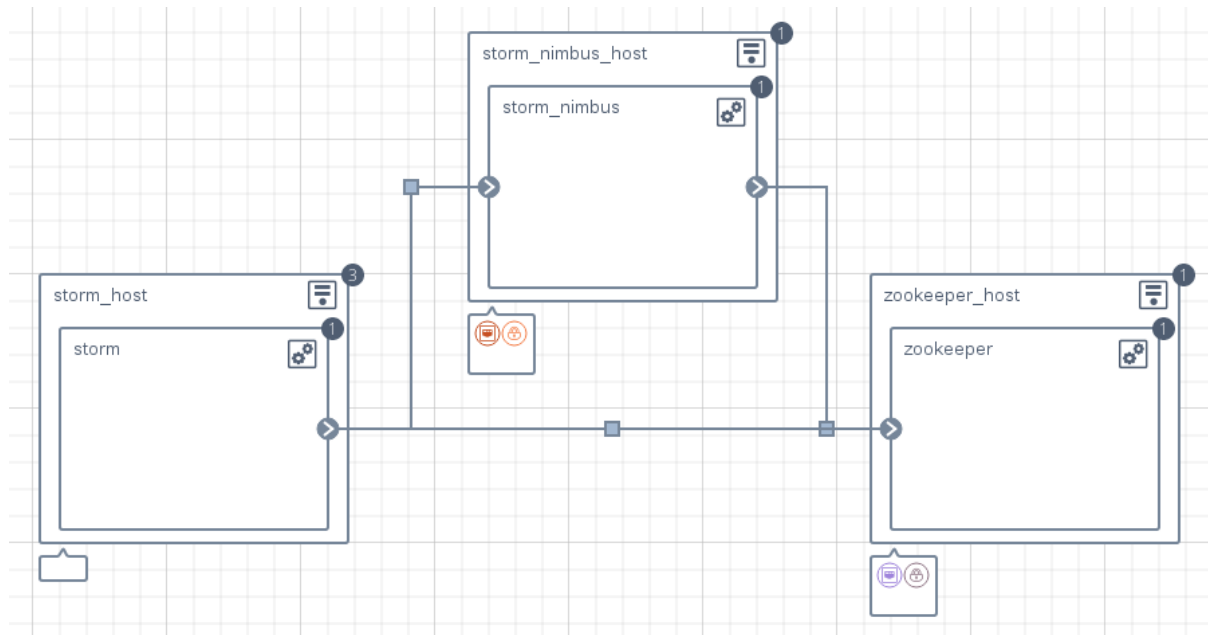


Figure 5: A graphical representation of the example TOSCA blueprint for a Storm topology

To briefly describe the blueprint, we can see that each of the top-level node templates use a type called `dice.medium_host`, which represents a medium-sized virtual machine (i.e., with 2 GB of RAM, 2 virtual CPUs and 10 GB of storage). Other flavours are available for larger or smaller nodes.

The services themselves are then represented as node templates connected with the “contained in” relationship to their respective hosts. These services include Zookeeper, which orchestrates the services during their runtime, Storm Nimbus, which coordinates the Storm cluster, and a regular Storm services. The related services are, in turn, connected with “connected to” relationships.

The blueprint also specifies the infrastructure-related properties of the topology. The Storm Nimbus node needs to expose its network interfaces to the public in order to provide the status web GUI functionality. Thus, we define a node representing a floating IP, and a node for the security group, regulating the traffic between the public Internet and the Storm Nimbus node.

4.1.3. Tools usage

The Deployment Tool will work with **deployments or blueprints**. We use the terms blueprint and deployment as synonyms here, because there exists 1:1 mapping between them (we consider each blueprint submission as its own deployment). The deployment is described in the deployment documents (blueprints), complete with all the services and applications needed to run independently of any other instance of the same application or any other application. The definition is based upon one Cloudify's deployment: *"a virtual environment on your Cloudify manager with all of the software components needed to execute the application lifecycle described in a blueprint, based on the inputs provided in the `cfy deployments create` command."*¹

¹ Cloudify's deployment definition: <http://getcloudify.org/guide/3.2/quickstart.html>

Another abstraction offered by deployment tool is virtual container that can contain zero or one deployment. Main purpose of virtual container is to offer a stable endpoint for scenarios where redeployments happen often (e.g., automatic testing).

