Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

DICE delivery tools – Intermediate version

Deliverable 5.2

Deliverable:	D5.2
Title:	DICE delivery tools - Intermediate 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), Damian Andrew Tamburri (PMI)
Reviewers:	Diego Pérez (ZAR), Alberto Romeu (PRO)
Type (R/P/DEC):	Demonstrator
Version:	1.01
Date:	11-April-2017
Status:	Final
Dissemination level:	Public
Download page:	http://www.dice-h2020.eu/deliverables/
Copyright:	Copyright © 2017, DICE consortium – All rights reserved

DICE partners

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

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

Executive summary

This report accompanies the intermediate release of the DICE delivery tools: DICE Deployment Tool, DICE Continuous Integration and DICE Configuration Optimisation. The purpose of these tools in the DICE methodology is to create a runtime of a DIA described in a DDSM / TOSCA blueprint, provide scheduled or on-commit execution of complex automated tasks on top of the DIA, and offer recommendation for the optimal configuration for the DIA's deployment.

Improvements of the DICE Deployment Tool since Y1 include a new web graphical user interface, extended TOSCA technology library and removal of Chef Server dependency. The Administrators now can set in the central service the parameters that describe the platform. This lifts a burden from developers, who can now focus on managing application-related parameters. Resulting TOSCA blueprints will work on any supported platform without change in any of the parameters or structure. Users can also decide to enable automatic connection of Storm, Spark or Cassandra node to DICE Monitoring. The DICE TOSCA technology library now supports also Zookeeper, Kafka, HDFS, YARN and bash or Python custom scripts.

The DICE Continuous Integration's Jenkins plug in is now compatible with Jenkins version 2 and is visually improved. We also provide a number of templates and instructions on how to schedule various DICE tools to process user's DIAs. This demonstrates how the DICE Continuous Integration can serve as a data repository, storing and serving version-bound or build-bound data for subsequent runs of the tool.

The DICE Configuration Optimisation is the tool for empirically arriving at the application configuration, that offers an optimal performance. We upgraded the original approach, BO4CO, which exploits Gaussian Process, and introduced the Transfer Learning for Configuration Optimisation (TL4CO). We can now take advantage of the observations from one version of the DIA to quickly arrive at better results in another version of the DIA. We validated the tool by optimizing Hadoop configurations to achieve optimal performance of two Hadoop jobs, WordCount and TeraSort. We also optimized Cassandra to have an improved performance for a mix of read and write operations comparing to the default configuration.

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
BO4CO	Bayesian Optimisation for Configuration Optimisation
TL4CO	Transfer Learning for Configuration Optimisation
DIA	Data Intensive Application
HDFS	Hadoop File System
GUI	Graphical User Interface
VCS	Version Control System
VM	Virtual Machine
JSON	JavaScript Object Notation
YAML	YAML Ain’t Markup Language[1]

[1] http://yaml.org/

Table of contents

List of Figures

List of Tables

1 Introduction

The DICE methodology provides to the DevOps [1] teams a full cycle of DIA development, including: design, off-line simulation and analysis, configuration, deployment, monitoring, enhancement and anomaly detection steps. In this methodology, the deployment and configuration tools represent the crucial enablers for transitioning from design to a runtime of the DIA. In this report we provide the details on these tools in their intermediate version, which we produced at the end of Y2.

Like in the initial release [2], this package consists of Delivery Tools, which comprise **DICE Deployment Tool** and **DICE Continuous Integration**, and **DICE Configuration Optimisation**. These tools aim to a) provide a simple, reliable and repeatable way of deploying DIAs of various complexity, b) provide recommendations about the best configuration for the DIA's deployment, and c) give insight on the performance improvements or regressions of the DIA.

This report covers the effort of the DICE project's WP5, specifically T5.1 Deployment plan execution and T5.2 Continuous integration. The document is a Y2 update of the Y1's D5.1 DICE delivery tools – Initial version [2]. We plan to release the final report in M30 in D5.3 DICE delivery tools – Final version.

In the rest of this section we summarize the changes and improvements since Y1. The Section 2 summarizes the requirements for the DICE delivery tools, extracted from the D1.2 [3]. In the Section 3, we present the top-level architecture of the delivery and configuration tools. In the Section 4 we present each tool in a deeper technical level, also presenting their usage and validation results. Finally, in Section 5 we present the conclusions.

1.1 What is new in Year 2

Updates in Y2 to the presented components were an outcome of a development process, which aimed at a) addressing any open or partially addressed requirements, b) implement any new requirements, which have arisen from early feedback from the use case providers, and c) to improve general stability and usability of the tools.

1.1.1 DICE Deployment Tool

Building Big Data clusters has traditionally involved manual process of setting up and configuring individual nodes and services, which is a lengthy process. With the DICE Deployment Tool, we aim to make the process of deploying DIAs as simple and painless process as possible. Using OASIS TOSCA as the format of DIA blueprints, we promote the principle of Infrastructure as Code, reducing issues of handling and maintaining complex systems to maintaining high-level "configuration" of the applications in a version control system.

New features and properties of the DICE Deployment Tool include the following:

- Integration with the DICE Deployment Modelling Tool (DICER), which effectively enables GUI for assisted visual modelling of the deployments and will be reported in M27 in D2.4
- Graphical user web interface, which provides a visual and intuitive way for using most of the DICE Deployment Tool's functionality.
- Improved support of the existing DICE TOSCA Technology library technologies (Storm, Zookeeper, Cassandra) and support for new technologies (Spark, HDFS, YAML, Kafka).

- Based on the needs of the use case providers, we also supported a node type for running scripts, enabling basic installation and running options for any other not yet supported technology.
- Chef Server is not needed anymore, because we have modified the Chef plug-in in Cassandra to use local Chef-Zero mode.

1.1.2 DICE Continuous Integration

The Continuous Integration service provides regular, scheduled or event-based running of tools that are important in the DIA development, testing and deployment lifecycle. In the DevOps approach it is an important element for early feedback of any functional or performance issues with the application being developed. In DICE, we upgrade its functionality to also store and visualize application's performance through development history. We also use it as a glue between deployment, configuration, quality testing and enhancement steps.

In Y2, we have applied the following updates:

- Integration with DICE Configuration Optimisation
- Added templates for future integration with Quality Testing, Enhancement Tools
- Compatibility with Jenkins version 2.
- Improved appearance of the build history reports.

1.1.3 DICE Configuration Optimisation

The Configuration Optimisation (CO) tool provides a software mechanism to explore alternative configurations for a DIA and identify the optimal one with respect to a given performance metric (e.g., throughput, response time, ...). The initial version of this tool, presented in deliverable *D5.1 - DICE delivery tools - Initial version* [2], is based on an algorithm, called BO4CO, which drives the search for an optimal configuration using a technique known as Bayesian Optimisation, which can cope with variability in the measurements and allow to customize the optimal trade-off between exploitation of existing measurements and exploration of new configurations. A large-scale validation has been performed for Storm-based DIA.

In Y2 we have further extended the CO tool and its validation

- Transfer learning algorithm (TL4CO) to reuse measurements for an old version of the application upon tuning the performance of a new version.
- Integration and test of CO against Apache Hadoop and Apache Cassandra.
- Validation of BO4CO and TL4CO against the application used in the ATC case study.

2 Requirements

In Deliverable D1.2, the Y1 update [3], we presented the requirement analysis for the DICE project. This section includes summaries of the requirements that we did not present in D1.2. Also, we have updated the R5.4.2's description to better fit the features and goals of the tool and added R5.43 to address privacy and security aspects. The actors involved include CI_TOOLS, which represent the DICE Continuous Integration tools, and the DEPLOYMENT_TOOLS, which represent the DICE deployment tool.

ID	R5.4.2
Title	Translation tools autonomy
Priority	Must have
Description:	The DEPLOYMENT_TOOLS MUST take all of its DIA-related input from the DDSM, which directly translate into the TOSCA model, or from the ADMINISTRATOR set values. Therefore it MUST NOT require any additional user's input in an interactive way.

ID	R5.4.5
Title	Deployment tools transparency
Priority	Should have
Description:	The DEPLOYMENT_TOOLS SHOULD NOT require from ADMINISTRATOR to take part in any individual deployment.

ID	R5.4.6
Title	Deployment plans extendability
Priority	Could have
Description:	The DEPLOYMENT_TOOLS MAY be extended by the ADMINISTRATOR with other building blocks not in the core set.

ID	R5.7.1
Title	Data loading hook
Priority	Should have
Description:	DEPLOYMENT_TOOLS SHOULD provide a well-defined way to accept the initial bulk data that they can load.

ID	R5.27.1
Title	Brute-force approach for CONFIGURATION_OPTIMIZATION deployment
Priority	Should have
Description:	CONFIGURATION_OPTIMIZATION SHOULD apply intelligent ML methods in order to enable a sequential

	decision making approach that selects a promising configuration setting at each iteration. CONFIGURATION_OPTIMIZATION should find the best possible configuration at the end within the

ID	R5.27.6
Title	CONFIGURATION_OPTIMIZATION experiment runs
Priority	Must have
Description:	CONFIGURATION_OPTIMIZATION MUST be able to derive the experiment by running the application under test with specific configuration setting by contacting DEPLOYMENT_TOOL. CONFIGURATION_OPTIMIZATION MUST be able to retrieve the monitoring data regarding the experiments by contacting MONITORING_PLATFORM.

ID	R5.27.7
Title	Configuration optimisation of the system under test over different versions
Priority	Should have
Description:	CONFIGURATION_OPTIMIZATION SHOULD be able to utilize the performance data that have been collected regarding previous versions of the system under test in the delivery pipeline.

ID	R5.27.8
Title	Configuration Optimisation's input and output
Priority	Must have
Description:	CONFIGURATION_OPTIMIZATION MUST be able to receive a TOSCA blueprint, which describes the application under test including any initial configuration. It MUST return a TOSCA blueprint updated with optimal parameters, or a stand-alone configuration file.

ID	R5.43
Title	Practices and patterns for security and privacy
Priority	Must have
Description:	The DEPLOYMENT_TOOLS MUST enable applying practices and patterns to ensure that the deployed application is reasonably secure and protecting privacy.

3 Architecture

3.1 High level architecture

The DICE delivery and configuration tools are the components in the DICE methodology, which consume the deployment model of the DIA and turn that model into its runtime application counterpart. The updated DICE architecture document [4] provides an overview across the whole project and also details interactions between all the DICE components, which are in a dependency relationship. Figure 1 below shows a zoomed-in detailed view of the architecture.

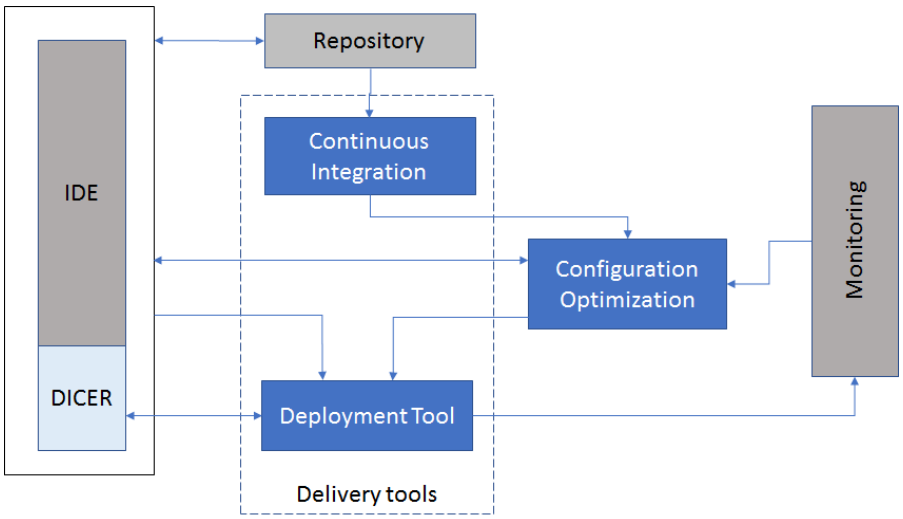


Figure 1: DICE delivery and configuration tools architecture. Delivery and configuration tools are represented in blue boxes, while external components are in grey boxes

In general terms, the architecture is in Y2 mostly the same as in Y1. One change worth mentioning is the addition of the **DICER** tool, which plays a double role in the delivery tools operations. In one respect, DICER is a part of the **IDE**, providing to users a graphical way to author DICE deployment diagrams (DDSM) and transform them into TOSCA blueprints. In another respect, the transformation into TOSCA works also from a command line tool or a service. Consequently, the **DICE Deployment Tool** accepts the DDSM models along with their TOSCA blueprint counterparts while the IDE and internally uses DICER to perform the transformation.

