Developing Data-Intensive Cloud Applications with Iterative Quality Enhancements

Iterative quality enhancement tools – Initial version

Deliverable 4.5

Deliverable:	D4.5
Title:	Iterative quality enhancement tools – Initial version
Editor(s):	Giuliano Casale, Chen Li (IMP)
Contributor(s):	Giuliano Casale (IMP), Chen Li (IMP), Weikun Wang (IMP), Jose-Ignacio Requeno (ZAR)
Reviewers:	Danilo Ardagna (PMI), Craig Sheridan (FLEXI)
Type (R/DEM/DEC):	DEM
Version:	1.0
Date:	31-July-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 documents the initial work on tools for iterative quality enhancement, developed as part of task T4.3. This component feeds results back into the design models to provide guidance to the developer on the quality offered by the application at runtime. In the initial version, the tool is able to estimate and fit application parameter related to memory and execution times and annotate UML models. Moreover, initial work has been carried out towards defining architecture and algorithms for anti-pattern detection and architecture refactoring. Initial validation has been carried across a variety of technologies, including Cassandra, Hadoop/MapReduce, and an in-memory DB.

Glossary

ADL	Architecture Description Language
ADT	Anomaly Detection Tool
APR	Anti-Patterns & Refactoring
PAML	Performance Anti-pattern Modeling Language
CTMC	Continuous-time Markov chain
DIAs	Data-intensive applications
DICE	Data-Intensive Cloud Applications with iterative quality enhancements
DMon	DICE Monitoring platform
FG	Filling-the-Gap
MARTE	Modeling and Analysis of Real-Time and Embedded Systems
MCR	MATLAB Compiler Runtime
MODAClouds	MOdel-Driven Approach for design and execution of applications on multiple Clouds
M2M	Model-to-Model Transformation
QN	Queueing Network
SLOs	Service level objectives
TraCT	Trace checking tool
UML	Unified Modelling Language
VM	Virtual Machine

Table of contents

List of Figures

List of Tables

1. Introduction and Context

This deliverable presents the initial release of the DICE Enhancement tools (ENHANCEMENT_TOOLS), which are being developed in task T4.3 within the WP4 work package. The main goal of the DICE Enhancement tools is to provide feedback to DICE developers on the application behaviour at runtime, leveraging the monitoring data from the DICE Monitoring Platform (MONITORING_TOOLS), in order to help them iteratively enhance the application design.

Enhancement tools introduce a new methodology and a prototype to close the gap between measurements and UML diagrams. According to our knowledge, no mature methodology appears available in the research literature in the context of data-intensive applications (DIAs) to address the difficult problem of going from measurements back to the software models, annotating UML to help reasoning about the application design. ENHANCEMENT_TOOLS aims at filling this gap.

The rest of this section presents the objectives of task T4.3, and discusses the relation to other WP4 tasks of DICE tool-chain. We also review objectives of this deliverable.

1.1. Objectives of T4.3

Task 4.3 in WP4 focuses on the development of ENHANCEMENT_TOOLS which aim at filling the gap between the runtime (monitoring data) and design time (models and tools). The general working principle of this toolset is as follows. Upon demand by the user, ENHANCEMENT_TOOLS queries the DICE Monitoring Platform (MONITORING_TOOLS) to obtain runtime monitoring data, normally gathered during application tests, but possibly also during production.

The tools then correlate this monitoring data to the DICE UML models developed within WP2, with the aim of bridging the semantic gap between UML abstractions and concrete system implementation. For example, ENHANCEMENT_TOOLS estimates parameters for the SIMULATION_TOOLS and OPTIMIZATION_TOOLS, developed as part of WP3, by inferring the execution times of application requests placed at the different resources of the DIA. Based on the acquired data, ENHANCEMENT_TOOLS allows the developer to conduct within the DICE IDE more precise simulations and optimizations, that can rely on experimental data, rather than guesses of unknown parameters. ENHANCEMENT_TOOLS will also support the developer in carrying out refactoring scenarios, with the aim of iteratively improving application quality in a DevOps fashion,

1.2. Relation to other WP4 tasks

The other main tools developed in WP4 are the MONITORING_TOOLS (DMon), the DICE Anomaly Detection Tool (DICE ADT) and the DICE Trace Checking Tool (DICE TraCT). While the monitoring platform is primarily concerned with acquiring monitoring data from the runtime environment, the other tools are concerned with verifying the quality of the executions, in terms of characteristics such as performance and correctness. These aims are clearly different from those of the ENHANCEMENT_TOOLS, which is foremost focused on inferring the parameters of UML models from monitoring data and guide the refactoring process.

1.3. Objectives of the Document

This document presents the initial release of ENHANCEMENT_TOOLS. The present report is the first of two deliverables, the second being set for release at M30. The current deliverable focuses in particular on parameter estimation and on setting the technical approach for the refactoring methodology. The next deliverable (at M30) is scheduled to focus on the refactoring methods and on expanding the parameter estimation capabilities of the tool, for example to capture technology-specific characteristics.

1.4. Structure of the document

The structure of this deliverable is as follows:

- Chapter 2 recaps on the requirements that T4.3 aims to cover.

- Chapter 3 presents an updated architecture and design details of the Enhancement tool, which extends the initial design provided in deliverable D1.3.
- Chapter 4 presents the design principle of DICE-FG which is the key part of this initial release Enhancement tool.
- Chapter 5 discusses the new scientific algorithms we have developed specifically for the DICE-FG module.
- Chapter 6 discusses the envisioned technical approach for application refactoring and anti-pattern detection, reporting on initial results on a cloud-based software system, as part of the Anti-Patterns & Refactoring (APR) submodule.
- Chapter 7 summarises achievements, overall progress, and outlines the future work.

Appendix A provides more detail on experimental validation of the tool against a number of cases studies.

2. Requirements

This section reviews the requirements of the ENHANCEMENT_TOOLS. Then we explain how requirements have been fulfilled in the current prototype.

2.1. Requirements

Deliverable D1.2 Requirements specifications [1] describes the requirements analysis for the DICE project. This section summarizes these requirements. We here list the Must have requirement of the ENHANCEMENT_TOOLS. Should have and could have requirements are available in [1] and in successive versions of D1.2 released on the DICE website[1].

Table 1: Resource consumption breakdown Requirement

ID	R4.11
Title	Resource consumption breakdown
Priority	Must have
Description	The DEVELOPER MUST be able to obtain via the ENHANCEMENT_TOOLS the resource consumption breakdown into its atomic components.

Table 2: Bottleneck Identification Requirement

ID	R4.12
Title	Bottleneck Identification
Priority	Must have
Description	The ENHANCEMENT_TOOLS MUST indicate which classes of requests represent bottlenecks for the application in a given deployment.

Table 3: Semi-automated anti-pattern detection Requirement.

ID	R4.13
Title	Semi-automated anti-pattern detection
Priority	Must have
Description	The ENHANCEMENT_TOOLS MUST feature a semi-automated analysis to detect and notify the presence of anti-patterns in the application design.

Table 4: Enhancement tools data acquisition.

ID	R4.17
Title	Enhancement tools data acquisition
Priority	Must have
Description	The ENHANCEMENT_TOOLS must perform its operations by retrieving the relevant monitoring data from the MONITORING_TOOLS.

Table 5: Enhancement tools model access Requirement.

ID	R4.18
Title	Enhancement tools model access
Priority	Must have
Description	The ENHANCEMENT_TOOLS MUST be able to access the DICE profile model associated to the considered version of the APPLICATION.

[1] www.dice-h2020.eu/deliverables/

Table 6: Parameterization of simulation and optimization models Requirement.

ID	R4.19
Title	Parameterization of simulation and optimization models.
Priority	Must have
Description	The ENHANCEMENT_TOOLS MUST extract or infer the input parameters needed by the SIMULATION_TOOLS and OPTIMIZATION_TOOLS to perform the quality analyses.

Table 7: Propagation of changes/automatic annotation of UML models Requirement.

ID	R4.27
Title	Propagation of changes/automatic annotation of UML models
Priority	Must have
Description	ENHANCEMENT_TOOLS MUST be capable of automatically updating UML models with analysis results (new values)

3. Architecture and Design of the Enhancement Tool

The DICE Enhancement tool is designed for iteratively enhancing the DIA quality. Enhancement tool aims at providing a performance and reliability analysis of big data applications, updating UML models with analysis results, and proposing a refactoring of the design, if performance anti-patterns are detected. Figure 1 shows the workflow we have defined in task T3.4 for the Enhancement tool, which covers all of its intended functionalities, which are discussed in details below.

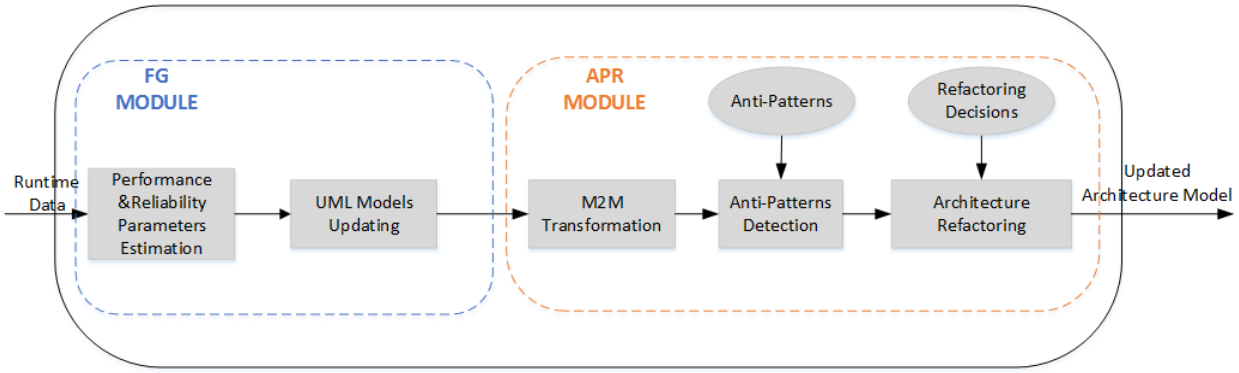


Figure 1. Workflow of Enhancement Tool

3.1. Core Components

The core components of the DICE Enhancement tools are two modules:

- DICE Filling-the-Gap (DICE-FG) module (see Figure 2), a tool focusing on statistical estimation of UML parameters used in simulation and optimization tool. This tool has been initially developed relying on a baseline, called FG, provided by the MODAClouds FP7 project as a way to close the gap between Development and Operations. Within DICE, the tool has undergone a major revision and is being integrated and adapted to operate on DIA datasets. Architectural changes have been introduced in DICE-FG, compared to the original FG.
- APR (Anti-Patterns & Refactoring) module, a tool for anti-patterns detection and refactoring. The tool aims at suggesting improvements to the designer of the DIAs, based on observed and predicted performance and reliability metrics. The goal is to optimize a reference metric, such as maximize latency or minimize mean time to failure (MTTF). Differently from DICE-FG, APR has no baseline software to start from, since the only available tools in this space are not for UML. Hence it will be an original contribution of DICE, to our knowledge novel in the UML space.