We implemented the DICE deployment tool as a service, exposing a RESTful web service interface as its front-end. The service is a thin layer, which uses a third party orchestration solution such as Apache Brooklyn or Cloudify for its back-end.

The Figure 6 illustrates the sequence of steps in the typical use of the tool:

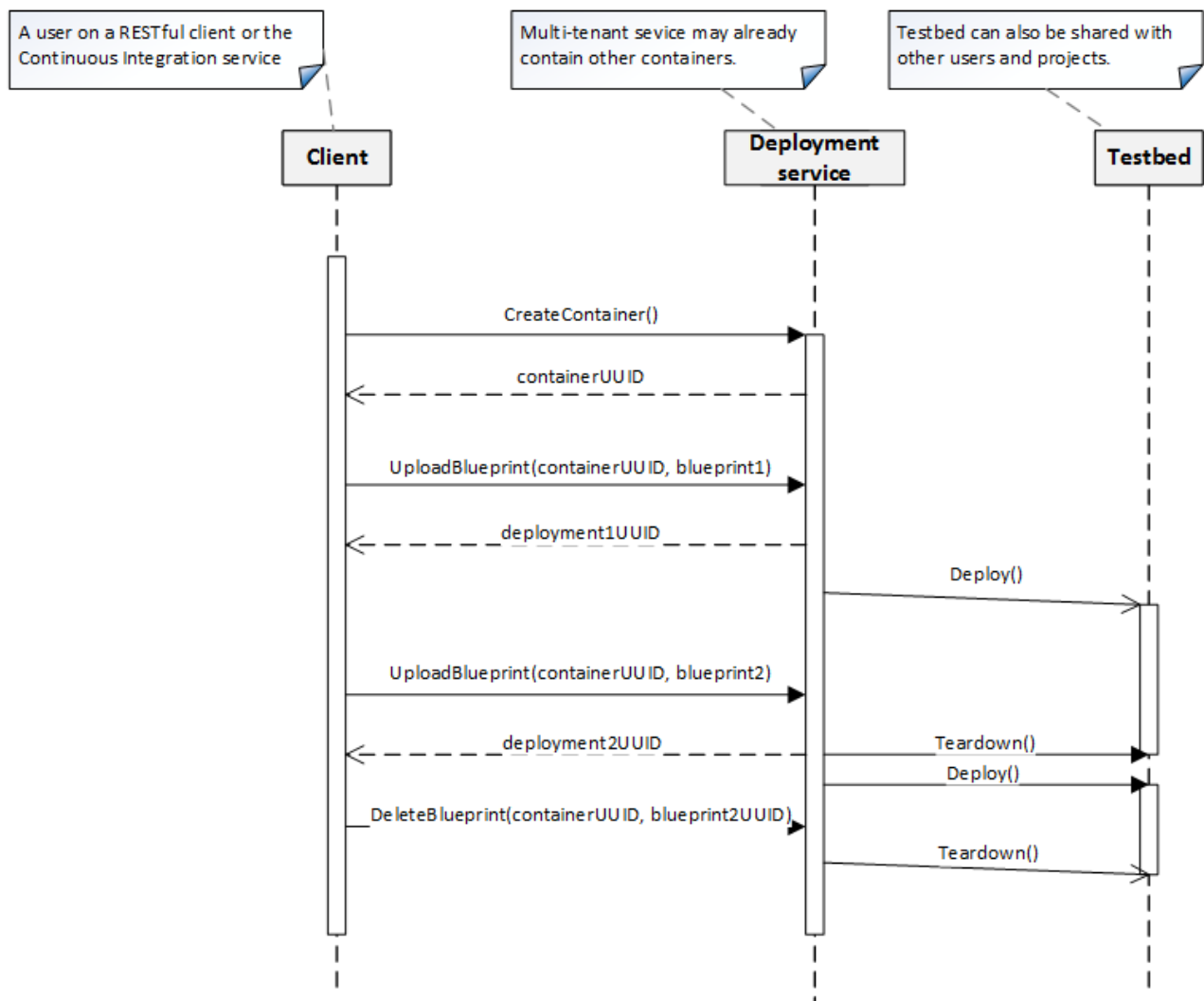


Figure 6: Basic Deployment Tool service sequence

- User creates new virtual container that will hold deployment.
- User uploads blueprint to the previously created virtual container and receives back an identifier that can be used in subsequent interactions with this deployment.
- After receiving the blueprint, virtual deployment tool initiates deploy using backend orchestrator. Users can monitor the progress by querying deployment tool using previously received deployment identifier.
- User updates blueprint and uploads it to the same virtual container as before.
- After receiving an updated blueprint, the virtual deployment tool initiates a teardown of existing deployment and deploy of updated blueprint (possibly in parallel).