The role of **Repository** is now focused towards storing and versioning application code and models. The data artifacts such as those from **Configuration Optimisation**'s iterations data, are now stored in **Continuous Integration** with each respective build. The Configuration Optimisation no longer needs to interact with Continuous Integration service, because the interaction is now one-way from Continuous Integration, which is a client to Configuration Optimisation.

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:

- 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. DEVELOPER actors 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. They require deployment automation to occur on demand (occasionally), on schedule or with each new commit into the Version Control System (VCS). They occasionally require computation of the optimal configuration of their applications' topologies, but it is also beneficial to obtain periodic improvements of the configuration.

- 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.
- QA_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 Primer: DICE Deployment Modelling DICER

Partners in WP2 have been developing a tool called DICE Deployment Modelling, also known as DICER. The official report about the tool will only be available in Y3 of DICE. Considering that we refer to DICER in several places of the D5.2 report, we first make a short introduction to DICER.

In DICE methodology, we endorse DICE UML deployment diagrams for specifying the topology of the DIAs to be deployed. The DICE UML profile [6] applied to such diagrams results in a DICE Deployment Specific Metamodel (DDSM), which in turn is what the users will use when modelling their DIAs.

The DICER tool provides a GUI for visual modelling of DIA DDSMs. It is built as to provide assisted component-based infrastructure design, providing a complete palette of the DICE supported components and the ability to guide the user towards a complete deployment model. Figure 2 shows an example GUI view for DICER, after the user has requested a validation of an incomplete DDSM. The validation problems dialog provides the user an information about what to add next to the DDSM.

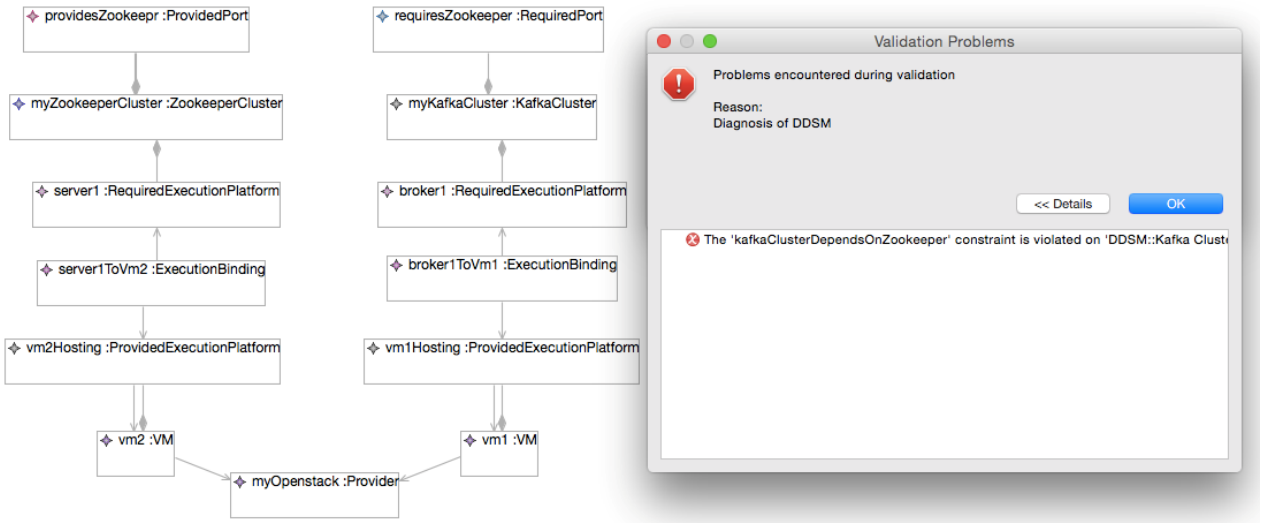


Figure 2: DICER's assisted component-based infrastructure design giving a validation warning

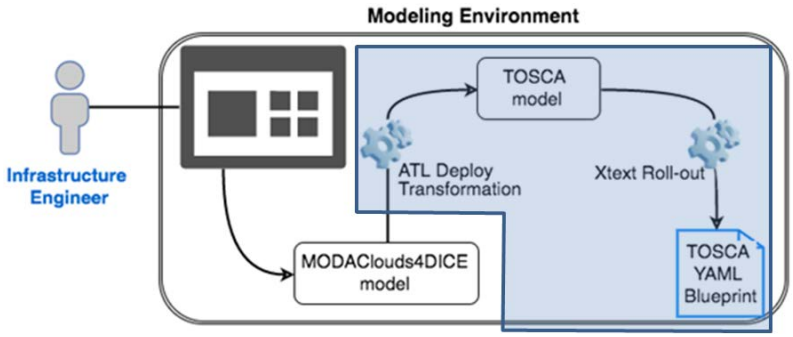


Figure 3: DICER tool workflow

The main benefit of the DICER tool is that it is capable of producing a TOSCA blueprint equivalent to the created DDSM. Figure 3 illustrates the stages that the model goes through before the TOSCA blueprint is generated. This, in turn, is an input of the DICE Deployment Tool.

4.2 DICE Deployment Tool

4.2.1 Main components

At the end of M24, the DICE deployment tool is a collection of the following components:

- DICE deployment service version 0.3.4
- DICE TOSCA technology library version 0.2.5
- DICE Chef Cookbooks version 0.1.9 (an extension of the DICE TOSCA technology library)
- Cloudify 3.4 (provided by the GigaSpaces).

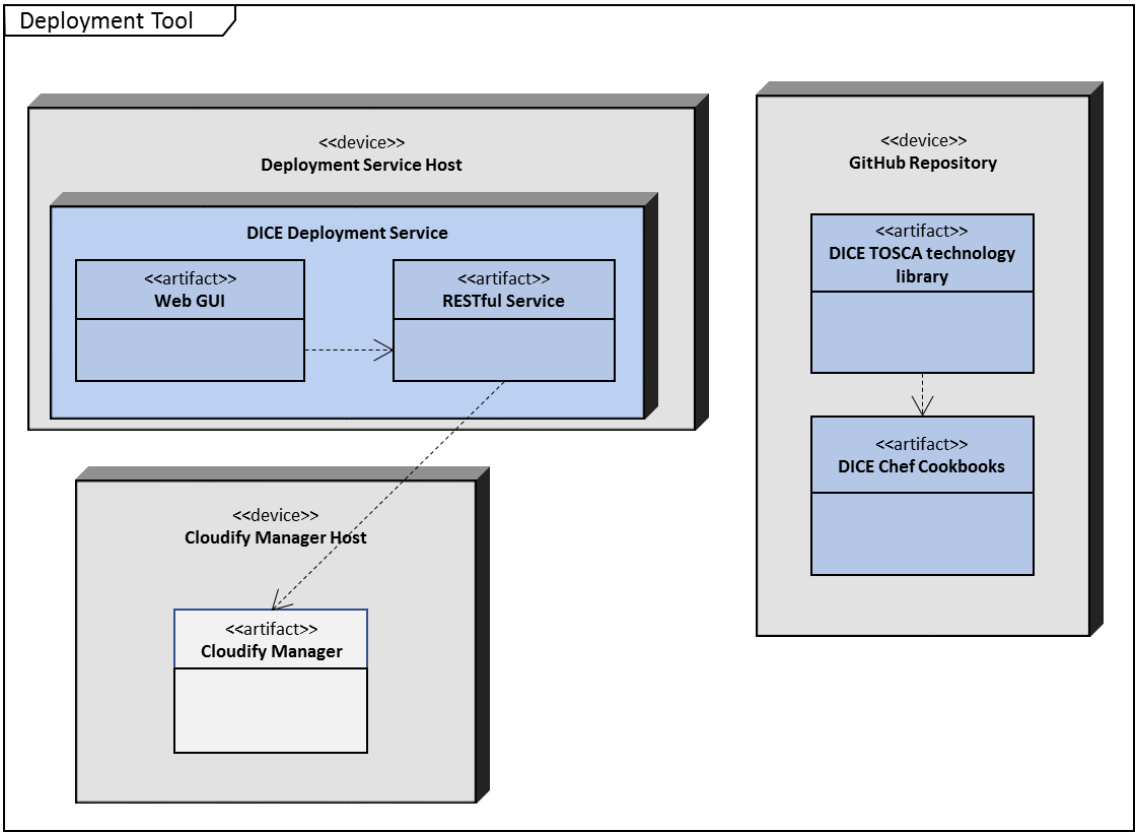


Figure 4: Deployment Diagram of the DICE Deployment Tool

Figure 4 represents a deployment diagram of the DICE Deployment Tool. The entities marked in blue are from DICE, while the others are from third parties.

4.2.1.1 Deployment Service

The Deployment Service is a RESTful web service, which provides an abstraction of API on top of the cloud orchestration engine Cloudify. The service is accompanied by a web user interface and a command line client. This enables a very versatile use of the tool, including:

- interactively via the graphical web interface,
- from the command line interactively or in scripts (e.g., testing and experimenting with deploys, automating deployments, inclusion in Continuous Integration, integration with or into text editors such as Sublime Text[2]),

[2] https://www.sublimetext.com/

- from web service clients (e.g., IDE plugins).

Since the initial version, we have implemented several improvements and extensions of the Deployment Service. An important update is **inputs management** in the Deployment Service. Inputs in TOSCA are parameters, which can vary between different deployments of the same TOSCA blueprint. Most of the blueprints require at least the inputs, which provide parameters of the testbed's platform, e.g. credentials to the resource provisioning API, identifications of virtual machine flavours and operating system images. Normally this is a set of parameters that an administrator sets once for each testbed, so they do not belong with the user's development space. It also should not be versioned with the application, but with the testbed's infrastructure configuration. The inputs management functionality in the Deployment Service therefore provides such a single point of setting up relevant inputs. Upon deployment, the service's logic then smartly assigns to the deployment only the inputs that the blueprint actually requests. This eases the use of the Cloudify's strict matching between the provided and the required sets of the inputs.

The Deployment Service also supports an **integration with DICE Monitoring Tool** [7]. The majority of configuration and interactions with DICE Monitoring Tool are within the TOSCA technology library and their related Chef Cookbooks, but Deployment Service can be instructed to register an application with the DICE Monitoring Tool. This marks an execution runtime of an application and its deployment metrics for any downstream services, which need to analyse the metrics from the DICE Monitoring Tool.

Considering that the DICE Deployment Service is a central service, the Administrator needs to install it in or at the target test bed.

4.2.1.2 TOSCA technology library

The DICE TOSCA technology library is a collection of TOSCA node and relationship types, Cloudify plug-ins and references to DICE Chef Cookbooks. They enable generating or authoring TOSCA blueprints for DIAs without having to explicitly specify any scripts or configuration procedures for installing the DIA components. In its version 0.2.5, support for the following technologies is available:

- Zookeeper[3]
- Apache Storm[4], including user's Storm topology
- Apache Spark[5] in a stand-alone mode, including user's Spark topology
- Apache Cassandra[6]
- Apache Kafka[7] (without any relationships)
- Hadoop File System[8]
- Apache YARN[9]
- User's custom script (in bash or Python)

[3] https://zookeeper.apache.org/
[4] https://storm.apache.org/
[5] https://spark.apache.org/
[6] https://cassandra.apache.org/
[7] https://kafka.apache.org/
[8] http://hortonworks.com/apache/hdfs/
[9] http://hortonworks.com/apache/yarn/

Additionally, the DICE TOSCA technology library abstracts the concepts for **a firewall** in the network and a compute node's network address, which is selected from a pool of addresses visible from the user's LAN (**a floating address**).

Most of the supported components require that the nodes hosting them have discoverable fully qualified domain names (FQDN). The only practical way of addressing this requirement is to use a DNS service. DICE provides a blueprint, which installs an open source DNS server on the node hosting the DICE Deployment Service, configured to handle only the dynamically created nodes. The TOSCA technology library then **installs a DNS agent** to handle the FQDNs of the nodes. This hugely simplifies work with the Big Data services.

The DICE TOSCA technology library itself is a Cloudify plug-in, hosted in GitHub. Therefore it requires no installation, because Cloudify fetches it with each deploy.

4.2.1.3 Chef Cookbooks

Chef in DICE is the chosen technology for configuration management of the nodes to be installed for the DIA. They work as a regular set of Cookbooks, but the recipes in each Cookbook are structured around the following phases in TOSCA orchestration:

- creation or installation of a service, which can be carried out for all services in the DIA independently from any other service,
- configuration of the node, which may require knowledge of other nodes that provide some capability that this node depends on,
- starting of a service.