Together, DICE-FG and APR cover the entire workflow of the Enhancement Tool. Since the output of DICE-FG is required by APR, in the initial work carried out in WP4 up to M18, we have focused primarily on DICE-FG, whereas APR has been designed and the technical methodology validated, with the goal of delivering an implementation of a prototype at M24, in conjunction with the first release of the DICE framework, and a final version at M30, with the second release of the DICE framework.

3.1.1. Filling-the-Gap (DICE-FG) Module

DICE-FG is designed to achieve the following objectives:

- Provide statistical estimation algorithms to infer resource consumption of an application.
- Provide fitting algorithms to match monitoring data to parametric statistics distributions.
- Use the above algorithms to parameterize UML models annotated with the DICE profile.
- Acquire data via JSON and the DICE Monitoring platform (DMon).

The main logical components of the DICE-FG tool are the Analyzer and the Actuator. Below we describe each component:

- **DICE-FG Analyzer**: The DICE-FG Analyzer executes the statistical methods necessary to obtain the estimates of the performance models parameters, relying on the monitoring information available on the input files.
- **DICE-FG Actuator**: The DICE-FG Actuator updates the parameters in the UML models, e.g., resource demands, think times, which are obtained from the DICE-FG Analyzer.

DICE-FG relies on the MATLAB Compiler Runtime (MCR R2016a) for execution, this is a royalty-free environment that does not require a MATLAB license. MATLAB source code is also provided in the release. The proposed architecture is leaner than the original FG tool developed in MODAClouds, henceforth denoted as MODAClouds-FG, a decision we have taken considering that DICE-FG may have to cope with a much larger number of parameters and models than MODAClouds-FG, which was more intended for use at run-time in applications such as load-balancing. Here is a detailed comparison of the two tools, highlighting all the major differences.

Table 8: Comparison of MODAClouds-FG and DICE-FG

	MODAClouds-FG	DICE-FG
Algorithms	Estimation of execution times, think times, number of jobs.	Estimation of execution times, think times, number of jobs, memory usage, for single resources and collections of resources. Fitting of data to statistical distributions.
Data input	Fuseki local database based on MODAClouds ontology	JSON-based input, compatible with D-MON monitoring platform. An XML-based input configuration language to constraint, verify and optimize the performance of the analysis workflow.
Model output	Annotated layered queueing network model for use with LINE queueing network solver.	Annotated UML model which can be mapped to the DICE simulation and optimization tools.
Reporting	A PDF report is generated to summarize the outcomes.	Model annotations which can be inspected via the DICE Eclipse IDE GUI.

3.1.2. Anti-Patterns and Refactoring (APR) Module

An Anti-Pattern (AP) is identified as a bad design practice, e.g., *Blob*, *Empty Semi Trucks*, which might cross several levels of an application development cycle, e.g., architecture definition, development. We

consider, for each AP, a problem statement, which identifies the AP but observable problems, and the corresponding solution actions, which need to be applied to the software system in order to remove the SP. In our case, the solutions should support the feedback generation which leads to architecture refactoring (e.g., a set of modifications in the parameters of models).

The DICE Anti-Patterns and Refactoring (APR) module is designed to achieve the following objectives:

- Transforming UML diagrams annotated with DICE profiles to performance model (e.g., Petri Nets and/or Queueing Networks) for performance analysis.
- Specifying the selected popular AP of DIAs in a formal way (e.g., a logic formula which is suitable for model checking, executable codes).
- Detecting the potential AP from the performance model.
- Generating refactoring decisions to update the architecture model (manually or automatically) to fix the design flaws according to the AP solution.

The components of the APR module are Model-to-Model (M2M) Transformation, Anti-patterns Detection and Architecture Refactoring as detailed below.

- **Model-to-Model (M2M) Transformation**: The component is based on some of the transformations developed in T3.1 and APR-specific transformations developed in T4.3. It provides the transformation of annotated UML model with DICE Profile into quality analysis model. The target performance models can be Petri Nets or Queueing Networks.
- **Anti-patterns Detection**: The Anti-patterns detection component relies on the analysis results of the M2M Transformation component. The selected anti-patterns are formally specified for identifying if there are any anti-patterns issues in the model.
- **Architecture Refactoring**: According to the solution of discovered anti-patterns, refactoring decisions will be proposed, e.g., component replacement or component reassignment, to solve them. The Architecture model will be shared back to the DICE IDE for presentation, to the user in order to decide if the proposed modification should be applied or not.

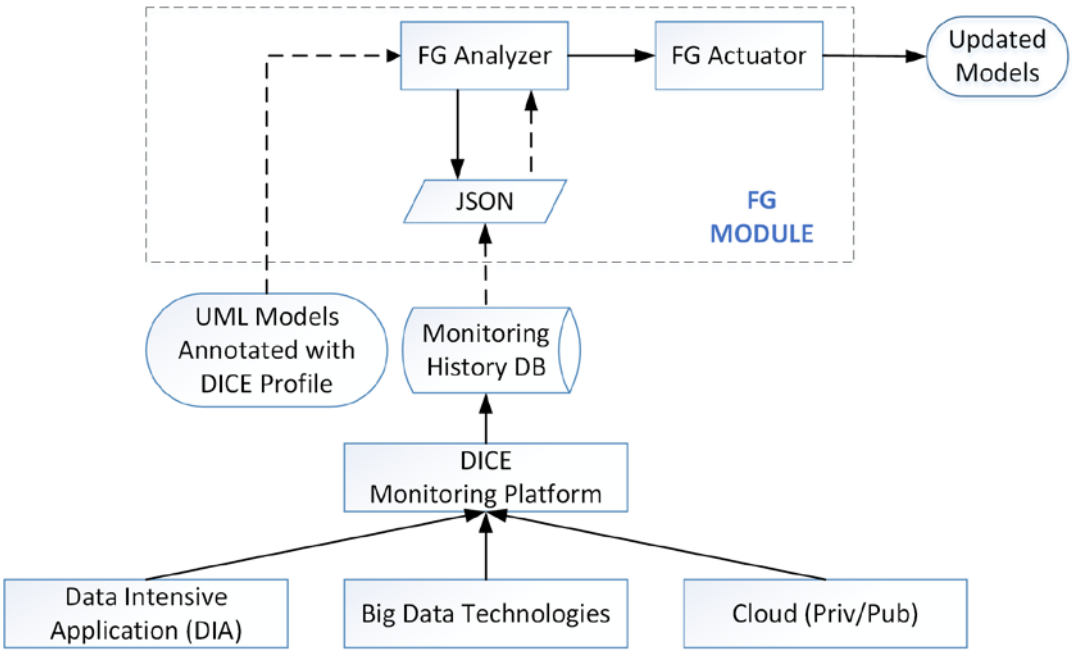


Figure 2. Interactions among the core components of DICE-FG Module.

3.2. Components Interaction

Based on the above description of the core component - DICE-FG tool, Figure 2 describes their interactions within the DICE-FG tool. For readability, the DICE Monitoring Platform is also included to show what data are collected. The dashed box highlights the sub-modules composing the DICE-FG tool, which offer more functionalities compared to the MODAClouds-FG baseline, as will be further elaborated in section 4.

However, differently from the baseline, data is now acquired into the DICE-FG through a JSON dump of the DICE Monitoring Platform, which in turn is capable of obtaining metrics for the DIA and the Big data platform running the DIA (e.g., log-files of Hadoop/MapReduce, Spark, etc), as well as from the underpinning virtual machines (VMs). These provide a richer set of input metrics compared to the MODAClouds baseline, which was concerned only with the VM metrics.

4. DICE-FG tool

In this section, we present the design principle for DICE-FG – the key component of this first release of the Enhancement tool.

4.1. Goal

As a core component of the Enhancement tool, the DICE-FG tool plays two roles:

- Updating parameters of design time model (UML models annotated with DICE Profile)
- Providing in the UML resource usage breakdown information for the data-intensive application.

Together these features provide to the DICE designer the possibility to:

- Benefit from a semi-automated parameterization of simulation and optimization models. This supports the state goal of DICE of reducing the learning curve of the DICE platform for users with limited skills in performance and reliability engineering.
- Inspect in Eclipse the automated annotations placed by DICE-FG to understand the resource usage placed by a workload across software and infrastructure resources.

The above features are graphically illustrated in Figure 3, which illustrates the model parameterization process undertaken **without** DICE-FG. Figure 4 instead shows the result **with** the DICE-FG automatic parameterization, i.e., the determination of parameters such as resource processing rates that are indispensable to predict performance and reliability through the cycle of iterative refinement.

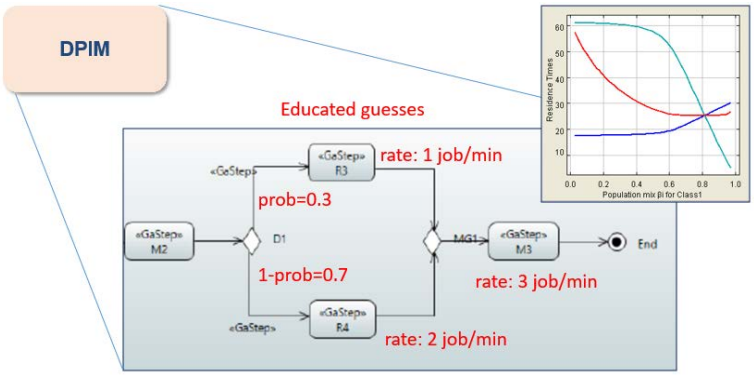


Figure 3. Manual guess of parameters by the DICE designer at DPIM model level for performance prediction

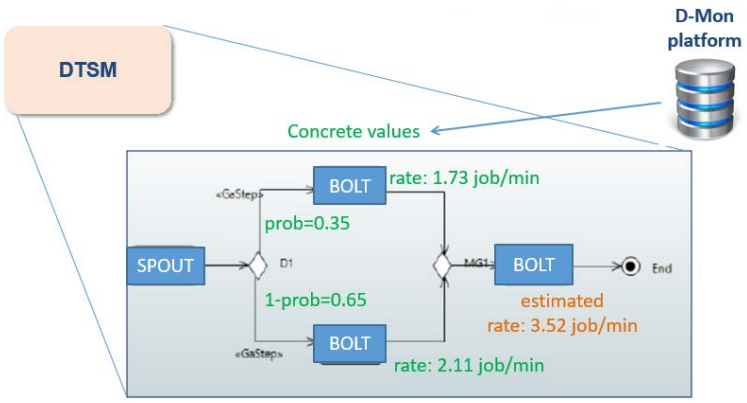


Figure 4. DICE-FG avoids at DTSM the parameter guessing through inference and fitting of monitoring data

Note that Figure 4 effectively provides a **resource usage breakdown**. This is because the rate of processing is easily related to the time a request spends at each resource as follows, i.e., let *R* be the rate of processing of a request at a given resource, then the mean time spent in execution at the resource is 1/*R*, after discount of contention overheads. Such mean time parameter is explicitly captured by the

hostDemand parameter in the DICE UML model and one of the key parameters estimated by DICE-FG. Other parameter of interests include the parallelism level at a resource, and the inter-arrival times of jobs, among others.