- To tear down an existing deployment, user simply requests deletion from deployment tool using deployment identifier.

4.1.4. Deployment tool REST API

In order to support the previously described functionality, the deployment tool exports the following API endpoints:

1. *POST /containers*
Create a new virtual container. Call response contains container identifier (UUID) and status information (initially, container is empty).
2. *GET /containers*
List all virtual containers and their status information.
3. *GET /containers/{identifier}*
Display the status information about the selected virtual container.
4. *DELETE /containers/{identifier}*
Remove the selected virtual container.
5. *POST /containers/{identifier}/blueprints*
Upload a new blueprint to the selected container and start a deploy. If the container already contains a deployment, teardown is first initiated. The call's response contains the deployment identifier and status.
6. *GET /blueprints*
List all blueprints and their statuses.
7. *GET /blueprints/{identifier}*
Get status of the selected blueprint.
8. *DELETE /blueprints/{identifier}*
Tear down the selected deployment and delete the accompanying blueprint.

Note that creating and deleting deployments are time consuming operations. Each of the call is therefore asynchronous, meaning that it returns as soon as it successfully records and initiates an operation, but does not wait for the end of the operation. In order to monitor the progress, the user can continually poll the deployment tool about the deployment status.

4.1.5. Obtaining the Deployment Tool

The DICE Deployment Tool is available on GitHub in multiple components:

- <https://github.com/dice-project/DICE-Deployment-Service> – the thin wrapper and the RESTful web service interface of the DICE Deployment Tool.
- <https://github.com/dice-project/DICE-FCO-Plugin-Cloudify> – the client for the FCO and the plug-in used by the Cloudify TOSCA blueprints for orchestrating deployments in the FCO.
- <https://github.com/dice-project/DICE-Deployment-Cloudify> – this repository contains the TOSCA definitions and the scripts for supporting the configuration of the supported Big Data services.
- <https://github.com/dice-project/DICE-Chef-Repository> – Chef repository with the cookbooks that the TOSCA definitions refer to.

4.2. DICE Continuous Integration Tool

4.2.1. Main components

The DICE Continuous Integration tool has the following main functions and purposes:

- To periodically, on certain events (such as a new commit into the project) or when triggered manually, perform continuous integration *jobs*.
- To host the continuous integration jobs, which consist of a sequence of steps, including: fetching the latest commit, compilation, verification (e.g., via user-provided unit tests), and deployment of an application.
- To schedule and run long tasks, such as configuration optimization.
- To record the progress of a project and present a visual representation of the project execution history.

To accommodate for these functional requirements, the DICE Continuous Integration tool consists of the following components:

- Jenkins [21] Continuous Integration tool is the basis for the DICE Continuous Integration tool. It is an open-source solution, which is both popular with many software teams as well as a tool of choice for the DICE use cases [2].
- A DICE plug-in for Jenkins. This plug-in specialises in collecting the results and performance metrics from the quality test runs of the application.
- A set of sample jobs, which can be easily adapted and reused in the actual projects, which use DICE tools.

4.2.2. Tool usage

Once the Administrator installs a Jenkins service with the DICE plug-in installed and configured, the DICE Continuous Integration tool can be used like any standard Jenkins tools. The users log into the portal, and the more privileged users can then create continuous integration jobs, specifying in each job the details such as the source of the code in a versioning system of choice (e.g., a Subversion or a GIT server), the repository and the branch to be monitored. They also specify the sequence of actions, which can be copied from one of the DICE sample jobs.

When a job is set, the developers can then use the system as normal: by committing their changes to the Versioning Control System such as Subversion, Jenkins will execute the job. The outcome of the job execution gets recorded in Jenkins.

The users can then visit the details page of the job, which shows the history of the past job executions. Figure 7 shows an example view. As the example shows, the view shows a chart of the selected metric as well as the tabular representation of the build history. If multiple performance metrics are available for the builds, the links under the chart enable switching between them.