While these Cookbooks may be used manually with a Chef client, they are better suited to be used by the Cloudify orchestrator. They too require no special installation, but get fetched by the Cloudify from a repository (e.g., GitHub) whenever they are required.

4.2.1.4 Cloudify

Cloudify is a cloud automation and orchestration engine. In DICE, we use it as a back-end for the actual TOSCA blueprint deployment. At the end of Y2, Cloudify is still the most functional and reliable solution for consuming TOSCA blueprints. The code base is maintained by a third party, the GigaSpaces.

Given that the code is open source, we were able to provide certain improvements to the existing functionality. First off, we provided code updates to the **OpenStack plug-in**, which enables that the Cloudify functions properly on top of the **OpenStack Mitaka** release. Our updates are now a part of the official Cloudify's development branch.

Additionally, we modified the **Chef plug-in** such that it **does not require Chef Server**, but always uses the Chef-Zero mode.

4.2.2 Tools usage

DICE Deployment Service provides logical deployment containers as receptacles of blueprints to be deployed. Figure 5 illustrates this concept: a blueprint submitted to a particular container will result in a deployment associated with that container. A new blueprint submitted to a logical deployment container, which has an already associated deployment, will result in a new deployment that will replace the previous one. In the figure, Blueprint B.1 has previously been deployed in

Logical Container 2. But then the users improved the application, resulting in the Blueprint B.2. After submitting this blueprint to Logical Container 2, the previous deployment has been removed and a new one installed. Users can create as many logical deployment containers as needed, for instance to use for personal experimentation of a new technology, specific branches in Continuous Integration, or for manual acceptance testing of new releases.

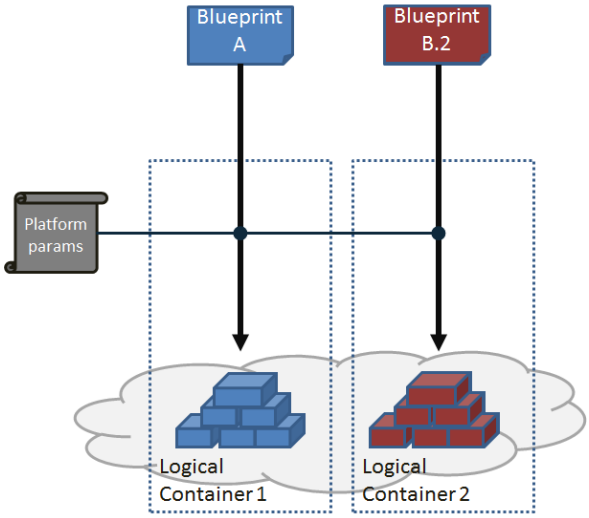


Figure 5: Illustration of Deployment Service's usage.

The general usage of the DICE Deployment Tool has remained the same as described in the previous report [2]. Normally, we would start with the **DICER** tool, where we create a DICE deployment model of our DIA. With DICER, we transform the deployment model into a corresponding TOSCA blueprint. Alternatively, we can create the blueprint by modifying the provided examples[10] or creating one from scratch.

A notable change from the blueprints in Y1 is that we can now refer to all the needed TOSCA plug-ins and imports from a single import line, as shown by Listing 1.

```
tosca_definitions_version: cloudify_dsl_1_3

imports:
  - http://dice-project.github.io/DICE-Deployment-Cloudify/spec/fco/0.2.5/plugin.yaml
```

Listing 1: Header of a TOSCA blueprint of a DIA that will be deployed in an FCO

As shown in the example, the required Cloudify DSL version of the TOSCA blueprint is now 1.3, which is an update from version 1.2 in Y1. A subpath of the `imports` line also visibly suggests that the blueprint will work on an OpenStack deployment. To target the blueprint to another platform, e.g. an OpenStack, simply replace `fco` with `openstack` in the URL and submit it to the DICE Deployment Service instance running in an OpenStack testbed.

We illustrate further benefits of the updated TOSCA library on Listing 2.

[10] https://github.com/dice-project/DICE-Deployment-Examples

```
StormMasterSecurityGroup:
  type: dice.firewall_rules.storm.Nimbus

StormMasterVM:
  type: dice.hosts.ubuntu.Medium
  relationships:
      - type: dice.relationships.ProtectedBy
        target: StormMasterSecurityGroup

StormNimbus:
  type: dice.components.storm.Nimbus
  properties:
    monitoring:
      enabled: true
  relationships:
    - type: dice.relationships.ContainedIn
      target: StormMasterVM
    - type: dice.relationships.storm.ConnectedToZookeeperQuorum
      target: ZookeeperCluster

WordCount:
  type: dice.components.storm.Topology
  properties:
    monitoring:
      enabled: true
    application: resources/wordcount.jar
    topology_name: dice-wordcount
    topology_class: org.apache.storm.starter.WordCountTopology
  relationships:
    - type: dice.relationships.storm.SubmittedBy
      target: StormNimbus
```

Listing 2: An example of a node template declaration in a blueprint for Storm

As illustrated, all the node template types and the relationship types are from the DICE deployment library and start with the "`dice.`" prefix. We have replaced the more generic node types and relationship types as well as the platform-specific ones with our own. For instance:

- `dice.firewall_rules.storm.Nimbus` represents a security group or a firewall that enables only the ports needed by Storm's nimbus to pass through. The user doesn't need to know any of these ports, because we set them in the library.
- `dice.hosts.ubuntu.Medium` will make the orchestrator instantiate a virtual machine of a medium flavour and with Ubuntu as its OS. The flavour specification and the base image to be used in provisioning this virtual machine have been set by the Administrator in the DICE Deployment Service, so again the user doesn't have to set any additional parameters or supply inputs.
- `dice.components.storm.Nimbus` type defines a Storm platform's nimbus (i.e., a Storm cluster master) node. Like in Y1, this makes the Cloudify use the Chef runlist declared in the library to install, configure and start the Storm's nimbus service and all the libraries that are required. The `StormNimbus` node template also has a property `monitoring` set to `{ enabled: true }`. As a result, this node will be connected to the DICE Monitoring Tool [4] at the address defined by the Administrators.
- `dice.components.storm.Topology` type enables that a user's own Storm application gets submitted to the Storm cluster defined in the blueprint.

- `dice.relationships.storm.SubmittedBy` is a relationship type, which tells Cloudify that it should use the target node as a submission point of the supplied application.

In Y1, the DICE Deployment Services supported submission of only the blueprint YAML file as an input for the deployment. This mode is still supported, and it works fine for the applications that either do not need any artifacts such as `.jar` files or data dumps, or if these artifacts are available for download at a network address. But for a more dynamic environment such as the IDE, where the compiled libraries can quickly change, it is more convenient to **bundle the blueprint** with all the additional resources and submit the bundle for deployment. Cloudify already supported this mode, and in Y2 we have enabled it for the DICE Deployment Service as well. Now we can reference `.jar` files with the user's Storm topology (in `dice.components.storm.Topology`), Spark application (in `dice.components.spark.Application`), the user's script file (in `dice.components.misc.ScriptRunner`) or any other similar resource as a file path in the bundle.

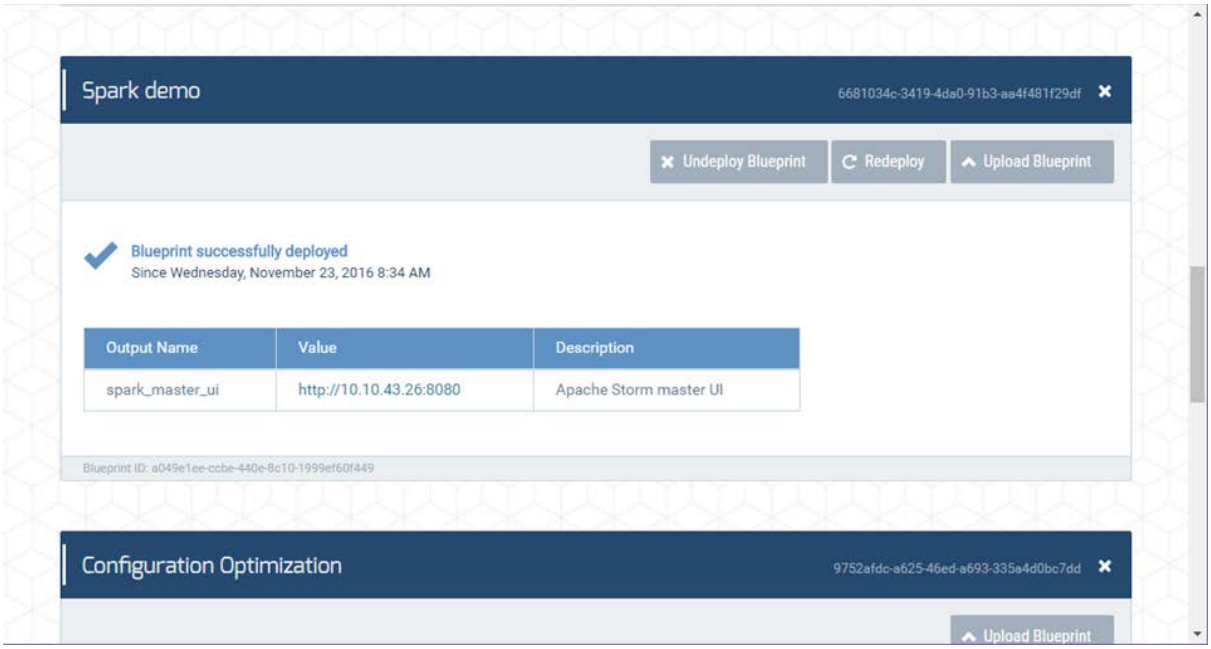


Figure 6: An example view of the DICE Deployment Service's web GUI.

Continuing from the example from the extract in Listing 2, the blueprint needs to be bundled into a `.tar.gz` file with the application `.jar` file. Listing 3 shows an example content of such a bundle.

```
└── wordcount
    ├── blueprint.yaml
    └── resources
        └── wordcount.jar
```

Listing 3: Contents of an example blueprint bundle for Storm word count

This bundle is ready to be submitted to a logical deployment container of choice. The deployment takes several minutes of time, the duration being dependent on complexity of the blueprint and available resources in the test bed. The **web GUI** will display the rough progress of the deployment.

When the deployment is finished, it will result in a view similar to that in Figure 6, where a table will display any outputs declared in the blueprint. In the shown example, the view contains an URL to the web user interface of the Storm's nimbus node that the user can click and navigate to the Storm cluster interface.

4.2.3 Validation and results

The goal of the validation is to prove that the tools function as intended. For the DICE Deployment Service, the expected behaviour can be summarized by a specification in the Gherkin[11] language [10] as shown in Listing 4. For the tool to be usable and reliable, the sequence described in the behaviour needs to work in the prescribed manner every time even after a high number of executions.

```
Given that a DICE Deployment Service instance is available at 'URL'
And the logical deployment container 'CONTAINER_UUID' is available
And I have a blueprint for a Storm application in '/home/matej/storm-wordcount.tar.gz'
When I submit my blueprint to container 'CONTAINER_UUID' at 'URL'
Then the deployment to 'CONTAINER_UUID' should succeed in less than 90 minutes
And the deployment's outputs should provide values for nimbus_url and topology_id
When I query for the topology with topology_id at nimbus_url
Then the Storm topology should be running
```

Listing 4: A specification for the expected behaviour of DICE Deployment Service when deploying a Storm application

The listed behaviour applies to a Storm topology, where we based the success criteria on the requirement that the Storm topology is live and running at the end of the sequence. More granular criteria such as the correct operation of the Storm topology is beyond the scope of the validation, because it already belongs to the functional testing of the application's logic itself.

We can devise similar behaviours for other supported technologies.

In our validation runs, we were able to consistently and reliably execute the required sequence. However, we did find that the reliability of the deployment strongly depends on the reliability of the underlying platform. If the test bed is too heavily loaded with tasks that take up a lot of I/O bandwidth, then parts of the deployment process would sporadically time out and fail. Also the platform manager needs to be able to reliably provision and release network addresses. We argue that in the first case, such a test bed is not usable for any serious use, because it would starve regular applications and processes as well. In the second case, the issue is also outside the scope of DICE.

Another important validation criterion for the DICE Deployment Service is one we have already expressed: the deployments need to succeed in a reasonable time. In Listing 4 we have set the upper limit to conservative 90 minutes. Compared to a traditional approach where we perform deployment of all services manually, this goal represents several levels of magnitude in improvement: hours instead of days or weeks. However, since we are aiming for a DevOps approach, the less time spent spinning up a new deployment means a faster feedback on the quality of the deployed application.