4.2. Updating UML parameters with DICE-FG

The DICE-FG tool operation process involves three stages: configuration, analysis (either estimation or distribution fitting), and model update. To be specific, firstly, the DICE-FG tool is interfaced to D-MON and the model repository. Secondly, DICE-FG Analyzer performs inference analysis on the datasets provided in input via D-MON. Then, DICE-FG updates the parameters of DICE UML models according to the results of estimation and fitting. Integration activities planned at later stage of the project will ensure that such annotated UML is returned to the IDE.

The following table describes the output parameters supported by DICE-FG with corresponding examples. We point to deliverable *D2.1 - Design and quality abstractions - Initial version* for a technical overview of DICE UML models and their parameters.

Table 9: Output parameters currently supported by DICE-FG tool

Output	Parameter Name	Description	Example
UML Models – MARTE Profile	hostDemand	Execution time. This is the real CPU demand, after contention overheads are discarded.	Mean time a mapper takes to process a task spawned by a Hadoop/MR job.
	extDelay	Inter-issue times of successive jobs	Average time between submission of jobs to a resource
	population	Average number of jobs running in the system	Number of jobs observed at the resource during testing
UML Models – DICE Profile	hadoopExtDelay	Inter-issue times of successive Hadoop jobs	Average time between submission of jobs to a Hadoop/MR or Spark cluster.
	hadoopPopulation	Average number of Hadoop jobs running in the system	Number of Hadoop/MR or Spark jobs observed in the system during testing.
	respT	Response time (elapsed time since a user submits a job to the cluster and return of the result)	Execution time of the map and reduce phases plus the time spent in the queues and communication delays

We have also been working towards annotating *average memory requirements* of individual jobs. At the moment this can be estimated by DICE-FG from the data, but there is a lack of a suitable annotation in the DICE profile, since memory annotations are inherited from UML MARTE for a host, but not for individual job. This limitation is now identified and will be addressed in the next released of the DICE profile.

4.3. Running DICE-FG

In this section, we overview the input data format and the configuration file provided to the DICE-FG tool. These are the only inputs required to run the tool, which may be invoked from the command line, e.g., as follows

```
./bin/run_dicefg.sh MCR_FOLDER ./tests/test1/configuration_val.xml
```

where MCR_FOLDER needs to be replaced with the installation folder of the Matlab Compiler Runtime, and *configuration_val.xml* is a test configuration file, similar to the one provided in the next section. After the execution of DICE-FG terminates, normally within very few seconds, the UML model(s) specified within the configuration file will be annotated with concrete value of the unknown parameters.

Detailed installation and running instructions for DICE-FG are available on the DICE-FG wiki at https://github.com/dice-project/DICE-Enhancement-FG/wiki/.

4.3.1. Configuration Files

We here focus on the specification of the input data that is requested to the user in order to use DICE-FG. The input parameters for DICE-FG tool are specified in a dedicated XML file. Examples are included within the DICE-FG distribution, including the one below:

```
<?xml version="1.0" encoding="utf-8"?>
<fg>
    <configuration>
        <parameter type='Verbose' value='1' />
    </configuration>
    <dataset>
        <period start='0' end='1492581' />
        <file type='ResourceList' path='./tests/test2/test-resources.json' />
        <file type='ResourceDataFile' path='./tests/test2/test-resdata.json' />
        <file type='ResourceClassList' path='./tests/test2/test-resclasses.json' />
        <file type='SystemDataFile' path='./tests/test2/test-sysdata.json' />
        <file type='SystemClassList' path='./tests/test2/test-sysclasses.json' />
    </dataset>
    <fitting type='fit-norm' flags=''>
        <resource name='resource1' flags=''>
            <output handler='uml-marte' path='./tests/test2/model.uml'>
                <metric name='qlen' class='class1' confidence='mean' param='$redT' type='hostDemand' />
            </output>
        </resource>
    </fitting>
    <estimation type='est-qmle' flags='warmUp=0'>
        <resource name='resource1' flags='numServers=1'>
            <output handler='uml-marte' path='./tests/test2/model.uml'>
                <metric class='class1' confidence='mean' param='$redT' type='hostDemand' />
                <metric class='class1' confidence='upper' param='$RT' type='hostDemand' />
                <metric class='class1' confidence='lower' param='$mapT' type='hostDemand' />
            </output>
        </resource>
    </estimation>
</fg>
```

The above configuration file specifies a complete DICE-FG analysis, consisting of a statistical distribution fitting step and an estimation analysis step. At the end of this execution, the parameters *$redT*, *$RT*, and *$mapT* in the UML model *./tests/test2/model.uml* will be replaced by concrete numbers.

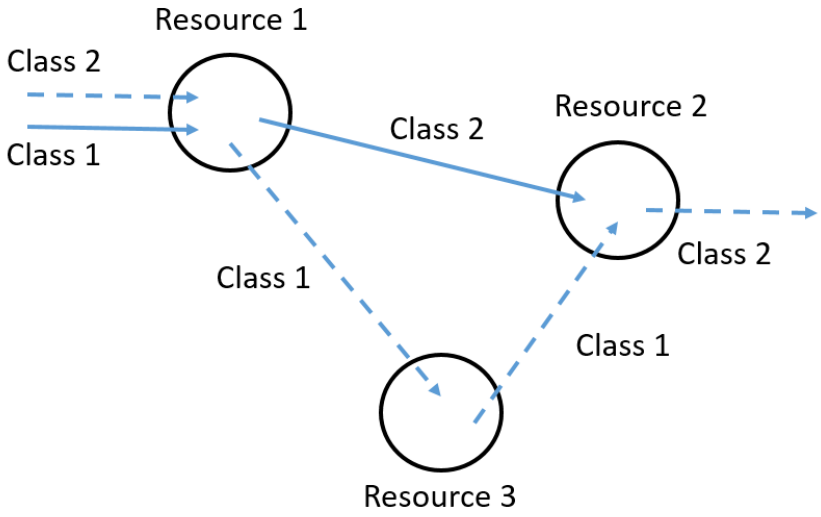
The configuration file relies on several XML element tree:

- The <*configuration*> tree specifies general configuration parameters of DICE-FG, such as the amount of data shown on the standard output.
- The <*dataset*> tree specifies the dataset to be loaded in memory at the beginning of the DICE-FG execution. Each execution of DICE-FG can rely only on a single dataset. The dataset will be loaded from MAT (Matlab native) or JSON files, transformed into an internal data structure, validated and in some case sanitized for erroneous or missing entries. A dataset is defined by a collection of files:
    - *ResourceDataFile* contains the measurements that are used for estimation or fitting, collected at the level of the individual resources that compose the system.

  - *ResourceClassList* is a list of text labels that assign names to different classes of jobs that arrive at the resources. It is assumed that properties of different job classes have been measured separately (e.g., for a NoSQL DB response times one may collect in *ResourceDataFile* separate measurements for read operations and write operations).
  - *ResourceList* lists the resources at which the measurements have been collected.
  - *SystemDataFile* contains the measurements which are used for estimation or fitting, collected across a collection of resources that compose the system. For example, the end-to-end response time is a property that typically depends on the traversal of multiple resources.
  - *SystemClassList* provides a list of system-wide classes. These can either be in 1-to-1 mapping with the *ResourceClassList* ones, or a combination therefore.

- The <*fitting*> tree defines a fitting analysis to be carried out with a specified algorithm (here **fit-norm**) on the given metric (here **qlen**) at the specified resource, which is specified using the <*resource*> tree.
- The <*output*> defines the handler in charge of writing the parameters to the UML models, in this example the UML MARTE handler.
- The <*estimation*> tree requires to estimate a missing parameter using statistical inference, for the given resource and metrics.
- The <*resource*> tree can be replaced by a <*system*> tree, which defines an estimation problem over a collection of resources, as opposed to a single resource. This requires the *SystemDataFile* and *SystemClassList* information.

More examples are provided within the DICE-FG release. We limit here to provide more details on the above notions of *System*, *Resource*, *SystemClass* and *ResourceClass* using the diagram below. The figure depicts and application composed of 3 resources:



- All resources process resource class 1 jobs.
- Resource 1 and Resource 2 both process jobs of resource class 2.
- A system class exists, composed of the dashed path.
- Four systems may be considered: (Resource 1, Resource 2), (Resource 1, Resource 3), (Resource 2, Resource 3), or (Resource 1, Resource 2, Resource 3, Resource 4).

The above definitions allow to specify in DICE-FG a variety of analyses, from estimating the requirements of individual resource classes at a specific resource, to fitting the response time distribution of a system class over the system resources. Such flexibility is useful to describe complex topology featured by DIAs, such as those based on stream processing systems like Storm.

We now discuss more in the details the parameters presented in the above configuration XML file and their allowed values.