Deliverable 1.1. State of the art analysis.

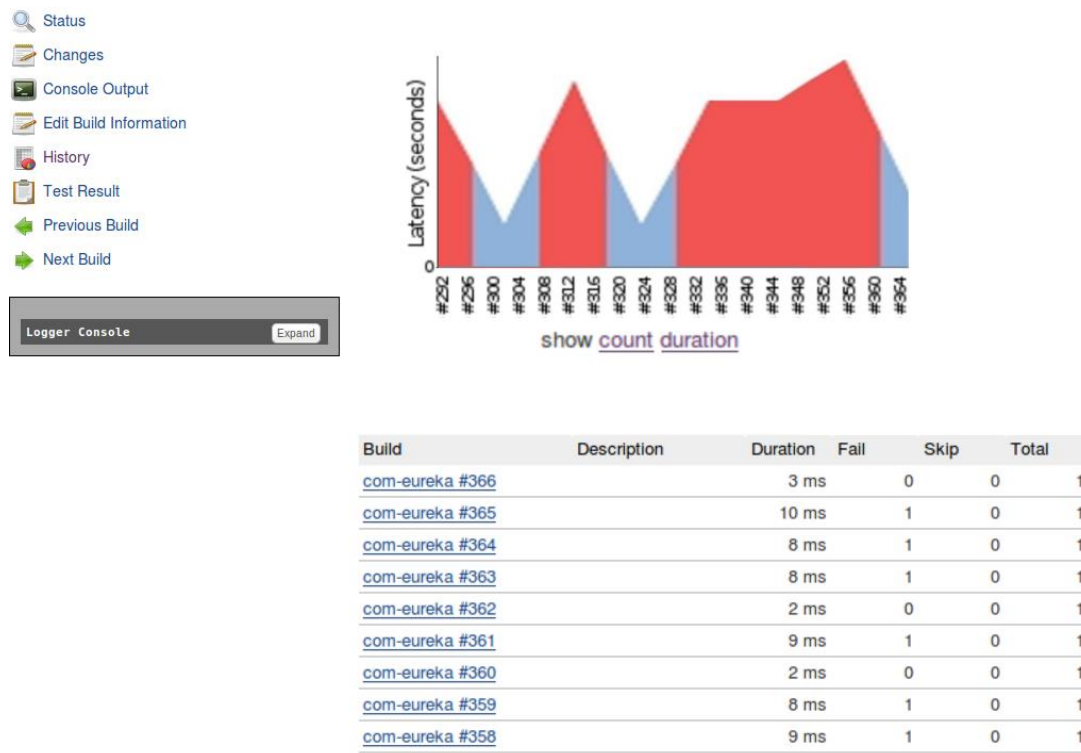



Figure 7: DICE plug-in for Jenkins displays a history of the measured performance metrics (e.g., latency) as a chart, and the history of the builds as a table


Configuring of the DICE Continuous Integration plug-in is possible in the post-build actions of the job settings, as Figure 8 shows. The administrator can provide the path pattern in the application's workspace where the test tools deposit the results. The "Retain long standard output/error" option enables or disables preserving and displaying in the job history of all the job's output. The Health report amplification factor influences how the overall job's health is computed.


Post-build Actions

☒ **Publish JUnit test result report** 

Test report XMLs

Fileset 'includes' setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is [the workspace root](#).

☒ Retain long standard output/error 

Health report amplification factor 

▼

Figure 8: Options available when configuring a continuous integration job for the DICE Continuous Integration plug-in

Health of the build is represented as a percentage, with 100% meaning a fully stable job, and 0% representing a fully failing job.

- A factor of 0.0 will disable the test result contribution to build health score.
- A factor of 0.1 means that 10% of tests failing will score 99% health.
- A factor of 1.0 means that 10% of tests failing will score 90% health. This is the default setting.
- A factor of 5.0 means that 10% of tests failing will score 50% health.
- A factor of 10.0 means that 10% of tests failing will score 0% health.

The use of the jobs in Continuous Integration will effectively be asynchronous. This means that either the user or a software client (e.g., an IDE plug-in) initiates a long-running job, but the action doesn't block any other workflow to wait for the job to be finished. Instead, the Continuous Integration tool enables the retrieval of the current status of the job (e.g., scheduled, running, finished), so that the client should periodically or occasionally poll for the status.