To quantify the time needed to deploy a blueprint, we ran repeatedly a selected set of blueprint deployments and timed their durations from submission into an empty logical deployment container until the DICE Deployment Service declared that the deployment has succeeded. We measured the

[11] https://cucumber.io/docs/reference

time required for complete teardown of a blueprint in similar manner: we took a fully deployed blueprint, started teardown procedure and measured the time until the DICE Deployment Service reported termination.

Table 1 shows the results. We ran the experiments on XLAB's internal OpenStack Mitaka. The Max level of dependence column represents the length of the longest path in blueprint's dependency DAG, not counting any virtual resources that are created almost instantly. For example, level 1 represents an application with services that are all peers and where none of the services depends on any other service. Level 2 represents a two-tier application, where a service (e.g., a database) needs to be deployed before another one (e.g., an application) can be deployed and configured. Figure 7 graphically shows these results, with the orange line representing median values, the boxes extend from the lower to upper quartile values of the data, and the whiskers extend over complete range of the data. The labels on the *x* axis correspond to the Code name column in Table 1.

Table 1: Summary and results of timing deployments

Blueprint	Code name	Technologies used	Max level of dependence	Deployment time [s]	Teardown time [s]
Storm WordCount	wordcount	Zookeeper, Storm	3	445	70
Spark Pi	sparkpi	Spark	3	442	55
Storm WikiStats	wikistats	Zookeeper, Storm, Cassandra	4	480	85
Hadoop cluster	Hadoop	Hadoop	4	550	65
Data pipeline	kafka-pipe	Zookeeper, Kafka, Cassandra, user scripts	4	510	70

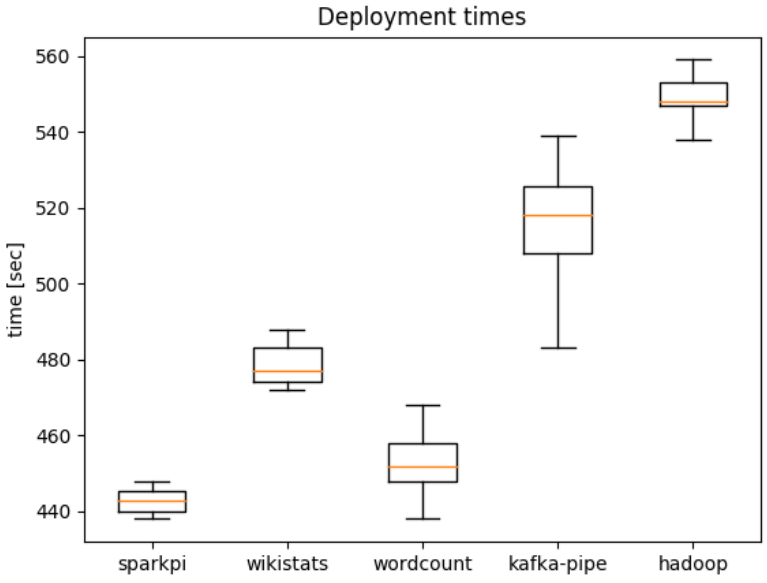


Figure 7: Blueprint deployment times

From the table, we can see that the deployment times are roughly correlated with the max level of dependence. The biggest range of the deployment time is for the Data pipeline due to intermittent delays introduced by the difficulties in our testbed.

Figure 8 illustrates further details on the timings of a particular deploy for Hadoop, where the scheduling due to dependencies is clearly visible. The node templates named `*_ip_*` and `*_fw_*` instantiate into OpenStack's floating IP and security group (firewall) instances, respectively, which takes a minimal time to perform. Then the `*_vm_*` nodes follow, bringing up the virtual machines. Then the service configuration can follow: first, the name node, which has to exist before any other nodes can work, following by the resource manager and data nodes, all in parallel. Finally, the orchestrator instantiates the node managers.

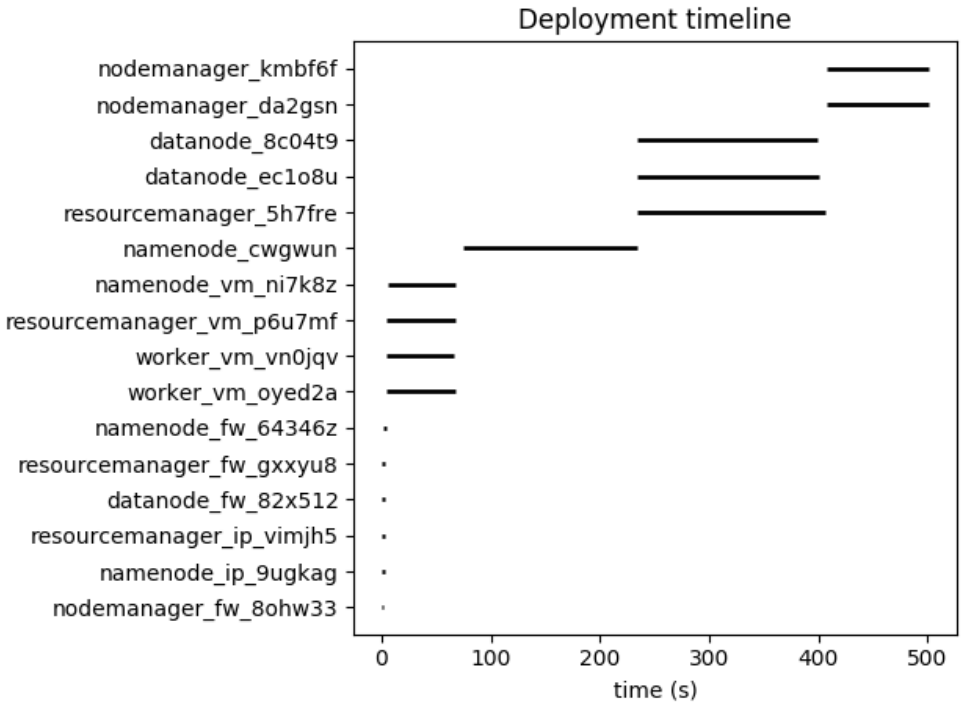


Figure 8: Apache Hadoop deployment timeline

4.2.4 Obtaining 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-Deployment-Examples – example blueprints for all the supported technologies.
- 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.

4.3 DICE Continuous Integration

4.3.1 Main components

The DICE Continuous Integration comprises the following components:

- Jenkins Continuous Integration[12], a popular and powerful open source solution, currently at version 2.36 (the long-term support version),
- DICE Jenkins plug-in, which collects quality testing performance results and visualises them in Continuous Integration project summary,
- sample job configurations for DICE tools.

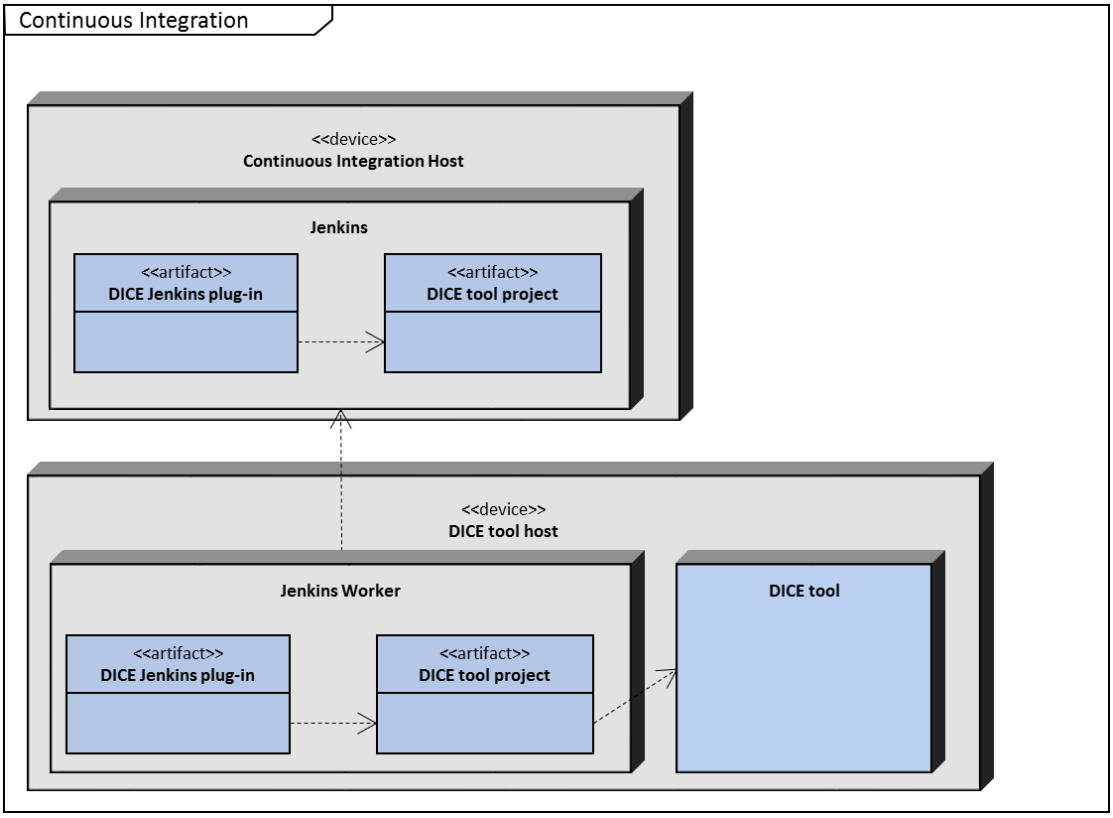


Figure 9: Deployment diagram of the DICE Continuous Integration

Figure 9 shows an example deployment of the DICE Continuous Integration. Components created by DICE are highlighted in blue. In this set-up, the Jenkins runs as a central service. The main Jenkins node is complemented by Jenkins Worker nodes, which are running on the remote nodes that are hosting DICE tools such as Configuration Optimisation on an Enhancement Tool. The DICE tool project on the diagram represents a generic Continuous Integration project, which runs the DICE tool on top of a user's DIA.

We provide additional details on the set-up in the following subsections.

4.3.2 Tools usage

Jenkins Continuous Integration runs as a central service, which has access to the test bed. The web user interface of Jenkins, in turn, has to be accessible to the users from their development environment. The Administrator also has to install the DICE plug-in.

[12] https://jenkins.io/

4.3.2.1 Build performance monitoring

The DICE plug-in is suitable for handling any Continuous Integration job, which results in a report on the application's performance. In DICE, both the Quality Testing tool and the Configuration Optimisation tool. The purpose of this exercise is to provide build performance monitoring, i.e., the ability to see the history of the general performance of the history of DIA's versions, e.g., average or maximum latency, batch processing time etc. In this context, we are not handling high-granular runtime performance monitoring, because that is available from DICE Monitoring Tool's Kibana service [4].

The user needs to first create a Jenkins freestyle type project, configured such that it is able to obtain from a VCS (Git, SVN, …) the application to be tested. The project needs to have one or more build steps, which execute the load testing of the application under test. The result of the build steps has to be a JSON file in a fixed path within the workspace. For example, the test might create a file `results/metrics.json` with contents shown in Listing 5.

```
{ "latency": 2.3, "throughput": 3 }
```

Listing 5: An example JSON file containing build's performance metrics

The project then needs to have a **post-build action** named **DICE's Quality check**. Figure 10 shows an example view of the setting, which will ensure that the DICE plug-in will capture the metrics output file after the end of each build run.

Figure 10: Configuring the DICE plug-in's post-build action in the build settings

We recommend to also add a post-build action typed Archive the artifacts. On the artifacts list add the `results/metrics.json` file and any output files that are a side product of the load generating tool. This is in particular useful if the next executions will benefit from results of the current run.