Table 10: Input parameters of DICE-FG tool

Element tree/Element	**Element/Attribute**	**Description**
Configuration	*Verbose*	Controls the verbose level of the tool, allowed values: ● 0: silent ● 1: normal ● 2: debug
Dataset	*period*	Timestamps defining the time window for the data. The times can be logical (e.g., for simulation data) or physical (e.g., UNIX timestamp).
File	*ResourceDataFile*	Path to resource data provided in .mat or .json format.
File	*SystemDataFile*	Path to system data provided in .mat or .json format.
File	*ResourceClassList*	Path to input class file in .mat or .json format. The list includes only classes in the *ResourceDataFile*.
File	*SystemClassList*	Path to input class file in .mat or .json format. The list includes only classes in the *SystemDataFile*.
File	*ResourceList*	Path to input resource file in .mat or .json format.
Estimation	*type*	Algorithm to be used for estimation or fitting. Supported estimation algorithms are as follows: ● **est--ci**: inference of average execution times from response time data. The method requires the logging for *all* jobs, as opposed to periodic sampling. ● **est--ubr**: inference of average execution times from samples of average throughputs and average utilization in each sampling window. ● **est--qmle**: inference of average execution times from queue-length data. ● **est--qbmr**: inference of average memory usage from queue-length and aggregate memory data. ● **est--maxpopulation**: obtains the maximum population of jobs observed at the resource. ● **est--maxavgpopulation**: obtains the maximum of the samples of the average population of jobs observed at the resource. ● **est--extdelay**: inference of the mean external delay between submission of successive jobs to the resource. Most of the above algorithms are based on DICE-sponsored papers or state-of-the-art algorithms. We point to the DICE-FG wiki for references and a description of individual methods.
Estimation	*flags*	A string of text with custom options, see wiki for extended documentation.
Fitting	*type*	Supported fitting algorithms are:

		• **fit-norm:** fit data to a normal distribution. • **fit-gamma**: fit data to a gamma distribution. • **fit-exp**: fit data to an exponential distribution. • **fit-erl**: fit data to an Erlang distribution. • **fit-ph2:** fit data to a 2-state PH distribution • **fit-map2**: fit time series to a 2-state Markov modulated Poisson process
fitting	*flags*	A string of text with custom options, see wiki for extended documentation.
resource	*name*	Indicates the resource label, chosen within *ResourceList*, associated to the parameter of interest.
resource	*flags*	A string of text with custom options, see wiki for extended documentation.
Metric	*confidence*	Supported values: • **none**: the returned value does not make use of confidence intervals. • **upper**: the returned value of the parameters is taken at the upper end of the confidence interval (95% significance). • **lower**: the returned value of the parameters is taken at the lower end of the confidence interval (95% significance).
metric	*class*	Class label, from those read in *ClassList*, associated to the parameter of interest.
metric	*name*	Metric label, used to indicate to DICE-FG which metric should be fitted. The parameter is not required by estimation algorithms. See *Section 4.3.2.1* for supported values (e.g., **arvT** for arrival times).
metric	*type*	Parameter type from Table 10, e.g., hostDemand, extDelay, etc.
metric	*param*	Name of context parameter to be annotated in the UML model.
output	*handler*	• **uml-marte**: annotate UML MARTE parameters • **uml-dice**: annotate UML DICE parameters
output	*path*	Path to UML file. The file will be overwritten.

More details about required parametrization for each DICE-FG option is available at the DICE-FG repository: https://github.com/dice-project/DICE-Enhancement-FG/wiki/ .

4.3.2. DICE-FG input data format

To standardize the use of the estimation algorithms, we have adopted a common data format, from which each algorithm can select the data it requires to perform the estimation. We assume the data has been or is being collected for an application that provides a number of different services, grouped in service classes.

There are a total of K different service classes and M different resources. Data is collected either by averaging in time windows, or for individual requests, or both, and can be specific to a class or aggregate.

Data is associated to a subsystem of *E*<=*M* resources. If *E*=1, the data is stored in *ResourceDataFile*, conversely if it is associated to a subsystem of multiple resources (*E*>1) the data is stored in *SystemDataFile*. We discuss the two cases separately.

4.3.2.1. Resource data

The data format is a data structure (MATLAB cell array) with 11 rows and *M(K+1)* columns, representing M groups of (*K + 1*) columns. The *i*-th group of (*K+1*) columns represents the measurements for the *i*-th resource and uses the first *K* columns to describe data for each service class, while the last column is reserved for aggregate data. The column index of class *r* at resource *i* is therefore **idx=(*i*-1)*(K+1)+*r*,** while for the aggregate data at resource *i* it is **idx=*i*(K+1)** .

For each column, the information provided in each row is the following:

Table 11: Information provided in each row

Row ID	***Metric type***	***Unit of measure***	***AnalyzeMetric***	***Description***
1	**Sampling timestamp**	sec	**ts**	Holds the timestamps corresponding to the end of each sampling interval
2	**Utilization**	n/a, in [0,1]	**util**	Holds the average CPU utilization for each sampling window. Typically, only overall CPU utilization is collected, thus only the column *K +1* will hold an array, while the other columns will be empty.
3	**Arrival timestamps**	sec	**arvT**	Holds the timestamps of the arrival of each request to the resource.
4	**Response time**	sec	**respT**	Holds the observed response time (departure time minus arrival time) of each request
5	**Average response time**	sec	**respTAvg**	Holds the mean response time of the requests processed in each sampling window. If no requests of a given class are processed in a sampling interval, the corresponding entry in the array is set to zero.
6	**Average throughput**	jobs/sec	**tputAvg**	Holds the throughput observed for each service class in each sampling window. The throughput is computed as the total number of requests processed in the sampling interval, divided by the length of the interval (in seconds).
7	**Departure**	sec	**depT**	Holds the timestamps of the departure of each request from the resource.

	times			
8	**Queue-length**	jobs	**Qlen**	Holds the actual queue-length (number of jobs in the system) seen at sampling instants.
9	**Average queue length**	jobs	**qlenAvg**	Holds the average queue-length (number of jobs in the system) for each service class in each sampling period.
10	**JobId**	n/a, integer	**jobId**	Holds the id of the job that generated the sample (e.g., id of arriving job)
11	**Memory usage**	kB	**mem**	Holds the memory usage in each sampling window. More accurate results can be obtained if the memory is computed as the total memory usage minus the memory allocation due to operating system or other services running in the background.
12	**Average memory usage**	kB	**memAvg**	Holds the average memory usage in each sampling window. More accurate results can be obtained if the memory is computed as the total average memory usage minus the memory allocation due to operating system or other services running in the background.

4.3.2.2. System data

At the moment, DICE-FG supports only system-wise estimation for metrics recorded on the entire set of resources, i.e., $E=M$. For example, the end-to-end response time, the system throughput, and the total number of jobs in the system. The specification of data is again based on columnar data, with the first 11 rows as for the resource data. However, the following rows are also included:

12. **System matrix**: a binary matrix with M rows (resources) and K columns (classes). If element (m,k) is set to 1, then it is assumed that the measure includes the resource consumption of class-k jobs at resource m. It is possible to set to 1 several classes $k1, k2,..,kr$ on the same resource m, in this case all the class data of these classes will be summed to determine the resource consumption at resource m. The outside world (in open systems) and the delay node representing the external delay (extDelay/think time) of the users (in closed systems) are not included in the M resource rows.

13. **System routing matrix (optional)**: a probability matrix of order $K(M+1)$ specifying the route of requests across the system of resources. The upper-left submatrix of order KM represent the routing probabilities of the K classes across the M resources. For example, the first K rows represent the routing probabilities out of resource 1, with the k-th row representing class-k jobs. Note that this format allows one to specify class-switching, i.e., that a class leaving a resource can enter a resource into another class. The remaining entries represent probabilities of flows from/to the outside world (open topology) or to the delay node representing the external delay (extDelay/think time) of the users (closed topology).

Given the above routing matrix, the information provided in the first 11 rows is interpreted as follows:

Table 12: Information provided in the first 11 rows

AnalyzeMetric	*Interpretation in SystemData*
ts	Timestamps at the end of the sampling periods. For aggregate measurements, the sampling period ends when all the required metrics have been collected.
util	Percentage of admitted jobs in the sampling period, assuming a limited number of jobs can be admitted in the subsystem.
arvT	Timestamps of arrivals from node *M+1* into any of the M resources.
respT	Response time between arrival from node *M+1* to return to node *M+1*, for each request visiting the *M* resources (or a subset thereof).
respTAvg	Average value of response times seen during each sampling period.
tputAvg	Mean departure rate from any of the M resources to node M+1 seen during each sampling period.
depT	Timestamps of arrivals from any of the M resources to node M+1.
qlen	Number of jobs observed in the system at the end of each sampling period.
qlenAvg	Average number of jobs observed in the system at the end of each sampling period.
mem	Memory usage summed across the M resources as seen at arrival instants of jobs from node M+1.
memAvg	Cumulative average memory usage summed across the M resources as seen at the end of each sampling period.

4.3.3. Specifying DICE-FG input data via JSON

Two functions are provided with DICE-FG to convert the common data format to/from JSON. The common data format is composed by the **data, resources** and **classes** cell arrays. These can be converted to JSON from MATLAB using the command:

fg2json('hmr', resdata, sysdata, resources, resclasses, sysclasses)

Where ‘hmr’ is a use specified text prefixed. This will create five JSON files: *hmr-resdata.json*, *hmr-sysdata.json*, *hmr-resources.json*, *hmr-sysclasses.json* and *hmr-reclasses.json*. To reload these files into the MATLAB environment, DICE-FG uses the command:

[resdata, sysdata, resources, resclasses, sysclasses] = json2fg(folder, 'hmr')

Where folder is the path to the folder containing both JSON files, e.g., the output of MATLAB *pwd* command for the current directory. JSON files can also be specified directly, without use of MATLAB. Examples are provided with this tool release.

4.3.4. Integration with DMon

To perform the analysis, the DICE-FG tool first needs to obtain runtime data on the DIA from the DICE Monitoring Platform (DMon). Within DICE we have extended DICE-FG to operate with DMon and we report in this section the integration approach. The DICE-FG tool sends the following JSON query string to DMon to collect the runtime information (See the Figure 5).


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


Figure 5. Example of query (JSON format)

DICE Monitoring Platform will return a JSON string which includes CPU utilization, job information, etc. Figure 6 shows the example of obtained JSON results.


```
{
  "DMON": {
   "_shards": {
    "failed": 0,
    "successful": 2,
    "total": 2
   },
   "hits": {
    "hits": [
     {
      "_index": "logstash-2016.06.28",
      "_type": "yarn_jobstasks",
      "_id": "yarn-jobstasks",
      "_score": 1,
      "_source": {
       "jobs": [
        {
         "tasks": {
          "task": [
           {
            "finishTime": 1466544980408,
            "successfulAttempt": "attempt_1463611639333_0002_m_000004_0",
            "elapsedTime": 9807,
            "state": "SUCCEEDED",
            "startTime": 1466544970601,
            "progress": 100,
            "type": "MAP",
            "id": "task_1463611639333_0002_m_000004"
           },
          .....
 }
}
```


Figure 6. Example of obtained runtime data (JSON format)

This data is then used to automatically generate a valid set of input files for DICE-FG. This is done by loading in memory the JSON file and operating a basic parsing of individual metrics until recovering the required information by DICE-FG shown in Table 13.