4.2.3. Obtaining Continuous Integration Tool

The DICE Continuous Integration tool consists of the standard Jenkins [21] install and a plug-in, which is available at the GitHub:

- <https://github.com/dice-project/DICE-Jenkins-Plugin>

4.3. Configuration Optimization Tools

Once a data-intensive system approaches its final stages of deployment for shipping to the end users it becomes increasingly important to tune its performance and reliability [24]. This is a time-consuming operation, since the number of candidate configurations can grow very large. Configuration optimization tool guides this phase in order to find optimum configuration to be set for the real system. There are currently a number of mathematical methods and approaches that can be used in the search for optimal configuration. The Configuration optimisation tool presented below employs BO4CO [25], a machine learning algorithm that facilitates iterative search for the best configuration settings. The description of the tool and its behaviour, along with potential use cases and initial experimental results is presented in the sections below.

4.3.1. Main components of Configuration Optimization Tool

The high-level structure of the configuration optimization tool consists of three separate components, as shown in Figure 9: (i) configuration optimizer (the machine learning algorithm), (ii) configuration testing (the coordinator), (iii) performance repository (for storing performance data that feed the algorithm). The tester specifies the parameters of interest and possible values for each parameter. The configuration optimization tool then based on currently available performance measures select the next configuration of the stream topology to be tested by the configuration testing component. Once the performance of the topology with the currently selected configuration has been measured, it will be stored to the performance data repository in order to be retrieved by the BO4CO algorithm. This sequential process (of configuration selection and experimental measurement) will be continued until the maximum number of allowed test is performed.

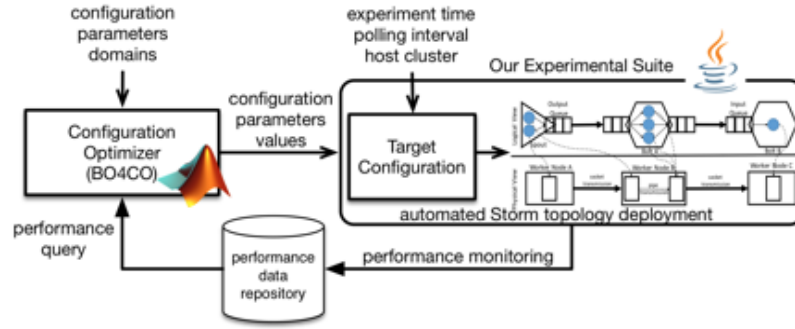


Figure 9: Configuration optimization architecture

4.3.2. Behaviour of the Configuration Optimization Tool

Logic of the tool is iterative (left part of Figure 10). The configuration optimizer automatically selects a configuration at each iteration employing BO4CO [25] that determines the best configuration to test next in the procedure. BO4CO estimates the response surface [26] of the system using observed performance data. It selects the next configuration to test using the estimated data searching for points that has a good chance of being the optimum configuration.

The configuration testing then automatically deploys the topology on a testbed (e.g., Storm cluster), then the end-to-end performance of the topology determined by the configurations is measured and stored in a data repository to be used by configuration optimizer. Configuration optimizer exploits the historical data to fit a model in order to decide which configuration to select next. Finally, after the predetermined number of iterations, the optimal configuration with the lowest latency or highest throughput depending on the user preference is selected as the main output of the tool. Note that the internal machine learning model will be then used for performance configuration for regions that we have not performed any experiments. In the results section, we provide experimental results comparing the prediction power of our internal model with polynomial fits.

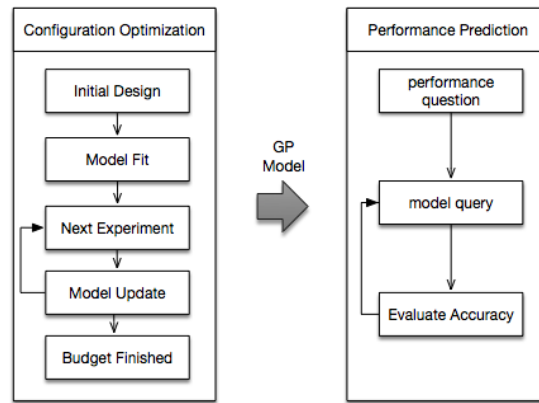


Figure 10: Internal behaviour of configuration optimization tool

Internally, the configuration testing automatically deploys (via DICE deployment tool) a topology on a multi-node cluster, like the one represented in Figure 11 with 1 Nimbus node, 2 Zookeeper nodes and 3 Supervisors. 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 configurations are then set in the template by appropriate values. In order to perform model fitting, the tool requires to retrieve the performance data and

augment new points in the repository. This performance data serves 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. We are also able to automatically communicate the configuration parameters that needs to be changed and its associated values through the configuration optimizer. The DICE monitoring tool also provides end to end throughput and latency measurements.