With these project parameters set, the project's job can now execute one or more builds. When the user then visits the project page in Jenkins, the result will be similar to the one on Figure 11. This view shows several interesting features:

- On the left side in the navigation list, the link **DICE graph** leads to a custom page described later in this section.
- On the right side of the page, a graph is showing past builds' results.
- In the middle, Jenkins provides links for the last successful build's artifacts (Last Successful Arifacts). We can see the `metrics.json` entry containing the previous (assuming it was successful) build's testing outputs. These artifacts have a well-defined URL, e.g.: http://jenkins-address.lan/jenkins/job/WikiStats-QT/lastSuccessfulBuild/artifact/mock-qt/results/metrics.json

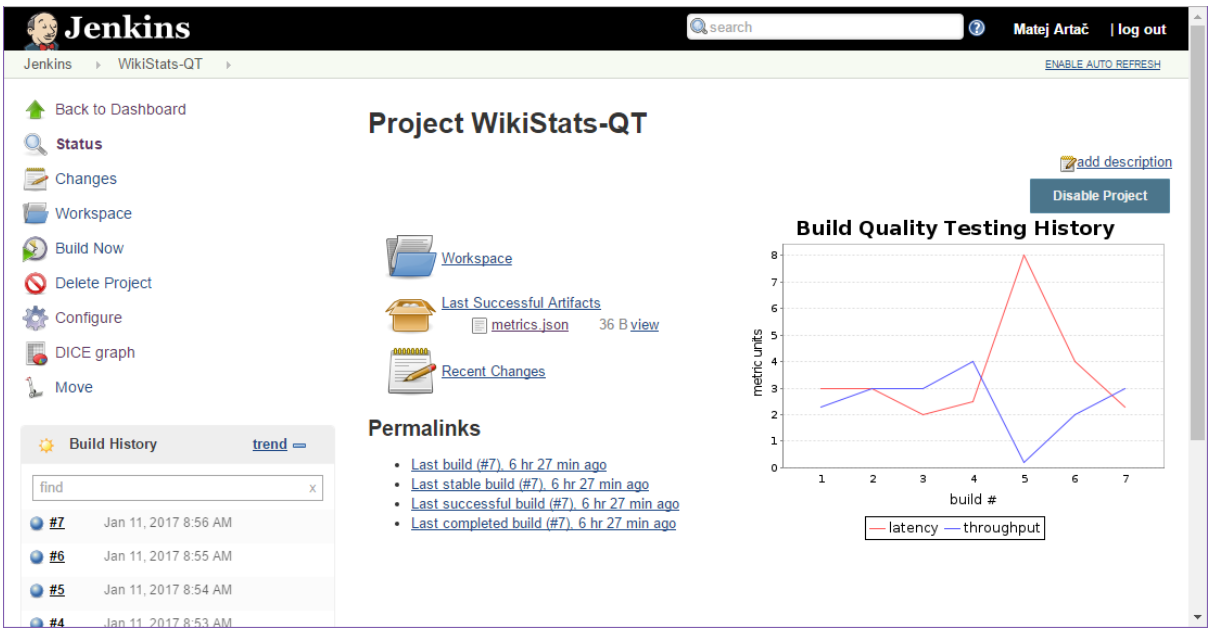
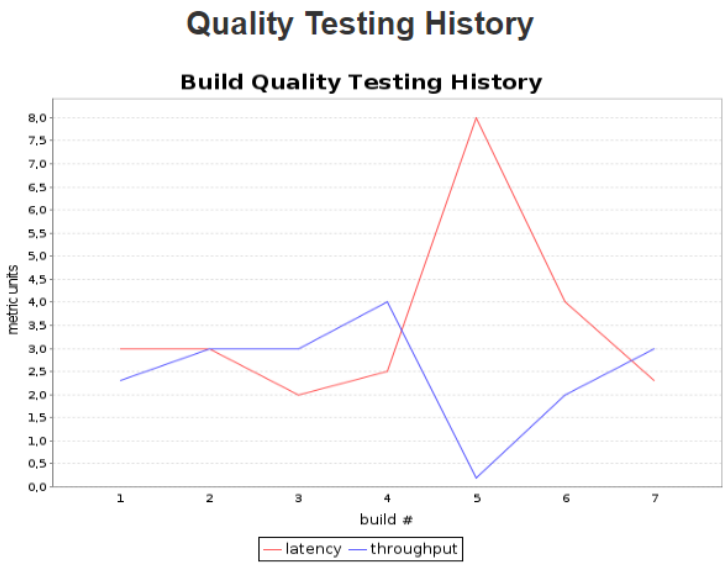


Figure 11: Example view of the project's status.

Clicking on the **DICE graph** link brings the user to a more detailed view of the history of the builds. Figure 12 shows an example of this view.



Build #	throughput	latency
7	3.0	2.299999952316284
6	2.0	4.0
5	0.20000000298023224	8.0
4	4.0	2.5
3	3.0	2.0
2	3.0	3.0
1	2.299999952316284	3.0

Figure 12: DICE Quality Testing history view in the Jenkins Continuous Integration.

4.3.2.2 Scheduling DICE tools

As already described, DICE Continuous Integration can serve as a glue between DICE components in the DICE DevOps methodology. In this section, we provide specific instructions on how to create projects that use other DICE tools. We assume that the central Jenkins node has already been installed and configured by the ADMINISTRATOR.

In setting up the DICE environment, ADMINISTRATOR installs the DICE tools (Configuration Optimisation, Enhancement Tools, etc.) on one or more dedicated nodes in the test bed. We recommend that the ADMINISTRATOR also installs a Jenkins worker on each DICE tools host. This will make the execution of the Continuous Integration jobs easier, because Jenkins will take care of transmitting all the necessary files from and to the main Jenkins service. Another benefit is that we can prepare the build step scripts as if they are being executed locally to the DICE tools installations (because they actually will be).

The ADMINISTRATOR then needs to add new nodes in Jenkins control panel and assign proper tags that reflect what tools are available on this node. This will enable filtering out the workers actually capable of running the tool relevant for the Continuous Integration projects.

For each of the DIA project (i.e., a branch in the VCS containing the DIA's development line) and for each DICE tool, the ADMINISTRATOR needs to create a new Jenkins project and set run restrictions based on the assigned tags from previous step.

Table 2 summarizes the main settings to be set for a Configuration Optimisation project. The Configuration Optimisation wrapper `run_bo4co.sh` optimizes configuration for selected blueprint and stores final result. These results are then archived by the Jenkins and displayed using DICE Quality Check Jenkins plugin.

Table 2: Continuous Integration project for scheduling Configuration Opt

Tool	Configuration Optimisation
Source Code Management	Configure source of input data for this project (blueprints, configuration, etc.).
Build step	Execute shell: `configuration_optimization/run_bo4co.sh $BLUEPRINT $PARAMETERS`
Post-build Action	Archive the artifacts: `results/*`
Post-build Action	DICE's Quality check: `results/summary.json`

Table 3 illustrates setting up a Quality Testing project. Quality Testing wrapper *run_qt.sh* runs selected application on variety of inputs and returns one or more quality metrics. These metrics are archived by Jenkins and displayed using DICE Quality Check plugin.

Table 3: Continuous Integration project for scheduling Quality Testing

Tool	Quality Testing Tool
Source Code Management	Configure source of input data for this project (blueprints, configuration, etc.).
Build step	Execute shell: `quality_testing/run_qt.sh $BLUEPRINT`
Post-build Action	Archive the artifacts: `output/result.json`
Post-build Action	DICE's Quality check: `results/summary.json`

We assume that the tool will expose a RESTful interface with a simple call for triggering the tool, which will then handle the VCS updates and commits on its own. For this type of tool, no local Jenkins worker is required, as long as the RESTful interface of the tool is accessible from the main Jenkins node. The Enhancement tool wrapper *run_et.sh* takes care of notifying the tool to start running, complete with the proper parameters. Table 4 illustrates a set-up of such a project to run such a tool.

Table 4: Continuous Integration project for scheduling Enhancement Tool

Tool	Enhancement tool
Build step	Execute shell: `enhancement/run_et.sh $APPLICATION_ID $SCM_REFERENCE`
Post-build Action	Archive the artifacts: `output/*`

4.3.3 Validation and results

Validation of the DICE Continuous Integration was informal in its nature. Our methodology was to create mock-ups of DICE services and scheduled a series of projects to run periodically and on-demand. We have visually inspected the results in the Jenkins web interface. The criteria for passing the validation were the following:

- It is possible to create Continuous Integration projects, which run long-running jobs on schedule or on VCS update.
- The build results are available for subsequent builds. They are also available for later download in the CI repository.
- Quality testing summary results are available as a graph and as a table. The history gets updated with each new build.

We successfully validated the tool on all criteria. This complements the validity checks of the behaviour of our DICE plug-in for Jenkins through test-driven development.

4.3.4 Obtaining DICE Continuous Integration

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

4.4 Configuration Optimisation

4.4.1 Main components

The DICE Configuration Optimisation is a single tool, which can be run from the command line. It externally relies on the DICE Deployment Service for deploying the applications that are subject to optimisation, and the DICE Monitoring Tool for collecting the metrics.

Internally, the Configuration Optimisation used BO4CO in its first iteration [2]. In the current release the use of the CO tool with the BO4CO algorithm introduced in [2] was extended to the Apache Hadoop-based DIAs. The objective was the same as for the Storm-based DIAs: to find optimal configuration settings for the application when having experimental or computational budget restrictions. Additionally, the new concept – TL4CO – is introduced and tested as a part of the CO tool. TL stands for ‘transfer learning’ and means that optimisation algorithm at the core of the CO tool is initialised using the information from previous configurations. This allows the algorithm to converge faster to the same solution as BO4CO. The sections below describe the contributions in further detail.

4.4.1.1 BO4CO Recap

The problem of finding a global optimal configuration in the configuration space is known to be computationally complex. The heart of the DICE CO tool - BO4CO algorithm, presented in detail in the deliverable D5.1 [2], is able to find the best possible local optimum (and, hence, the application configuration settings corresponding to it) with the limited experimental budget.

BO4CO is a new auto-tuning algorithm which adopts Gaussian Processes (GPs) to iteratively estimate the mean and confidence interval of a response variable (such as application’s latency, throughput etc.) at the part of configuration space that has not been searched and sequentially drives the experimentation. BO4CO estimates the response surface (the surface of all possible solutions) of the system using historical performance data of a specific metric. According to the estimate it selects the next configuration which is located in the most promising region of the surface and runs an application with this configuration to obtain the input data for the next iteration. After a sufficiently large number of iterations BO4CO will eventually find an optimal configuration meeting the application developer’s requirements and Service-Level Agreements (SLAs) [2].

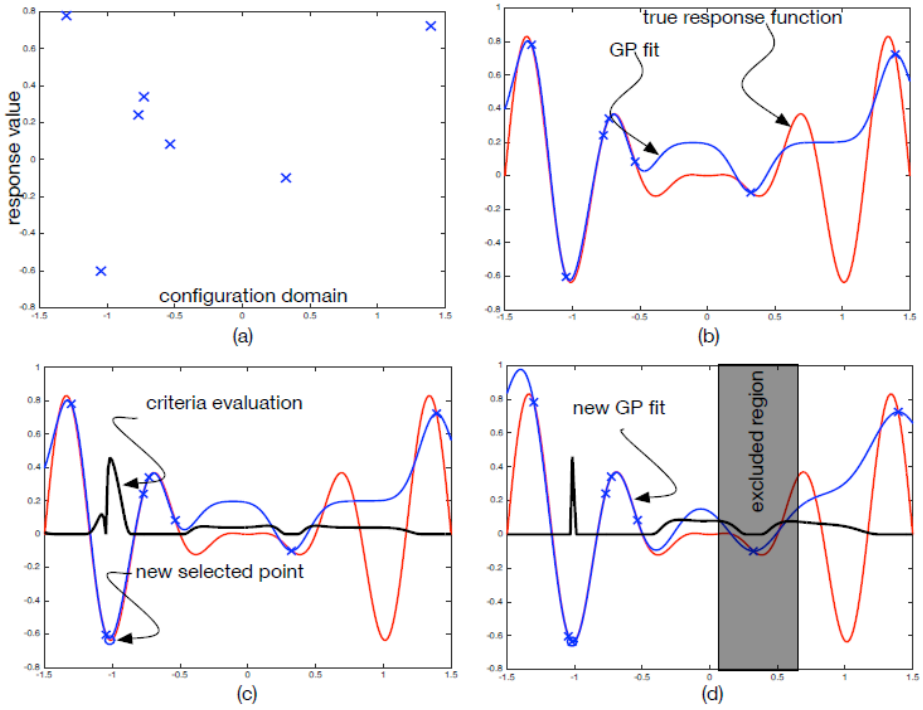


Figure 13: Illustration of BO4CO workflow (a) initial observations; (b) a GP model fit; (c) choosing the next point; (d) refitting a new GP model.

The BO4CO algorithm is illustrated in the Figure 13. At the beginning BO4CO uses Latin Hypercube Design (LHD) to produce an initial design, mainly because LHD ensures that the configuration samples are representative of the configuration space and can be taken one at a time, which is considered to be efficient in the high-dimensional spaces. After that, a GP model is fit to the observations obtained by running the application with the initial design generated in the previous step. Then, the model is used to calculate the selection criteria. The process is repeated until the configuration that can maximise the selection criterion is found.