5. DICE-FG Algorithms

In this chapter we define the technical advances to the DICE-FG tool developed as part of DICE. We first provide an overview and then summarize the algorithms part of DICE-FG.

5.1. Overview

The work done as part of DICE to extend the FG tool baseline to the specific needs of DIA development has focused on these following:

- **Memory usage patterns**. These are critical to understand the performance of in-memory operations. For such operations, the DIA designer wishes to avoid memory swapping, which can compromise performance and reliability, as it can render the application so slow to be unavailable. Therefore, the DICE-FG methods in DICE consider this metric. We describe the support of DICE-FG for this feature in Section 5.2.
- **Confidence intervals on estimates**. One of the requirements of the Enhancement tools involves providing information about the uncertainty on estimated requirements. This information can provide a measure of confidence on the quality of the inference and, for example, suggest to the developer and QA engineer that more test experiments are needed to gain confidence about the parameters of the simulation and optimization models. We have investigated this problem systematically, and we report results in Section 5.3.
- **Applicability to DIA of baseline algorithms for mean execution times**. DIAs are also ordinary Java-based software systems, hence several methods that apply to resource consumption estimation in ordinary applications can also be applied as-is to DIAs. However, we are not aware of systematic studies in the literature about this. During the second year we have investigated this research direction by applying DICE-FG to a case study being developed jointly with the MIKELANGELO H2020 project in the context of Cassandra performance engineering. We report in Section 5.3 and the appendix initial results for this line of work.
- **Distribution of execution times**. Compared to a canonical three-tier application, a DIA typically features smaller concurrency levels, since each operation is more intensive in terms of volumes of data processed or memory usage. Therefore, it is often the case that one has at disposal precise measurements about the running times of an application, which are not inflated by contention overheads. In this setting, it is possible to provide a distributional characterization of execution times, which increases the accuracy of simulation and optimization. In Section 5.4 we describe the extension of DICE-FG developed in DICE to fit execution time data into phase-type distributions.

5.2. Inference of Memory Patterns

5.2.1. Methodology

The main question we wanted to answer in our study on estimating memory consumption patterns is as follows: *Do we need to develop a full-fledged memory consumption model for the DIA or can we devise the same information by knowing the average behavior of each single activity from test runs*?

The answer we have found is that characterizing the behavior of individual activities and considering the effect of the superposition of multiple activities appears sufficient to capture memory behavior as a whole. This means that simple tests in isolation for each application activity can provide to DICE-FG the necessary data to annotate the UML diagrams with memory consumption parameters. This conclusion has been validated on a case study involving in-memory processing, in collaboration with an external stakeholder interested in the DICE results: SAP, which now runs a core business in Big data analytics through their HANA solution. The results of this work are described in [27] and are here summarized.

In this study we have run test experiments on a HANA testbed using an analytic workload (TPC-H) representative of business analytics applications. The experiments have been carried out an IBM x3950 X6 server running SLES 11 SP3. This test server features 8 processor sockets with a total of 120 physical cores and provides a total of 6 TB RAM corresponding to 750GB per socket. 64 experiments have been carried out, varying the factors shown in Table 14.

Table 13: Description of the factors

Factor	Description
x_1	Client think time (in seconds)
x_2	Number of active tenant databases
x_3	Database scale factor, uncompressed (in GB)
x_4	CPU-sockets a tenant database is assigned to
x_5	Degree of resource overlapping between two tenants
x_6	Number of clients/users per tenant

Each factor is assigned one of two possible levels, as shown in Table 15, which considers scenarios of varying complexity, involving multiple tenant databases.

Table 14: Lower and upper bound of each factor

Bound	x_1	x_2	x_3	x_4	x_5	x_6
Lower Bound (X_l)	0	2	30	2	full	8
Upper Bound (X_h)	10	4	300	4	isolated	16

Throughout each experiment, we have recorded total memory consumption of the analytic workload. Moreover, we have carried out isolation experiments, in which the memory consumed by each individual request was monitored, without memory interference from the other requests. Our analysis considered two aspects:

Memory inference via stochastic models. Whether the analytic models used in DICE, in particular those based on queueing network models and JMT, could correctly predict the total memory consumption using only the memory consumption information of individual requests. This is done using the formula:

$$M_i = \sum_{c=1}^{C} \frac{Q_{c,i}}{l_c} m_c$$

in which M_i represents the memory consumed at the i-th time interval, $Q_{c,i}$ is the mean number of jobs of class c in execution in the system during interval i, m_c is the memory consumption of jobs of class c obtained by isolation tests, and l_c is the parallelism level of a class c job. This memory model depends on the mean number of jobs in execution, $Q_{c,i}$, which can either be obtained by direct measurement or via simulation, in case of predictive studies. In this study, we use $Q_{c,i}$ computed by simulation and try to match with the above formula the memory consumption observed in the real system. We point to [27] for a description of the JMT queueing network model used to describe this application and report here the qualitative conclusions of our study.

Memory inference via regression. DICE-FG obtains from the above expressions the memory consumptions m_c. We studied how these can be used for model-based prediction compared to developing a memory consumption model based on response surfaces, which interpolate the experimental results through nonlinear regression methods. Clearly, such surfaces do not require the development of a stochastic model, hence they are simpler to fit to observations. DICE integrates the fitting of regression surfaces as part of the Anomaly Detection tool, developed in WP4, and we have used this feature to model memory consumption on this application, without knowledge of the memory behavior of

individual requests. We consider in particular response surfaces that encompass the six configuration parameters varied in the experiments, trained on a fraction of the data available (8, 16 or 32 experiments).

5.2.2. Obtaining memory weights from DICE-FG

The main takeaway is however that simulation is surprisingly accurate, in spite of the fact that there is an indirect estimation as we are relying just on data obtained during isolated runs. This implies a simple methodology for estimation of memory via DICE-FG:

- The user runs a set of isolation experiments for each job type, in which memory usage is obtained by DMon.
- DICE-FG retries this data and annotates the UML model with average memory consumption levels observed for each request in isolation.

The inference step is delegated to the computation of memory consumption using inference formulas such as the one provided above for M_i. If the performance and reliability analysis is carried out with JMT, it is however possible for the user to obtain a direct estimate of M_i by using a metric called *FCR - Mean Memory Consumption* that has been contributed by the DICE team to JMT.

An experimental validation of the predicting capabilities of the memory estimation features integrated in DICE-FG is given in Appendix A.1 of the present document.

5.3. Inference of Mean Execution Times

During M1-18 the DICE-FG tool has been part of a formal collaboration between the MODAClouds and DICE projects. The problem of inferring mean execution times has been systematically investigated in MODAClouds and a set of algorithms have been developed to obtain inference of mean execution times from utilization data, response time data, and queue-length occupancy data. A paper, sponsored by both projects, has been written with the aim of comparing the accuracy and execution times of the integrated methods [25] and a tutorial presented at ACM/SPEC ICPE 2016 to train the user community [26]. We point the DICE users to these material for an overview of the inference methods of mean execution times.

After the conclusion of MODAClouds, we have developed validation of applicability of the methods to DIA, as part of a case study on Apache Cassandra, currently being carried out as part of a collaboration between DICE and the MIKELANGELO H2020 project.

We report this case study in Appendix A.2, where we show the accurate prediction results obtained on Apache Cassandra thanks for the parameterization generated by DICE-FG.

5.4. Confidence Intervals on Mean Execution Times

A limitation of the MODAClouds-FG baseline tool is the fact that its estimation algorithms are designed to return confidence intervals on the produced estimates. This is undesirable, since QA engineers need to get an understanding of the quality of execution time estimates. We have addressed this limitation in DICE-FG by developing a new estimator of mean execution times, called QMLE, which has been integrated in DICE-FG and can return rigorous confidence intervals on the estimated mean execution times. We here briefly overview the QMLE method and its confidence interval generation feature.

The QMLE method, specified in DICE-FG via the **est-qmle** option, provides an approximate closed-form estimator for mean execution times in software systems. Assume that a dataset ***D*** has been collected, containing measurements of the number of jobs in execution, for each type and at each node of the DIA. We denote by $\widetilde{Q}_{ij}$ the mean number of requests of type *j* running at node *i* obtained via the monitoring tool, by N_j the total number of jobs running in the system, and by $\theta_{0,j}$ the mean think time between submission of successive jobs of type *j* by a user. The QMLE computes the mean execution times of type *j* at node *i*, denoted by θ_{ij}^{bs} , as [24]

$$\theta_{ij}^{bs} = \frac{\widetilde{Q}_{ij}(\boldsymbol{D})}{(N_j - \sum_{k=1}^{M} \widetilde{Q}_{kj})} \frac{\theta_{0,j}}{(1 + \sum_{h=1}^{R} \widetilde{Q}_{ih} - \widetilde{Q}_{ij}(\boldsymbol{D})/N_j)}$$

Here the *bs* superscript indicates that the formula of the estimator is obtained by relying on a theoretical approximation known as Bard-Schweitzer algorithm, we point the interested reader to [24] for details. The above formula allows to compute the 95% confidence interval of the mean execution times θ_{ij}^{bs} using the expression $\pm c\sqrt{(\boldsymbol{I}(\hat{\boldsymbol{\theta}})^{-1})_{ij,ij}}$ where c=1.96 and the term under square root is the element in inverse of the Fisher information matrix associated to the maximum-likelihood estimator of θ_{ij}^{bs} .

We provide explicit formulas for the Fisher information matrix in [24] and these have been integrated in DICE-FG tool as part of the DICE activities. A validation of the correctness of the expressions has been given in [24], we point the interested reader to the paper. In the paper we show that the results of the above confidence interval expression are exact.

In order to apply the above results to DICE-FG estimation results, the user simply needs to specify the *Confidence* parameter in the input XML, as discussed in Table 11. For example, if *Confidence* is set to upper then the confidence interval half-width at 95% significance level is automatically added to the mean execution time. The same principle is also applied to fitting of statistical distributions, for example with the **fit-normal** option confidence intervals are generated for both the estimators of the mean and standard deviation for the normal and used to correct the estimate of these parameters according to the setting of the *Confidence* parameter.

5.5. Estimation of Execution Time Distribution

5.5.1. Contribution and innovation

The synchronization intrinsic in the operation of DIAs is generally sensitive to the execution times of the activities to be synchronized. Consider for example the Map phase of Hadoop/MapReduce involving the processing of two tasks in parallel: after the first mapper completes its task, the platform awaits for the second mapper to also complete it task before moving to the Reduce phase. Clearly, if the second job incurs a long execution time - in statistical terms, it has execution times featuring a *long tail* - the system will be blocked for a longer period of time than a system with a *light tail* in the execution times.