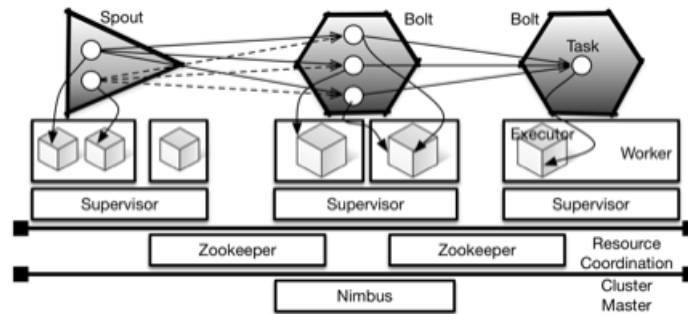


Figure 11: Storm architecture

4.3.3. Configuration Optimization Tool usage

The tools usage is as follows:

1. Tester provides a list of configuration parameters and potential (exhaustive) set of configuration options for each parameter in a YAML file. 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.
 - a. The tool then starts the test by retrieving the configuration template from model repository.
 - b. Prepares the testing scripts and the test bed and loads the historical data from data repository.
 - c. 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:
 - i. It deploys the configuration by specific parameter settings by using CI and DS tools.
 - ii. It builds and runs the topology on the test bed.
 - iii. It queries the monitoring and augments the experimental data to the data repository.
 - iv. It 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).
3. The configuration optimization tool then provides the optimum configuration in different formats: (i) text file, (ii) xml file, (iii) console, (iv) html format, (v) directly to the YAML configuration file associated to the running topology on the cluster it is running for end users.

4.3.4. Results

We performed several experiments with different Apache Storm topologies. WordCount (Figure 12) is an Apache Storm topology that we used in our experiments. The objective of the experiment is to show that configuration optimization tool is able to find the optimum configuration with few iterations.

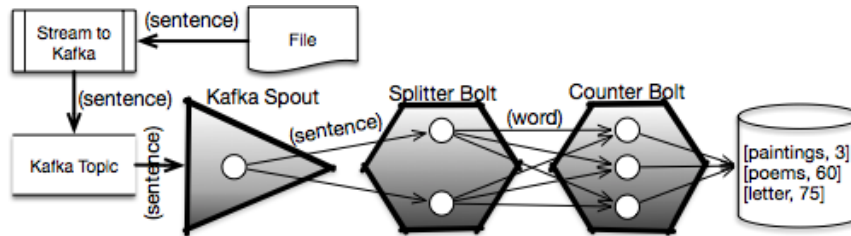


Figure 12: WordCount topology

We considered 13 parameters in our experiments mainly because they affect the latency considerably. However, configuration optimization tool is not dependent on these parameters and can work with any Storm based configuration [25][25].

- `topology.max.spout.pending`. The maximum number of tuples that can be pending on a spout.
- `topology.sleep.spout.wait.strategy.time.ms`. Time in ms the SleepEmptyEmitStrategy should sleep for.
- `storm.messaging.netty.min_wait_ms`. The min time netty waits to get the control back from OS.
- `spouts, splitters, counters, bolts`. The level of parallelisms of different Processing Elements (PEs).
- `heap`. The size of the worker heap.
- `storm.messaging.netty.buffer_size`. The size of the transfer queue between Storm deployment nodes.
- `topology.tick.tuple.freq.secs`. The frequency at which tick tuples are received.
- `top_level`. The length of a linear topology.
- `message_size, chunk_size`. The size of tuples and chunk of messages sent across PEs respectively.