To use this tool, the developer specifies which configuration parameters they are interested in and gives each parameter a range or possible values. Then the CO tool automatically selects the next configuration to be tested and, once this configuration has been measured, the performance data will be stored into the performance data repository in order to be used in the BO4CO algorithm. The sequential process continues until the maximum experimental budget has been reached and dumps the relatively optimal configuration for the application tested.

4.4.1.2 Transfer Learning for Configuration Optimisation (TL4CO)

Transfer Learning for Configuration Optimisation (TL4CO) is the evolution of the BO4CO algorithm used as a core of the DICE CO tool.

4.4.1.2.1 Introduction

Current solutions for developing more efficient configuration methods focus on improving the search mechanisms. We propose a different perspective, a DevOps-compliant solution that exploits the knowledge that has been gained in one system to accelerate the tuning of other similar systems. For example, significant correlations between the performance data of different versions of a system

were observed. We hypothesise that if we initialise the optimisation algorithm with the performance data from previous application versions instead of the performance data from the Latin Hypercube experiments, it would speed up the configuration tuning [11]. This is a reasonable assumption to make, because the previous configurations – even though they might not be optimal – would be 'closer' to the optimum than the Latin Hypercube Design.

We address the problem of finding optimal configurations under the following restrictions: (i) a configuration space composed of multiple parameters; (ii) a limited experimental budget is available; (iii) experimental data contains measurement noise. In this context, we demonstrate that transfer learning significantly contributes to the configuration optimisation process. While the literature on performance tuning (also known as auto-tuning [12]) is abundant with existing solutions for networked systems, databases, batch and stream processing, systems that address some of the above challenges (e.g., rule-based [13], design of experiments [14], model-based [12], [15], [16], [17], [18], search-based [17], [19], [20], [21], [22], [23], [24] and learning-based [25]), we only found one study that considers transfer learning [11].

Specifically, we propose TL4CO – an algorithm that leverages Multi-Task Gaussian Processes (MTGPs) [26] to continuously estimate the mean and confidence interval of the system performance at yet-to-be explored configurations. Using Bayesian Optimisation [27], the configuration optimisation process can account for all the available prior information, the acquired data on the current version, and apply a variety of kernel estimators [28] to locate regions where optimal configurations may reside. The key benefit of MTGPs over GPs that we benefit from in this work is that the prior information helps the model to converge to more accurate predictions much earlier. To validate the significance of transfer learning, we run the CO tool with TL4CO initialisation on the NoSQL benchmark (Apache Cassandra) and complex event processing application *Social Sensor* [29].

4.4.1.2.2 The need for transfer learning: a motivating example.

`WordCount` is a popular benchmark [30] that counts the number of words in the incoming stream. A Processing Element (PE) reads the input messages from a data source and pushes them into the system. The sentences are then split by the Splitter component and then counted by a Counter.

Figure 14 (a,b,c,d) shows the response surfaces for 4 different versions of `WordCount`, where splitters and counters, as configuration parameters, are varied. `WordCount` υ_1; υ_2 (also υ_3; υ_4) are identical in terms of the source code, but the deployment is different (we have deployed several other systems that compete for capacity in the same cluster). `WordCount` υ_1; υ_3 (also υ_2; υ_4) are deployed on a similar environment, but they differ in terms of the source code (we have artificially injected delays in the source code in the latter versions).

We have measured the correlation coefficients between the four versions. They are summarised in the Table 5 (upper triangle shows the coefficients while lower triangle shows the p-values). The correlations between the response functions are significant (p-values are less than 0.05). However, the correlation differs from version to version, and also some versions have inverse correlations (e.g. υ_1; υ_3). Also, more interestingly, different versions of the system have different optimum configurations: $x_{\upsilon 1}$ = (5; 1); $x_{\upsilon 2}$ = (6; 2); $x_{\upsilon 3}$ = (2; 13); $x_{\upsilon 4}$ = (2; 16). Current practices do not systematically use the knowledge gained from previous versions for performance tuning of the

current version under test despite such significant correlations [31]. The main reason is that the inherent search and optimisation techniques in current tuning methods cannot exploit the historical data to accelerate the sample generation or restrict the search space.

The response functions *f(·)* of the Figure 14 represent the dependency between the application's data-processing time (latency) and two configuration parameters – counters and splitters, for each of the four `WordCount` versions described above. These functions are strongly non-linear, non-convex and multi-modal. The performance difference between the best and worst settings is substantial, e.g., 65% in *v*4, providing a case for tuning. Moreover, the non-linear relations among the parameters imply that if one tries to minimise latency by acting just on one of the parameters, this may not lead to a global optimum [17].

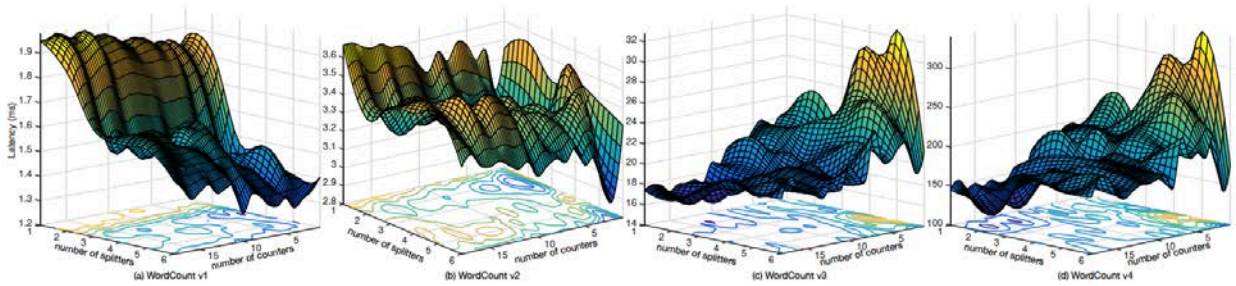


Figure 14: The response functions corresponding to 4 versions of WordCount that are different in terms of either *code* or *infrastructure*

Table 5: Performance measurements across different versions are significantly correlated, Pearson (Spearman) coefficients

	v1	v2	v3	v4
v1	1	0.41 (0.49)	-0.46 (-0.51)	-0.50 (-0.51)
v2	7.4e-06 (5.5e-08)	1	-0.20 (-0.27)	-0.18 (-0.24)
v3	6.9e-07 (1.3e-08)	0.04 (0.003)	1	0.94 (0.88)
v4	2.5e-08 (1.4e-08)	0.07 (0.01)	1.2e-52 (8.3e-36)	1

4.4.2 Tool's usage

The tool works from the command line. The experiment details to be carried out are encoded in the configuration file with the relative path `conf/exp_mt_config.yaml`.

The configuration file is comprised of several important parts: `runexp` specifies the experimental parameters, `services` comprises the details of the services which the CO tool uses, `application` contains the details of the application (e.g., Storm topology and the associated Java classes), and, most importantly, the details of the configuration parameters are specified in the `vars` field.

Listing 6 shows an example of parameters which specify the experimental budget (i.e., total number of iterations), the number of initial samples, the experimental time, polling interval and the interval time between each experimental iterations, all in milliseconds:

```
runexp:
  numIter: 100
  initialDesign: 10
  ...
  expTime: 300000
  metricPoll: 1000
  sleep_time: 10000
```

Listing 6: Example experiment settings in the TL4CO configuration file

Listing 7 demonstrates how to specify the configuration parameters that the CO tool will use in the optimisation process. The definition is complete with the name of the parameter, the names of the nodes in the DDSM/TOSCA model of the DIA, and various parameters defining the search space.

```
vars:
  - paramname: "mapreduce.map.memory.mb"
    node: ["namenode"]
    options: [128 512 1024 2048 4096]
    lowerbound: 0
    upperbound: 0
    integer: 0
    categorical: 1
```

Listing 7: Example of the specification of the target DIA's configuration parameter in the TL4CO's configuration file

To run the CO tool with TL4CO, the user then simply runs the start-up script, as shown in the Listing 8.

```
$ ./run_tl4co.sh
```

Listing 8: Running the TL4CO from the command line

4.4.3 Validation and results

4.4.3.1 BO4CO for Apache Hadoop

4.4.3.1.1 Hadoop Configuration Parameters

There are hundreds of configuration parameters in Hadoop, and they can be infrastructure-specific or application-specific. Hadoop provides default values for each configuration parameter which are put in the default XML parameters files and are inaccessible to users. Users can change settings by changing the non-default empty configuration files. There are three ways to change the configuration parameters: through hard-coding them in the application, through XML files (change them manually) and through command line interface [32].

Hadoop provides `Configuration` class and `Set()` method that allow users to access configuration parameters and change parameters' values within the application, as shown on the Listing 9.

```
Configuration conf = new Configuration();
conf.set("fs.default.name", hdfsUrl);
conf.set("dfs.datanode.address", "0.0.0.0:0");
conf.set("dfs.datanode.http.address", "0.0.0.0:0");
```

Listing 9: Setting configuration parameters in the application code

It is also possible to set parameter values directly in the XML files provided by Hadoop. Users can put property-value pairs into `core-site.xml, hdfs-site.xml, mapred-site.xml` and `yarn-site.xml`. Listing 10 shows an example for how to set basic parameters in hdfs-site.xml.

```
<!-- Put site-specific property overrides in this file. -->

<configuration>
        <property>
             <name>dfs.replication</name>
             <value>1</value>
        </property>
        <property>
             <name>dfs.namenode.name.dir</name>
             <value>file:/usr/local/hadoop/tmp/dfs/name</value>
        </property>
        <property>
             <name>dfs.datanode.data.dir</name>
             <value>file:/usr/local/hadoop/tmp/dfs/data</value>
        </property>

</configuration>
```

Listing 10: Setting configuration parameters via XML files

Users can also change default parameter values in the command line interface when submitting a Hadoop job. There are methods for shell scripts like `-conf` and `-D` which allow users to specify the configuration settings. Listing 11 shows an example of modifying configuration parameters when submitting a job via the command line interface.

```
hadoop@xidi-VirtualBox:~$ hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapre
duce-examples-2.7.2.jar wordcount -Dmapreduce.job.reduces=2 input output
```

Listing 11: Setting configuration parameters via command line

In this report we focus on the parameters that might affect Hadoop's performance. Hadoop configuration parameters potentially affecting its performance can be classified into three broad categories: related to the CPU utilisation, I/O and memory utilisation [32].

4.4.3.1.2 CPU-related Hadoop configuration parameters.

As shown in the Table 6, there are some parameters which might directly affect the CPU performance.

A good configuration parameters tuning strategy is necessary for better CPU performance for a Hadoop job.

Table 6: CPU-related Hadoop configuration parameters

Parameter name	Default value	Description
mapred.map.tasks	2	number of map tasks executed per job
mapred.reduce.tasks	1	number of reduce tasks executed per job
mapred.tasktracker.map.tasks.maximum	2	number of map tasks executed simultaneously by a task tracker per job
mapred.tasktracker.reduce.tasks.maximum	2	number of reduce tasks executed simultaneously by a task tracker per job
mapred.map.tasks.speculative.execution	true	if true, then multiple instances of some map tasks may be executed in parallel
mapred.reduce.tasks.speculative.execution	true	if true, then multiple instances of some reduce tasks may be executed in parallel

4.4.3.1.3 Memory-related Hadoop configuration parameters

As shown in Table 7, there are some parameters which might directly affect the memory performance.

Memory availability can influence how long a task would take to finish.

Table 7: Memory-related Hadoop configuration parameters

Parameter name	Default value	Description
mapreduce.map.memory.mb	1024	memory limit for map task
mapreduce.reduce.memory.mb	1024	memory limit for reduce task
mapreduce.task.io.sort.mb	100	memory limit when performing sorting task
mapreduce.task.io.sort.factor	10	number of streams to be merged when performing sorting task
mapred.child.java.opts	-Xmx200m	memory heap-size required

4.4.3.1.4 I/O-related Hadoop configuration parameters

As shown in the Table 8, there are some parameters which directly affect the performance of I/O. I/O parameters have a significant impact on the speed of the HDFS read and write operations.

Table 8: I/O-related Hadoop configuration parameters

Parameter name	Default value	Description
dfs.blocksize	134217728	block size in bytes for file
dfs.replication	3	number of blocks replicated
mapreduce.map.output.compress	false	decide whether map output is compressed or not
mapreduce.output.fileoutputformat.compress.type	RECORD	compression type of job output
mapreduce.map.output.compression.codec	org.apache.hadoop.io.compress.DefaultCodec	compression codec when map outputs are compressed

There are a lot of parameters that need to be set for Hadoop and tuned independently for each application, since different applications have different features. Each new release of Hadoop might deprecate some parameters as well as introduce new parameters which makes manual configuration tuning process even more labour-intensive.