The dataset has been collected by PMI using the Cineca infrastructure, a description of this dataset is provided in deliverable *D3.8 - DICE Optimization Tools* in Section 6.1. The cumulative distribution function of the execution times of the Map and Reduce phases is shown in the Figure 7.

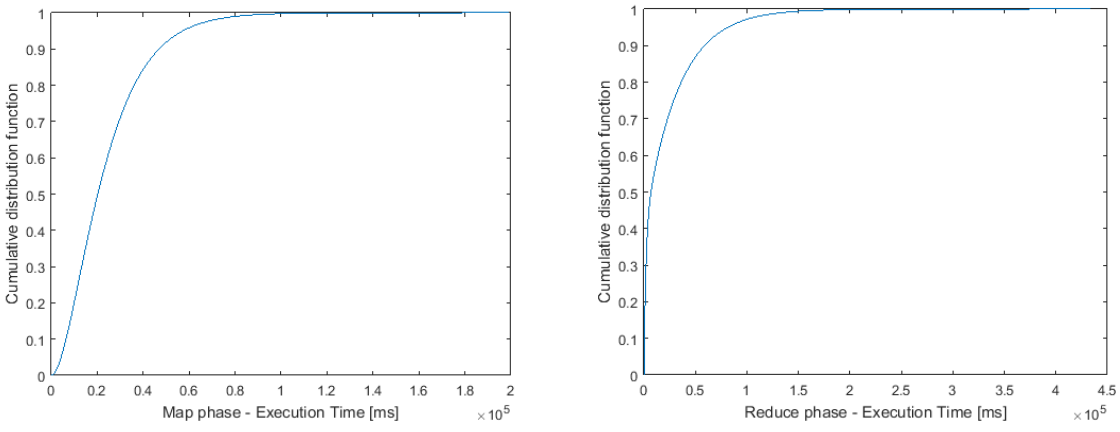


Figure 7. Execution time distribution in MapReduce experiments

It is interesting to note that in both cases the execution times of the phases is quite long tailed, with the cumulative distribution function extending over a wide range of values before converging to 100%. This essentially means that large samples are observed in the execution time of the application, which albeit rare can induce significant deviations from the expected execution times. As noted above, this can have a large impact in the modelling of synchronizations in DIAs.

As shown in Table 11, DICE-FG accepts several algorithms for fitting input data to parametric distributions, including Normal, Gamma, Exponential, and Erlang. We also added support for Markov-chain based distribution modelling, via the **fit-ph2** and **fit-map2** options. We here review just the former fitting option and point the interested reader to the DICE tutorial at ICPE 2016 for an introduction to the Markovian arrival process fitted with the **fit-map2** option [26].

5.5.2. Fitting distributions using phase-type models

To address the problem, we have added to DICE-FG the possibility of fitting execution time distributions. Our solution relies on phase-type distributions, which are a class of Markov models compatible for use with Stochastic Petri nets and Queueing networks, which are the two classes of stochastic models used in DICE.

A **phase-type distribution** (PH) is a model of a statistical distribution specified in terms of a continuous-time Markov chain (CTMC). A CTMC is a classic dynamical model, where one specifies the transitions rates between a set of m states in a matrix called *infinitesimal generator* **Q**, in which the element in position (i,j) represent the instantaneous rate of change from state *i* to state *j* and the diagonal elements are set so that each row sum is zero. A PH extends this notion by treating the m-th state as the **absorbing state**, i.e. a state where the execution of the dynamical model terminates, and by using the distribution of the time to enter this absorbing state as a tool to model arbitrary distributions. One specific advantage of this class of models is that it is easy to couple with stochastic Petri nets and queueing network models. An illustration of this notion is given below, where the **T** submatrix represent the transition rates between ordinary states, whereas the **t** subvector represents the rate of jump to the absorbing state.

$$\mathbf{Q} = \left[\begin{array}{c|c} \mathbf{T} & \mathbf{t} \\ \hline \mathbf{0} & 0 \end{array}\right]$$

Figure 8. Specifying the matrix T of a PH distribution

In PH, an arbitrary statistical distribution is modelled by the fitting the function $1 - \boldsymbol{\alpha} e^{\mathbf{T}t} \mathbb{1}$ where $\boldsymbol{\alpha}$ is an arbitrary probability vector to be fitted to the data, $e^{\mathbf{T}t}$ is the matrix exponential of **T** evaluated at point *t*, and $\mathbb{1}$ is a column vector of ones. Based on the above definitions, it is easy to see that in order to fit a PH distribution one needs to assign the rates in the matrix **T** and vector $\boldsymbol{\alpha}$ automatically, up to matching the desired cumulative distribution function.

5.5.3. Fitting phase-type models using DICE-FG

We have added to DICE phase-type moment-matching methods by adapting and integrating in DICE-FG the algorithms available in the KPC-Toolbox[2], an open source fitting toolbox maintained by the IMP team. The KPC-Toolbox is able to automatically fit a PH distribution based on the empirical moments of the input trace. This is well-suited to fit long-tailed data, since moment-based matching is known to have a high-quality fit of the tail, due to the sensitivity of means to outliers, whereas it is generally less accurate on the distribution body. We point to Appendix A.3 for the experimental validation of this feature on a data-intensive application based on Hadoop/MapReduce.

[2] https://github.com/kpctoolboxteam/kpc-toolbox

6. Performance & Reliability Anti-Pattern Detection

Task T4.3 in DICE aims also at exploring the possibility of defining an APR methodology for DICE models. It has been recognized since proposal stage that this feature is treated as an experimental one, given a shortage of results in this space in the research literature on this topic. Therefore, we have focused the initial work on the APR tool on outlining the methodology to follow. In this section, we review the result of this initial investigation on techniques for detection of performance anti-patterns. Then we report on initial experiments we have conducted that suggest viability of the approach. Finally, we discuss the approach that we will follow in the next period in order to concretely develop the APR tool.

6.1. Technique Review of Anti-Pattern Detection

In software engineering, anti-patterns are recurrent problems identified by incorrect software decisions at different hierarchical levels (architecture, development, or project management). Software AP are largely studied in the industry. They are catalogued according to the source problem and a generic solution is suggested [2], [3], [4]. The increasing size and complexity of the software projects involves the rising of new obstacles more frequently. For that reason, the identification of AP at the early steps of the project life cycle saves money, time and effort.

In the context of big data technologies, which often executes thousands of tasks with gigabytes of data, the impact of AP in performance is even more evident. It becomes crucial to detect and solve the software pitfalls and bottlenecks that hamper the performance and scalability of the system. The benefits of introducing performance AP in Cloud environments are discussed in [5].

Most of the performance AP in DIAs are related to the partition and distribution of the data and the computations. The definition of execution pipelines and the selection of the degree of parallelism is usually carried out at the architecture or design level. However, the discovery of potential dangers when conceptualizing and designing a new system requires additional information (i.e., expected execution time of the code or the number of resources or components) for calculating the performance metrics that will warn the appearance of performance AP.

Recent works follow the same schema presented in [6] for automating the detection and solution of performance AP (see Figure 9). The approach consists of 1) modeling the system with a high-level description language, 2) transform it to a performance model through a Model-to-Model (M2M) transformation guided by performance annotations, 3) the simulation of the performance model for getting performance metrics, 4) the interpretation of the results for finding performance AP with respect to the structure of the system, and finally 5) feedback the original model for solving the problems. A software refactoring will usually improve the software structure and potentially solve the flaws.

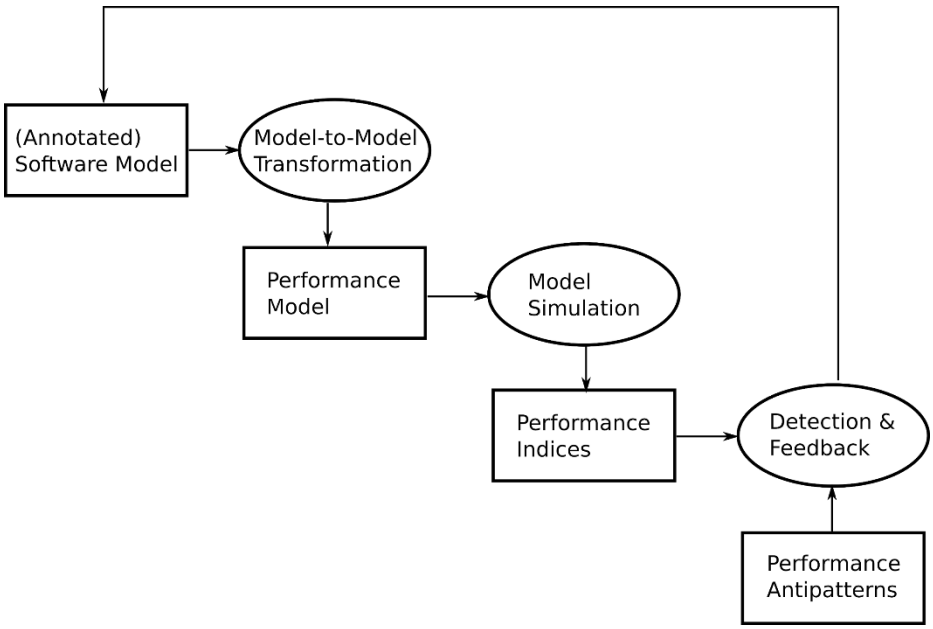


Figure 9. An overview of Anti-Patterns detection process

For instance, the authors in [7] have applied a M2M transformation from UML diagrams annotated with MARTE profiles into queuing networks. They define the AP in terms of OCL rules that are evaluated at the UML level. Next, the paper [8], [8] use the Palladio Component Model (PCM) [10] for describing component-based software architectures, extended queuing networks as performance models, and performance AP defined by a set of rules and actions. Finally, the work in [11] presents a Performance Anti-Pattern Modeling Language (PAML) for models described in the Architecture Description Language (ADL). These papers differ on the modeling language, the performance model, the language that they use for expressing AP, and the AP that they can detect and solve.

6.2. Our Approach

Based on the above technique review of anti-patterns detection, we propose to follow a similar methodology for automatically detecting and solving performance problems.

The first option is using UML diagrams annotated with DICE profiles as modeling language and Petri nets as performance models. Our goal is the integration of this approach inside the DICE framework in order to take advantage of all the research and tools developed in the project. We suggest the introduction of model checking technologies for the automatic detection of performance AP over Petri nets. The detection of performance AP using model checking is a novelty with respect to previous works.