Figure 13 shows the improvements of Storm topology end to end latency (the difference between the timestamp once the job arrives to the topology and the time it has been processed and leaves the topology) when we find optimal configuration comparing with the default values, note that the default values [22] is typically used in Storm topologies. This shows an improvement of 3 orders of magnitudes comparing with the default values. The results show that the tool finds the optimum configuration only within first 100 iterations. The full factorial combination of the configuration parameters in this experiment is 3840 and 100 experiments is equal to 2% of the total experiments. Note that in order to measure the difference between the optimum configuration found by the configuration optimization tool we performed the full factorial experiments (i.e., 3840 experiments each for 8 minutes over $3480 \times 8 / 60 / 24 = 21$ days).

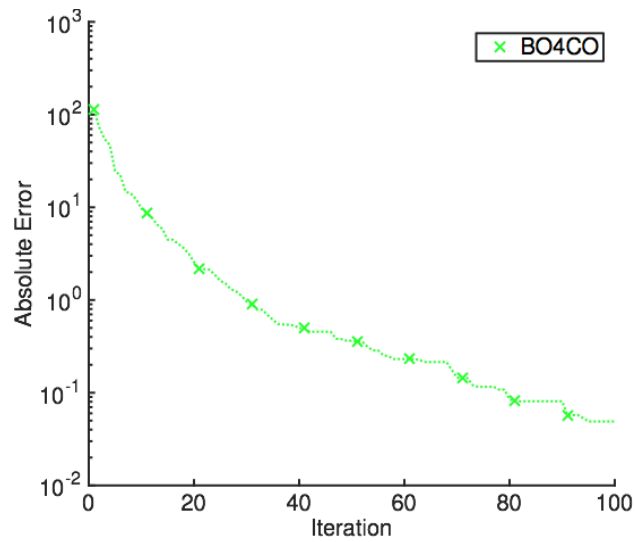


Figure 13: Distance to optimum configuration, error is in milliseconds

4.3.5. Obtaining Configuration Optimization Tool

The configuration optimization tool will be provided at the following address:

- <https://github.com/dice-project/DICE-Configuration-BO4CO>

5. Conclusion

Teams developing data-intensive applications inevitably have to deal with stacks of third-party services of varying complexity. A case in point is the Lambda architecture exemplified in the Oryx 2 framework, which is composed of at least 5 different services. Their set-up and configuration can be intimidating to inexperienced teams, and can thus be a deterring factor when deciding for the application's underlying technology or experimenting with various topologies. As a result, the applications created in a traditional way have a long Time-to-Market (TTM) stage.

Modern software market is highly competitive thus necessitating the quick delivery of the quality product. Therefore, it is no longer enough to build an application that is eventually brought to the environment where it can be tested, validated and, ultimately, used. Instead, it is important that the application is deployed in the test bed as soon as possible. That is, as soon as offline tools such as compilers, simulation tools and optimization declare the design and code to be valid. The best approach to achieve this is to streamline the whole development process. In this way, the developers feel as little traction as possible.

The DICE Deployment Tool and the DICE Continuous Integration Tool resolve the problem of complexity of deploying stacks of services in a test bed as well as the need to carry these tasks frequently. Our DICE Deployment Tool readily consumes the TOSCA documents, which contain the application's topology description. By using the supported building blocks' node types and cookbook recipes, the application deployment is a trivial matter.

Bringing the application to a working state is, then, only one part of the goal when designing data-intensive applications. It is then equally important that the application performs efficiently. It is often a matter of tuning the configuration values of the various services to the application and the deployment topology. Rather than leaving the application to the default parameters set in the Big Data frameworks such as Apache Storm, Apache Spark or Apache Hadoop, DICE framework includes the Configuration Optimization Tool. This tool is the result of an on-going research, involving suitable optimization techniques and machine learning approaches such as the BO4CO. Given a number of constraints such as time, the tool finds the optimum configuration under which the application performs with lower latency and higher throughput compared to default settings.

The downside of the DICE Deployment Tool initial version is that it currently has no capability to perform incremental updates of the deployments. This means that even small changes in the configuration or the overall topology of the application require a full redeployment and reinstallation of the application. This can become time consuming. To speed up the redeployments, we will look into solutions for shortening the most expensive operations (such as extensive package downloads and repository updates) by using caches or whole machine snapshots.

Staying on the edge of the newest technology as a toolset means that the users of the DICE tools will also be able to experiment with, experience and take advantage of the trending data storage and processing engines. This is why we will support additional technologies to be available with the upcoming versions of the DICE delivery and configuration tools.