4.4.3.2 Experiments and evaluation

4.4.3.2.1 Testbed setup

We use a cluster of three VMs to perform tests, including one NameNode and three DataNodes. The master node is both NameNode and DataNode. All the VMs used for testing are hosted on the Imperial College's Infrastructure-as-a-service private cloud based on Apache CloudStack. The host name of master node is '`cloud-vm-47-223`'. The operating system of the Master node of the cluster is Ubuntu v15.10 and the hardware statistics are shown in the

Table 9 and Table 10:

Table 9: Master Node Hardware

Item	Value
Number of CPU cores	2
CPU, MHz	1000
Memory, MB	3072
Disk size, GB	30

Table 10: Slave Node Hardware

Item	Value
Number of CPU cores	2
CPU, MHz	2000
Memory, MB	2048
Disk size, GB	100

The host names of slave nodes are '`cloud-vm-45-22`' and '`cloud-vm-45-25`' respectively.

4.4.3.2.2 Test plan

We use 1GB dataset to perform the tests on two different widely-used Hadoop MapReduce benchmark applications – `Wordcount` [30] and `Terasort` [33]. `WordCount` was introduced earlier in the Section 4.4.1.2.2, and `Terasort` measures the amount of time it takes to sort 1 TB of randomly distributed data. We test each application with two configurations: two and five parameter configurations (we vary these parameters while the rest remain unchanged throughout the experiments). For two-parameter experiments we chose `mapreduce.job.reduces` and `mapreduce.tasktracker.reduce.tasks.maximum` because they influence each other during the job execution [34]. For five-parameter experiments we selected parameters that might have significant impact on the Hadoop performance. The test plans for these two sets are shown in the Table 11 and Table 12 respectively:

Table 11: Test plan for 2-parameter configuration

Parameters	Default values	WordCount	TeraSort
mapreduce.job.reduces	1	1, 2, 3	1, 2, 3
mapreduce.tasktracker.reduce.tasks.maximum	2	2, 3	2, 3

Table 12: Test plan for 5-parameter configuration

Parameters	Default values	WordCount	TeraSort
mapreduce.job.reduces	1	1, 2, 3	1, 2, 3
mapreduce.tasktracker.map.tasks.maximum	2	2, 3	2, 3
mapreduce.tasktracker.reduce.tasks.maximum	2	2, 3	2, 3
mapreduce.task.io.sort.mb	100	100,110,120	100,110,120

mapreduce.task.io.sort.factor	10	10,20,40,80	10,20,40,80

According to [12], the optimal setting for `mapreduce.tasktracker.reduce.tasks.maximum` should be set between half of the number of CPU cores per node and two times of the number of CPU cores per node, i.e. in our case between 1 and 4. Here we choose values 2 and 3 to reduce experiments time. As listed in the Table 6, `mapreduce.job.reduces` sets the number of reduce tasks per job [35], which should be approximately equal to the number of reduce slots as found in [34]. Since parameter `mapreduce.tasktracker.reduce.tasks.maximum` controls the maximum number of map tasks run simultaneously by the task tracker (i.e. the number of reduce slots), we vary `mapreduce.job.reduces` in the range from 1 to 3.

Table 12 lists the five parameters we are interested in with their chosen ranges. The choice of the settings for the first and third parameters was explained above and the second parameter is similar to the third one, hence we also set it from 2 to 3. The parameter `mapreduce.task.io.sort.mb` controls the total amount of buffer memory to use when sorting files and it allocates 1 MB of memory for each merge stream by default in order to reduce seek times (time it takes for a disk drive to locate the area on the disk where the data to be read is stored). Normally, this parameter might reduce I/O times if increased, however it also requires more memory for each map task. If not much physical memory is available, the range for this parameter should be chosen carefully, otherwise it might cause job failure or stuck job. Parameter `mapreduce.task.io.sort.factor` indicates the number of streams to merge each time when sorting files and determines the number of open file handles. Because each merge stream occupies 1 MB of memory, this parameter should be set smaller than the fourth one to avoid `OutOfMemory` error.

For the first group, listed in the Table 11, there are six combinations of configuration in total and we'll be able to test them all (i.e. set the experimental budget `numIter` to 6) because it does not take much time to run the full set of tests. However, for the second group (listed in the Table 12), the total number of combinations is 144 and we limit the CO tool to 60 iterations (an experimental budget). Ten replications of each experiment were made to reduce experimental errors.

4.4.3.2.3 Test data generation

Because `Wordcount` and `TeraSort` work with text data arranged into different structures, we generate two datasets 1 GB each using `RandomTextWriter` [36], [37] and `TeraGen` [38], [39] Java scripts for `WordCount` and `TeraSort` respectively. The command for generating test data with `RandomTextWriter` is shown on the Listing 12.

With default settings `RandomTextWriter` generates 10 GB of data split into 10 files, each file containing 1 GB of data. We set `mapreduce.randomtextwriter.totalbytes` to 1 GB and `mapreduce.randomtextwriter.bytespermap` to 100MB so the script would generate the total 1 GB of data with 10 files.

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar
randomtextwriter -Dmapreduce.randomtextwriter.totalbytes=1000000000
-Dmapreduce.randomtextwriter.bytespermap=100000000 wordcountData
```

Listing 12: Generation of test data with `Randomtextwriter` via command line interface

The data generated is in plain text with random words, which is very suitable for `WordCount`. The format of the generated text is shown on the Listing 13. It is random in both length and letters.

```
Effie Scanic seditious Triphora hypoid breadwinner champer Mormyrus redescend michigan
supraoesophageal returnability steprelationship manilla archididascalian Haversian uncombable
allectory absvolt dipsomaniacal laurinoxylon pictorially glyphography propheticism interruptedness
astucious docimastical velaric reconciliable pictorially clanned outwealth metaphonical goladar
sud aurothiosulphuric bacterioblast eulogization toplike osteopaedion stiffish Haversia
```

Listing 13: Example sample of test data generated for `WordCount` application

`TeraGen` is a script generating test data for `TeraSort`. The command for the test data generation with `TeraGen` using the command line interface is shown on the Listing 14:

```
hadoop jar /usr/local/hadoop/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.2.jar
teragen -Dmapred.map.tasks=10 10000000 terasortData
```

Listing 14: Generation of test data with `TeraGen` via command line interface

`TeraGen` generates 100 bytes of data per row in the format shown below. In order to generate 1 GB of data we should add '10000000' to the command (the number of rows to be generated) and set `mapred.map.tasks` to 10 (which would result in 10 files).