Model checking is a paradigm stemming from computer science based on temporal logics which has been successfully applied in industry for system modeling and verification [12]. The model checking process consists of three phases: modeling both the system and properties with appropriate description languages, running the verification (checking the property validity with a model checking software) and analyzing the results (returning counterexamples if the property fails). Given a model and a set of properties, the verification process is completely automatized by a generic model checking tool. In our context, the models are the Petri nets obtained from UML diagrams; and a performance AP is a property that we desire to investigate if it is present in the model or not.

Thus, the next step consists of obtaining a formal representation for both the model and the performance AP. On the one hand, the DICE transformation tool-chain allows obtaining performance models (e.g., Petri net) from UML models stereotyped with the DICE profile for different big data technologies (see the DICE Deliverable 3.1-Transformations to Analysis models [13]). The Petri nets that result of the transformation are annotated with performance information (i.e., estimated execution time, number of resources or cores, etc.); and allow the computation of performance metrics (e.g., response time of a

server). These models are more suitable than UML diagrams for executing a model checking analysis. Tools such as GreatSPN [14] allow the verification of properties expressed in terms of temporal logic (e.g., CTL [15] or CSLta [16]) over Petri nets.

On the other hand, current efforts try to formulate the performance AP in terms of a first-order logic [17]. The temporal logics used by the verification tools are propositional formulas qualified in terms of time. In this kind of logics, the time is used for imposing a causal relationship between two set of states of the model determined by a propositional equation. Therefore, all properties expressible in first-order logic can also be expressed in temporal logic.

We are also considering another option which is similar to some methodologies [7,17,18]. The idea is translating the high level specification - UML models which annotated with DICE profiles into quantitatively analysis performance models – Queueing Network models (QN) [19]. Currently, we are investigating PUMA [20] which might help to translate the UML models to QN. After the QN model is generated, we can easily obtain the performance indices of interest (i.e., response time, throughput, etc.). Since performance anti-patterns problem and solution are usually described in natural language, we need to use a formal definition for the performance anti-patterns for anti-pattern detection. Like we mentioned before, there are several ways to define the performance anti-patterns, for example, OCL rules, first-order logic. We are currently doing the research on these areas to see if AP rules can be implement into our model. Once the anti-patterns are detected, we need to provide quick and efficient feedbacks to refactor our models. In our previous work [21,22,23], we developed an approach to refactoring cloud-based applications for reducing the total costs while optimizing the allocation of software components. Though DICE project focused on the Big-data application which is not the same as cloud application (e.g., the runtime information of infrastructure, platform, and application are different), the proposed approach also considers both the hardware and software knowledge to minimize the costs of cloud resources and it will be a valuable reference to the Big-data application.

Once the anti-patterns are detected, the Enhancement tool will generate the feedback and refactoring (manually or automatically) the architecture model. The refactoring decisions will help to modify the application. Table 16 shows some popular refactoring decisions we are considering to use.

Table 15: Popular refactoring decisions

Refactoring Decisions	Description
Replacement	A software component (e.g., an Application Server) is replaced with a different software component that provides the functionalities of the replaced component (e.g., another Application Server from a different vendor).
Merge	Two distinct software components (e.g., a Web Server and an Application Server) are replaced with a software component that provides the functionalities of the two replaced components.
Reassignment	It is the functional separation of a software component instance into two ones responsible for different classes of requests (e.g., an instance of Application Server is divided into two instances: one to register new users and one to authenticate new users).

Furthermore, deployment decisions (i.e., allocation decisions for each software component to a set of resources.) and design constraints (i.e., limiting the application of refactoring and deployment decisions when a software component cannot be replaced or replicated) are also considered for supporting architecture refactoring.

In order to synthetically describe the above approaches, Table 17 gives a conceptual idea of core processing of how we might implement the performance analysis, anti-patterns detection and refactoring.

Table 16: Anti-pattern and refactoring processing

General process	Approach
Annotated Model	UML diagrams annotated with DICE Profile
Performance Model	Investigating current performance model, e.g., Petri nets, Queueing networks
Performance Indices	E.g. Response Time, Throughput, Resource Demand
Performance Anti-patterns	Investigating current anti-patterns formal definition, e.g., OCL rules, first-order logic, code level
Results Interpretation & Feedback Generation	Refactoring architecture models according to detection and solution of Anti-patterns

In summary, the methodologies that we will follow for the detection and fixing AP issues in a big data context consist of:

- The description of the system using UML diagrams annotated with the DICE profile.
- The identification and selection of the more important AP for the analysis in big data systems.
- The transformation of UML diagrams into performance models expressed as Petri nets or queueing networks, relying on transformations developed in task T3.1 and ad-hoc transformations for APR.
- The formalization of performance AP in code level or with a logic suitable for model checking.
- The evaluation of the performance AP in the model.
- If the verification tool discovers that the performance AP is present in the system, we execute a refactorization of the software that mitigates the flaw.

6.3. Initial Work on Refactoring Methods

In order to reduce the total costs for running cloud-based applications while fulfilling service level objectives (SLOs), we investigated a model-based approach for optimizing the costs of running cloud-based applications. We used model-driven application refactorings, i.e., experimenting software alternatives that optimize the application model, to minimize the cost of deploying them in the cloud [28].

This approach focuses on the decision-making steps of the iterative process illustrated in Figure 10. The starting point is constituted by the system model, which depends on the application and the environment (e.g., expected number of users, expected price fluctuations, etc.), while the output is the information on how to deploy the application.

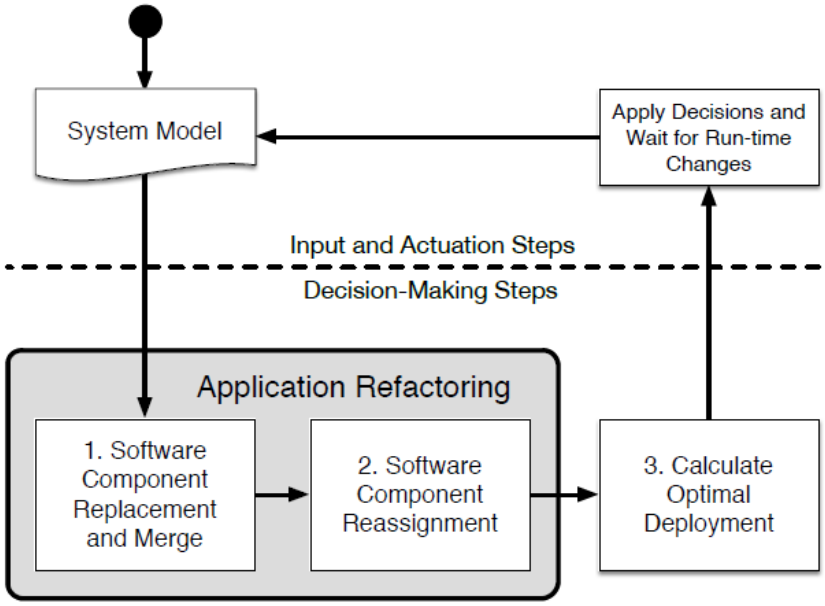


Figure 10. State diagram of our iterative approach

We identified three main steps of our decision-making process: (i) apply to the initial model the merge and replace refactorings to reduce the overall computational needs of the system; (ii) reassign requests and create component replicas to reduce the complexity of the models; (iii) calculate an optimal deployment by deciding which resources to rent and how to map components to them. After the third step, we obtain a decision on the desirable configuration of the system that minimizes the overall costs, which may be used to perform a reconfiguration of the deployed system. Since the application requirements and the environment may change overtime, we expect our approach to be implemented as the decision-making part of a loop in which the optimization and adaptation processes are repeated when there are significant changes in the system model.

We evaluated our approach on a cloud-based distributed application [29], which represents an enterprise-level business to business e-commerce workload of a realistic complexity. This represents an initial step to deliver a proof-of-concept on a class of applications that are well understood, the next step will be to focus on a specific data-intensive application and general the approach. The system is composed of an Application Server, and a Database subsystem. Figure 11 reports the results we obtained by modifying the design constraints on component replicability. Additional refactoring analyses are presented in [20].

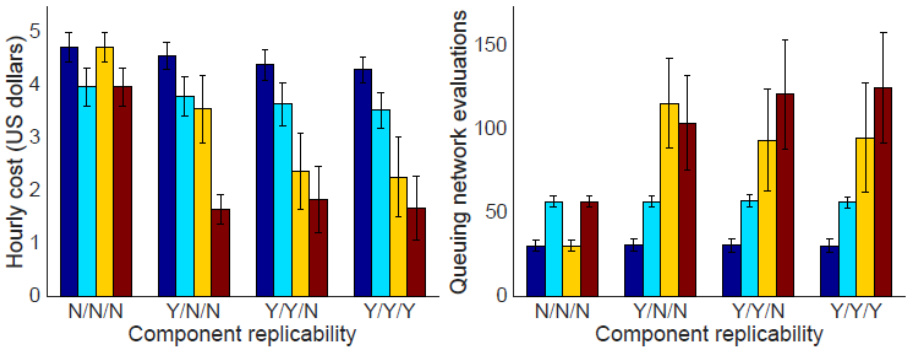


Figure 11. Varying the design constraints in terms of component replicability ("Y" yes, "N" no). Components are ordered in this way: Application Server, Database Server, Database I/O

The experiments results show that our approach is able to reduce the costs in all scenarios up to 60% when compared to an approach that does not use model-driven application refactoring. In most of our

experiments the reassignment refactoring is typically responsible for 50% of such improvement, while the replacement refactoring (which depends on the alternative components provided as input) is responsible for the remaining 10%. The cost for this improvement is paid in terms of additional QN evaluations, which has shown an increase of up to 4 times. However, since our QNs are evaluated in a matter of seconds in our system, and the increment in convergence speed is constant, we expect our approach to be fast enough to be used to drive periodical reconfigurations of the system. The MATLAB source code and the Amazon EC2 price traces we used to perform these experiments are available on Zenodo (cf., [20]), these provide a baseline for the implementation of the APR module.

The above work is an “exploratory” investigation which is based on our chosen cloud applications, we are currently thinking to borrow some above ideas to help us to perform the architecture refactoring for Big-data applications of DICE project.