5.1. DICE Requirement compliance

In the Section 2 we provided a summary of the requirements. The Table 1 indicates the level that the DICE Delivery Tools comply in their initial release. The *Level of fulfilment* column has the following values:

- ✗ - not supported in the initial version yet
- ✓ - initial support
- ✓ - medium level support
- ✓ - fully supported

Table 1: Level of compliance of the initial version of the DICE delivery tools with the initial set of requirements

Requirement	Title	Priority	Level of fulfilment
R5.3	Continuous integration tools deployment	MUST	✓
R5.4	Translation of TOSCA models	MUST	✓
R5.4.1	Deployment plan support	MUST	✓
R5.4.7	Deployment of the application in a test environment	MUST	✓
R5.4.8	Starting the monitoring tools	MUST	✓
R5.5	User-provided initial data retrieval	MUST	✓
R5.7	Data loading support	MUST	✗
R5.16	Provide monitoring of the quality aspect of the development evolution (quality regression)	MUST	✓
R5.19	Deployment configuration review	MUST	✗
R5.20	Build acceptance	MUST	✓
R5.27	Recommender Engine and Optimization	MUST	✓
R5.27.2	Recommender Engine API	MUST	✓

As a part of our future work, we will continue to work towards fully supporting the requirements. In particular:

- R5.4 is an on-going collaboration between WP5 and WP2 to ensure that the model-to-text transform, which transforms the DDSM into a corresponding TOSCA document, produces a workable TOSCA blueprint.
- R5.4.8 will be the effort with the WP4 at the beginning of year 2.
- R5.5 is in the process of being defined and should be implemented in year 2.
- R5.7 depends on the definition for R5.5.
- R5.16 is in the stage of being tested and should be validated by the use cases in the beginning of year 2.
- R5.19 will be designed in year 2.

Deliverable 1.1. State of the art analysis.

- R5.27 is subject of integration, which is planned for M18.

References

- [1] DICE consortium, DICE deliverable 1.1: State of the Art Analysis, July 2015
- [2] DICE consortium, DICE deliverable 1.2 Requirement Specification, July 2015
- [3] DICE consortium, DICE deliverable 1.3 Architecture definition and integration plan - Initial version, January 2016
- [4] Cloudify: <http://getcloudify.org/>
- [5] Chef <https://www.chef.io/chef/>
- [6] Infrastructure as a service manifest, <http://www.infrastructures.org/>
- [7] Derek Palma and Thomas Spatzier, *Topology and Orchestration Specification for Cloud Applications Version 1.0*. OASIS standard, November 2013 <http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html>
- [8] YAML <http://yaml.org/>
- [9] Flexiant Cloud Orchestrator: <https://www.flexiant.com/flexiant-cloud-orchestrator>
- [10] Apache Storm <http://storm.apache.org/>
- [11] Apache Zookeeper <https://zookeeper.apache.org/>
- [12] Apache Kafka <http://kafka.apache.org>
- [13] Apache Spark <https://spark.apache.org/>
- [14] Hadoop YARN, <http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html>
- [15] Puppet <https://puppetlabs.com/puppet/what-is-puppet>
- [16] Ansible automation tool for application deployment <http://www.ansible.com/home>
- [17] Ubuntu Juju: <http://www.ubuntu.com/cloud/tools/juju>
- [18] Apache Brooklyn: <https://brooklyn.incubator.apache.org/>
- [19] https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=camp#technical
- [20] Martin Fowler, *Continuous Integration*, May 2006, <http://www.martinfowler.com/articles/continuousIntegration.html>
- [21] Jenkins <https://jenkins-ci.org/>
- [22] <https://github.com/apache/storm/blob/master/conf/defaults.yaml>
- [23] Balalaie A, Heydarnoori A, Jamshidi P., *Microservices Architecture Enables DevOps: an Experience Report on Migration to a Cloud-Native Architecture*. IEEE Software, 2016.

- [24] Andreas Brunnert, 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).
- [25] Pooyan Jamshidi, Giuliano, Casale, *An Uncertainty-Aware Approach to Optimal Configuration of Stream Processing Systems*, ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC), 2016, under evaluation.
- [26] A. Khuri, J.A. Cornell, S. S. Sabiani: *Response Surfaces: Design and Analyses, revised and expanded*, 1996.