```
<10 bytes key> <10 bytes rowid> <78 bytes filler>
```

The 'key' is some random characters with the ASCII value of each character in the range [32, 126]. The 'rowid' is an integer and the 'filler' contains 7 groups of characters with 10 characters in each group (8 characters in the last group) whose range is [A-Z].

4.4.3.2.4 Experimental results.

The two chosen Hadoop applications – `WordCount` and `TeraSort` – were evaluated with the Configuration Optimisation tool using the experimental data from the Sections 4.4.3.2.2 and 4.4.3.2.3. The performance metric chosen was latency (the data processing time of the application), measured in milliseconds. As shown in the Figure 15, for both `Wordcount` and `TeraSort` applications the CO tool improves their performance. Since the experiments were conducted on the rather small cluster with limited resources, and input data size is small compared to the real industry case, the tool did not produce dramatic improvement in application performance. Additionally, the set of parameters chosen for tuning by the CO tool was small due to the limited resources and computational time available, which might have also influenced the results (e.g. some of the chosen parameters were not that 'influential' on the application performance as we assumed or their default values are already well-tuned by Hadoop developers). However, these preliminary results look promising and we will continue the experiments with more powerful hardware resources and more parameters to tune.

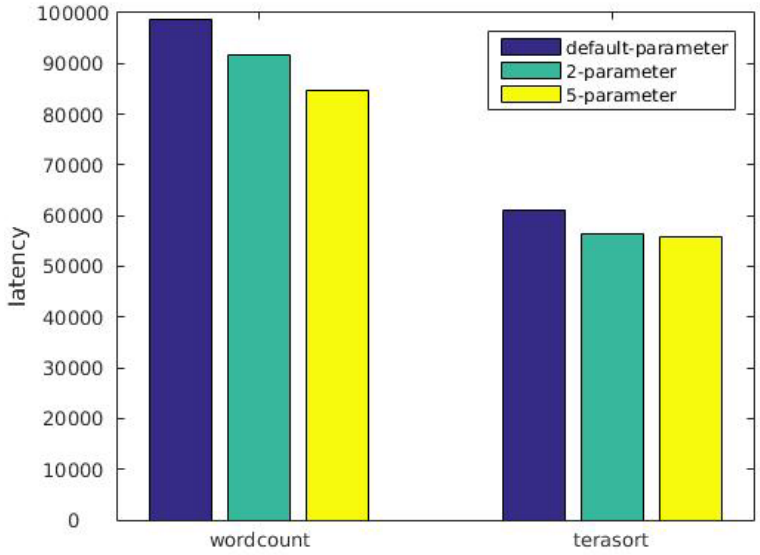


Figure 15: Experimental results for the `WordCount` and `TeraSort` applications

4.4.3.3 NoSQL benchmark optimisation (Apache Cassandra).

Since Apache Cassandra is designed to be write-efficient [40] (i.e. its configuration is optimised for write operations efficiency), we set a more challenging task of optimising for the read operations as well. Figure 16 shows the measurements of latency versus throughput for both read and write operations in the 20-paremeter configuration space. The configurations that are found by the CO tool (with TL4CO and BO4CO algorithms), the default settings and the one prescribed by experts [41] are annotated. The results show that the configuration that the CO tool with TL4CO initialisation finds only after 20 iterations results in a slightly lower latency but much higher throughput compared to the one suggested by the experts.

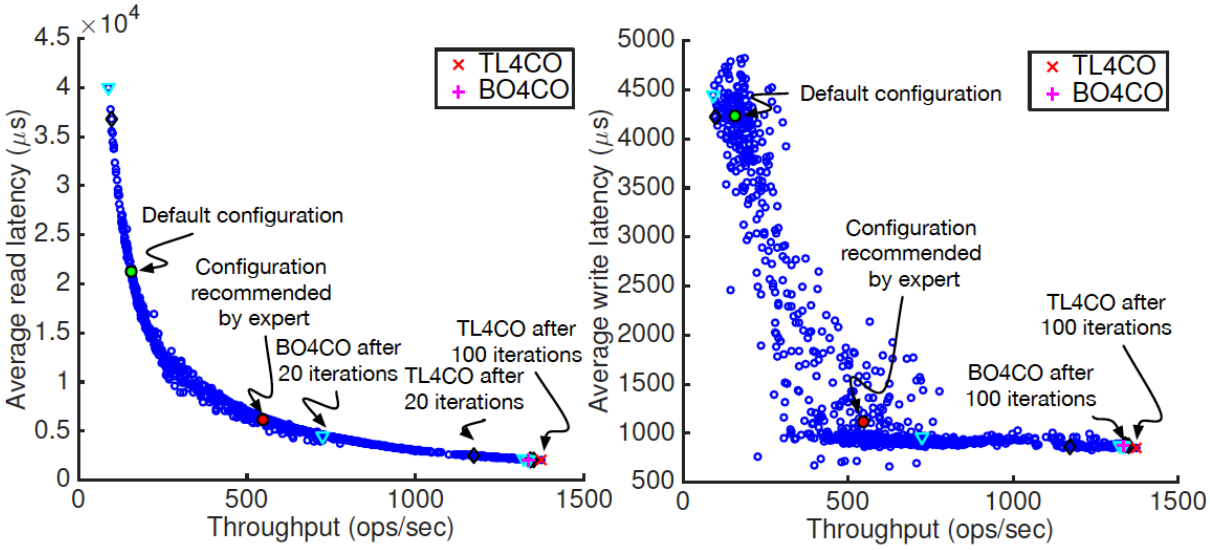


Figure 16: TL4CO in the NoSQL experiment comparing with BO4CO and expert. Each point corresponds to a performance (averaged over 10 min) of the system with a different configuration, resulted in 1024 points (lower right points are better)

4.4.3.4 Case study: ATC's Social Sensor.

We have also applied CO tool (both with BO4CO and TL4CO algorithms) to the ATC's *Social Sensor* [29]. It is a Storm and MongoDB-based DIA, mining data from online sources (e.g. social media, news websites) and processing it to extract trending topics. Social Sensor is a complex event-processing system consisting of nine distributed components and is latency-sensitive due to both combining data from multiple sources and looking for events in real time.

We have selected 20 configuration parameters with two levels for each, thus creating a configuration space of the size $|\mathbb{X}| = 2^{20}$. The exhaustive search (to run all possible combinations of these parameters) would take 10^6 * 10 min = 6944 days = 19 years. We compare the throughput and latency obtained by running *Social Sensor* with configuration parameters chosen by the CO tool to measured using the default configuration. The results in the Figure 17 show a significant gain in performance with the configuration suggested by the CO tool. The improvement in throughput of more than twice compared to the default configuration was achieved after only 100 iterations ($100/2^{20}$ = 0,0095% of the possible configurations) which took 100 * 10 min = 16h to run.

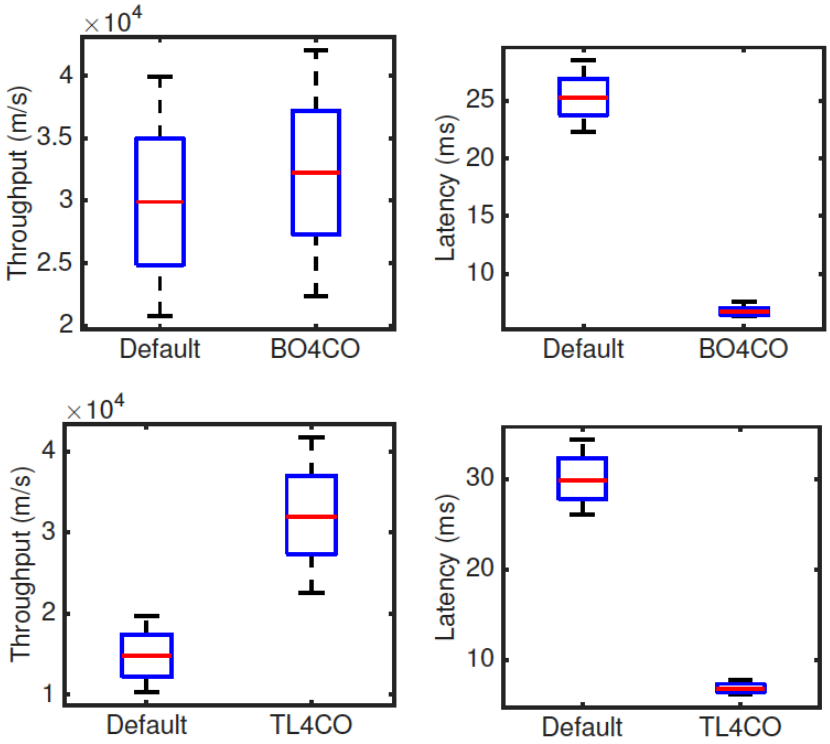


Figure 17: Application of CO tool (with BO4CO and TL4CO algorithms) to the *Social Sensor* [29]

4.4.4 Obtaining Configuration Optimisation

Both the BO4CO and the TL4CO are available at GitHub:

- https://github.com/dice-project/DICE-Configuration-TL4CO - TL4CO
- https://github.com/dice-project/DICE-Configuration-BO4CO - BO4CO

5 Conclusion

In a highly competitive setting such as software development, any tool that can speed up the process of testing and deploying components of a product are bound to be valuable. Big Data technologies have been available in approachable open source form for several years now. However, their adoption by many both novice and experienced developers is still a slow process.

The DICE delivery and configuration tools in their initial version were a prototype with a working, but small base of supported technologies. In one year, we have added a number of technologies to the library, now enabling to configure and deploy a much richer spectrum of building blocks in topologies of an arbitrary complexity. We also improved the usability, so that the deployment is even more accessible, easy to use. We find it important to empower users to use the tools in the best way possible for their own workflow and setting, and we believe that we have achieved that by offering a number of ways of invoking these tools. Web GUI, RESTful interface and command line utility are all valid ways to do the work. We demonstrated the usability of these interfaces by providing a plug-in for Eclipse [5], and by easily configuring another popular editor Sublime Text to perform one-click deployments right from the editor.

Speed without sacrificing quality was also a motivation in improving the DICE Configuration Optimisation. By introducing the TL4CO algorithm, every run of the configuration optimisation produces data, which will improve results of the next run. This saves time for the users, reduces cost of computation needed to obtain the recommendation, while at the same improving long-term performance of the DIA under test.

Considering that the DICE tools themselves are components, which need to be deployed themselves, we built Chef cookbooks and created TOSCA blueprints for them as well. That way, even the initial mandatory hurdle of having to configure and install the delivery and configuration tools is much smaller.

We believe that the intermediate version of the delivery tools already represents a compelling package, which can be used daily or even more regularly to speed up time to market of new DIAs. Still, we will look into further improving the offering. This will go in two directions. First, we will add any missing or newly compelling technologies to the DICE TOSCA technology library. Second, we will research how to further increase the speed and responsiveness of the tools. The process of extending the DICE TOSCA technology library involves open technologies, so in principle any moderately skilled users can add their own building blocks by following our examples. For the less skilled users, the script runner node types can provide the needed minimal library extension support.

In the final period of the project we plan to extend the functionality of the DICE Delivery Tool in a way to support the Privacy-by-Design paradigm. This paradigm is presented in the peer deliverable [6] at the modelling level. The challenge will be to use the tools available and capabilities granted by the Big Data building blocks to either enforce the required policies, or at least log any violations of the policies.

For the Configuration Optimization, we will focus on further integrating the tool with other DICE tools such as the DICE IDE, DICE Monitoring Tool and Continuous Integration.

5.1 DICE Requirement compliance

In the Section 2 we provided a summary of the requirements. Table 13 indicates the level that the DICE Delivery Tools comply in their initial release. The *Level of fulfilment* column has the following values:

- ✗ - not supported yet
- ✓ - initial support
- ✓✓ - medium level support
- ✓✓✓ - fully supported

Table 13: 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	SHOULD	✓✓✓
R5.4	TOSCA format for blueprints	MUST	✓✓✓
R5.4.1	Big Data technology support	MUST	✓✓
R5.4.2	Translation tools autonomy	MUST	✓✓✓
R5.4.5	Deployment tools transparency	SHOULD	✓✓✓
R5.4.6	Deployment plans extendability	SHOULD	✓✓
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	✓✓✓
R5.7.1	Data loading hook	SHOULD	✓✓
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	Configuration Optimisation	MUST	✓✓✓
R5.27.1	Brute-force approach for CONFIGURATION_OPTIMIZATION deployment	SHOULD	✓✓✓
R5.27.6	CONFIGURATION_OPTIMIZATION experiment runs	MUST	✓✓✓
R5.27.7	Configuration optimisation of the system under test over different versions	SHOULD	✓✓✓
R5.27.8	Configuration Optimisation's input and output	MUST	✓✓✓
R5.43	Practices and patterns for security and privacy	MUST	✗

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

- R5.4.1: we plan to provide support for MongoDB by M26.

- R5.4.6: a part of the TOSCA library expandability effort will be documenting the process needed for adding support for new technologies.
- R5.4.8: in M25, we will enable automatic connection of Cassandra and MongoDB nodes to be monitored by the DICE Monitoring Tool.
- R5.7.1: by M30 we plan to enable loading of data into Cassandra from offline files, which will be referenced to from the new TOSCA blueprint node types.
- R5.19 and R5.20 will be researched and reported by M30.
- R5.43: security and privacy by design will be a topic of research in the last year of the project. We expect by M30 to provide proof of concept solutions at least for Cassandra.

References

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

[2] DICE consortium, *DICE deliverable 5.1: DICE delivery tools – Initial version*, January 2016

[3] DICE consortium, *DICE deliverable 1.2 Requirement Specification*, July 2015

[4] DICE consortium, *DICE deliverable 1.4 Architecture definition and integration plan - Final version*, January 2017

[5] DICE consortium, *DICE deliverable 1.5 DICE framework - Initial version*, January 2017

[6] DICE consortium, *DICE deliverable 2.2 Design and quality abstractions - Final version*, January 2017

[7] DICE consortium, *DICE deliverable 4.2 Monitoring and Data warehousing tools - Final version*, January 2017

[8] DICE consortium, *DICE deliverable 5.4 DICE testing tools – Initial version*, January 2017

[9] DICE consortium, *DICE deliverable 5.6 Cloud testbed setup*, July 2016

[10] Wynne M., Hellesøy A., *The Cucumber Book - Behaviour-Driven Development for Testers and Developers*. The Pragmatic Bookshelf. January 2012.

[11] H. Chen, W. Zhang, and G. Jiang. Experience transfer for the configuration tuning in large-scale computing systems. *Knowledge and Data Engineering, IEEE Transactions on*, 23(3):388–401, 2011.

[12] N. Yigitbasi et al. Towards machine learning-based auto-tuning of mapreduce. In *Proc. MASCOTS*, 2013.

[13] E. Kwan et al. Automatic database configuration for DB2 universal database: Compressing years of performance expertise into seconds of execution. In *Proc. BTW*, volume 20, 2003.

[14] N. Siegmund et al. Performance-influence models for highly configurable systems. In *Proc. FSE*, 2015.

[15] D. A. Menascé et al. Preserving QoS of e-commerce sites through self-tuning: a performance model approach. In *Proc. of EC. ACM*, 2001.

[16] T. Johnston et al. Performance tuning of mapreduce jobs using surrogate-based modeling. *Proc. ICCS*, 2015.

[17] P. Jamshidi and G. Casale. An uncertainty-aware approach to optimal configuration of stream processing systems. In *Proc. MASCOTS*, 2016.

[18] D. G. Sullivan, M. I. Seltzer, and A. Pfeffer. *Using probabilistic reasoning to automate software tuning*, volume 32. ACM, 2004.

[19] B. Xi et al. A smart hill-climbing algorithm for application server configuration. In *Proc. WWW*, 2004.

[20] T. Osogami and S. Kato. Optimizing system configurations quickly by guessing at the performance. In *Proc. SIGMETRICS. ACM*, 2007.

[21] R. Thonangi et al. Finding good configurations in high-dimensional spaces: Doing more with less. In *Proc. MASCOTS*. IEEE, 2008.

[22] N. Hansen and A. Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[23] B. Behzad et al. Taming parallel I/O complexity with auto-tuning. In *Proc. ACM SC*, 2013.

[24] T. Ye and S. Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. *Proc. SIGMETRICS*, 2003.

[25] X. Bu, J. Rao, and C.-Z. Xu. A reinforcement learning approach to online Web systems auto-configuration. In *Proc. ICDCS*, 2009.

[26] E. V. Bonilla, K. M. Chai, and C. Williams. Multi-task Gaussian process prediction. In *Proc. of NIPS*, pages 153–160, 2007.

[27] B. Shahriari et al. Taking the human out of the loop: a review of Bayesian optimization. Technical report, 2015.

[28] D. J. Lizotte et al. An experimental methodology for response surface optimization methods. Global Optimization, 53:699–736, 2012.
[29] Social Sensor socialsensor.eu
[30] A. Ghazal et al. Bigbench: towards an industry standard benchmark for big data analytics. In *Proc. of SIGMOD*, pages 1197–1208. ACM, 2013.
[31] L. Bass, I. Weber, and L. Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.
[32] Bhavin J Mathiya and Vinodkumar L Desai. Apache Hadoop yarn parameter configuration challenges and optimization. In *Soft-Computing and Networks Security (ICSNS), 2015 International Conference on*, pages 1–6. IEEE, 2015. pages 1, 14
[33] Terasort https://www.mapr.com/resources/terasort-benchmark-comparison-yarn
[34] Kewen Wang, Xuelian Lin, and Wenzhong Tang. Predator an experience guided configuration optimizer for hadoop mapreduce. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 419–426. IEEE, 2012. pages 1, 10, 34, 47
[35] Apache Hadoop. Default settings for mapred.xml. https://hadoop.apache.org/docs/r2.7.2/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml
[36] RandomTextWriter Java script https://github.com/facebookarchive/hadoop-20/blob/master/src/examples/org/apache/hadoop/examples/RandomTextWriter.java
[37] RandomTextWriter documentation https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/examples/RandomTextWriter.html
[38] TeraGen Java script https://github.com/apache/hadoop/blob/trunk/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples/terasort/TeraGen.java
[39] TeraGen documentation https://hadoop.apache.org/docs/r1.0.4/api/org/apache/hadoop/examples/terasort/TeraGen.html
[40] B. F. Cooper et al. Benchmarking cloud serving systems with ycsb. In *Proc. of SOCC. ACM*, 2010.
[41] N. Nelubin and B. Engber. Ultra-high performance NoSQL benchmarking: Analyzing durability and performance tradeoffs. Technical report, 2013.