7. Conclusion and Future Plan

7.1. Achievements

In conclusion of this deliverable we summarize the key achievements of this first release of the ENHANCEMENT_TOOLS:

- A module developed in the MODAClouds EU project, named MODAClouds-FG, has been adopted as a baseline for the estimation of mean execution times of jobs in DIAs.
- We have demonstrated the applicability of existing methods present in DICE-FG to DIAs on three case studies involving:
    - Apache Cassandra
    - Apache Hadoop/MapReduce (see also deliverable *D3.8*, Section 6).
    - An in-memory database system (SAP HANA).
- DICE-FG has been extended along several dimensions that are important to model DIAs:
    - Estimation of memory consumption.
    - Estimation of execution time distributions.
    - Computation of confidence intervals for mean execution times.
- We have investigated the technical approach for the future Anti-Patterns Detection and Refactoring (APR) tool, and demonstrated the promise of the methodology on a case study involving an enterprise cloud application.

The DICE FG tool is available online on DICE's Github repository. The following are the main links:

- DICE-FG Source Code: https://github.com/dice-project/DICE-Enhancement-FG
- DICE-FG Documentation: https://github.com/dice-project/DICE-Enhancement-FG/wiki

The APR module capabilities are planned for official release by M30, with the final version of this report.

2.2. Summary of progress at M18

The following table summarize the status of requirements implementation at the end of reporting period (M18).

Table 17: Requirements for Enhancement Tool

Requirement	Status at M18
R4.11: Resource consumption breakdown	✔: the DICE-FG module is capable of extracting resource consumption data (memory, CPU time) for individual tasks at arbitrary nodes. The estimated data breaks down the usage of individual resources through job types that visit the resource.
R4.12: Bottleneck identification	✔: by estimating the true execution times of requests, sanitized from contention overheads, the DICE-FG makes it trivial to identify bottlenecks. That is, the node with the largest mean execution time will be the bottleneck resource for a job type. Such feature is going to be completed by adding bottleneck identification capabilities in the APR module.

R4.13: Semi-automated anti-pattern detection	✔: an initial architecture and high-level approach defined, and initial proof-of-concept defined. The APR tool is set for initial prototype release at M24, and finalization with the next version of this deliverable at M30 (deliverable D4.6).
R4.17: Enhancement tools data acquisition	✔: We have interfaced DICE-FG module with the DMon platform. APR module will not need direct access to the DMon. More metrics will be accessed in the feature to extend the breadth of the automatic UML parameterization.
R4.18: Enhancement tools model access	✗: this feature is an integration feature to be developed in the next period. Currently integration is operated manually, in the future it will be automated.
R4.19: Parameterization of simulation and optimization models	✔: we have conducted validation studies on Hadoop/MapReduce (c.f. D3.8, Section 6), Cassandra, and SAP HANA that illustrate the ability of the DICE-FG module to provide good estimates of parameters.
R4.27: Propagation of changes/automatic annotation of UML models	✔: DICE-FG can successfully modify UML models by annotating parameters. The APR module is planned to introduce changes in the UML models, but this feature is not available yet as the tool is due for release later.
✔- implemented at M18 ✗ - not implemented yet, due at M24/M30 ✔ - partial accomplishment at M18	

7.2. Future work for DICE-FG

- By M24 we plan to finalize the integration of DICE-FG with the DICE toolchain in terms of DICE profile model acquisition and automated annotation.
- By M30 we plan to extend the features of DICE-FG towards supporting the modelling of technologies supported by DICE but that have been investigated to a limited extent in DICE-FG tool. In particular, technologies such as Spark and Storm where the computation proceeds according to a direct-acyclic graph (DAG), in which processing activities are forked and joined (i.e., synchronized) in predefined ways (as in the Map and Reduce phase sequence in Hadoop/MR) or according to user-specified patterns (as in Storm, Spark, and Apache Tez). Estimation of resource consumption that ignores DAG topologies can neglect the influence of blocking in the estimation of execution times.

The current implementation of DICE-FG will be included in the first release of the DICE framework at M24.

7.3. Future work for APR

- By M24 we plan to 1) implement the model transformation between the UML models with DICE Profiles annotation and performance model, and 2) formally specify the AP problem and solution for the selected AP.
- By M30 we plan to 1) implement performance AP verification of the model, and 2) provide reasonable refactoring decisions (e.g. replacement, reassignment) to achieve the architecture refactoring manually or automatically.

The M30 implementation of APR will be included in the second release of the DICE framework at M30.

References

[1] DICE Consortium, Requirement Specification M16 update (Deliverable 1.2 companion document), 2016, M16 updated version. http://wp.doc.ic.ac.uk/dice-h2020/wp-content/uploads/sites/75/2016/05/Requirement-Specification-M16.pdf

[2] Smith, C. U. and Williams, L. G. (2000). Software performance antipatterns. In Workshop on Software and Performance, volume 17, pages 127-136.

[3] Smith, C. U. and Williams, L. G. (2002). New software performance antipatterns: More ways to shoot yourself in the foot. In Int. CMG Conference, pages 667-674.

[4] Smith, C. U. and Williams, L. G. (2003). More new software performance antipatterns: Even more ways to shoot yourself in the foot. In Computer Measurement Group Conference, pages 717-725.

[5] Trubiani, C. (2015). Introducing software performance antipatterns in cloud computing environments: Does it help or hurt? In Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, pages 207-210. ACM.

[6] Cortellessa, V. (2013). Performance antipatterns: State-of-art and future perspectives. In European Workshop on Performance Engineering, pages 1-6. Springer.

[7] Cortellessa, V., Di Marco, A., Eramo, R., Pierantonio, A., and Trubiani, C. (2010). Digging into uml models to remove performance antipatterns. In Proceedings of the 2010 ICSE Workshop on Quantitative Stochastic Models in the Verification and Design of Software Systems, pages 9-16. ACM.

[8] Trubiani, C. and Koziolek, A. (2011). Detection and solution of software performance antipatterns in palladio architectural models. In ACM SIGSOFT Software Engineering Notes, volume 36, pages 19-30. ACM.

[9] Trubiani, C., Koziolek, A., Cortellessa, V., and Reussner, R. (2014). Guilt-based handling of software performance antipatterns in palladio architectural models. Journal of Systems and Software, 95:141-165.

[10] Becker, S., Koziolek, H., and Reussner, R. (2009). The palladio component model for model-driven performance prediction. Journal of Systems and Software, 82(1):3-22.

[11] Cortellessa, V., De Sanctis, M., Di Marco, A., and Trubiani, C. (2012). Enabling performance antipatterns to arise from an adl-based software architecture. In Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on, pages 310-314. IEEE.

[12] Grumberg, O. and Veith, H. (2008). 25 years of model checking: History, achievements, perspectives. Springer, Berlin.

[13] Consortium, T. D. (2016). Transformations to Analysis models. Technical report, European Union's Horizon 2020 research and innovation programme.

[14] Dipartimento di informatica, Università di Torino (Dec., 2015). GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets. url: www.di.unito.it/~greatspn/index.html

[15] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Transactions on Programming Languages and Systems (TOPLAS), 8(2):244-263.

[16] Donatelli, S., Haddad, S., and Sproston, J. (2009). Model checking timed and stochastic properties with CSLTA. IEEE Transactions on Software Engineering, 35(2):224-240.

[17] Cortellessa, V., Di Marco, A., and Trubiani, C. (2014). An approach for modeling and detecting software performance antipatterns based on first-order logics. Software & Systems Modeling, 13(1):391-432.

[18] Xu, J. (2010). Rule-based automatic software performance diagnosis and improvement.Perform. 67: 585–611.

[19] Baskett, F., Chandy, K. M., Muntz, R. R., and Palacios, F. G.(1975). Open,Closed, and Mixed Networks of Queues with Different Classes of Customers. J. ACM, 22(2):248–260.

[20] Woodside, M., Petriu, D.C., Petriu, D.B., Shen, H., Israr, T. and Merseguer, J. (2005). Performance by Unified Model Analysis (PUMA), Proc. WOSP '2005, Mallorca, 1-12.

[21] Dubois, D. J. and Casale, G. (2015). Autonomic Provisioning and Application Mapping on Spot Cloud Resources. In ICCAC ’15, 57–68.

[22] Dubois, D. J. and Casale, G. (2016). OptiSpot: minimizing application deployment cost using spot cloud resources. Cluster Comp., 2016.

[23] Dubois, D. J, Trubiani, C., Casale, G. (2016). Model-driven Application Refactoring to Minimize Deployment Costs in Preemptible Cloud Resources. IEEE CLOUD 2016, San Francisco, CA, USA.

[24] W. Wang, G. Casale, A. Kattepur, M. Nambiar. Maximum Likelihood Estimation of Closed Queueing Network Demands from Queue Length Data, in Proc. of ACM/SPEC ICPE 2016.

[25] Weikun Wang, Juan F. Pérez, Giuliano Casale. Filling the gap: a tool to automate parameter estimation for software performance models. In Proc. of QUDOS@SIGSOFT FSE 2015, pages 31-32.

[26] Giuliano Casale, Simon Spinner, Weikun Wang. Automated Parameterization of Performance Models from Measurements. In Proc. of ICPE 2016, pages 325-326.

[27] Karsten Molka, Giuliano Casale. Experiments or simulation? A characterization of evaluation methods for in-memory databases. In Proc. of CNSM 2015, 201-209.

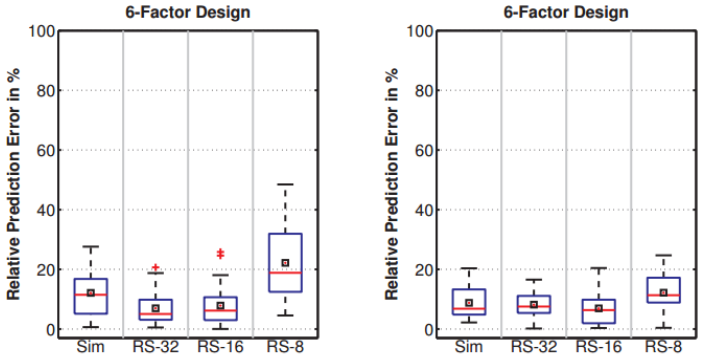
[28] Trubiani, C., Marco, A. D., Cortellessa, V., Mani, N., and Petriu, D. C. (2014). Exploring synergies between bottleneck analysis and performance antipatterns. In ICPE ’14, 75–86.

[29] SPEC. SPECjAppServer2002 Design Document https://www.spec.org/jAppServer2002/docs/DesignDocument.html

APPENDIX A. DICE-FG Validation Experiments

A.1 Validating memory usage estimation

The experimental results of the study are shown in Figure 12. Here **Sim** represents the memory consumption inference via stochastic models, where is simulated via JMT, and we denote the response surface models by **RS-8**, **RS-16**, **RS-32**, where the number *nn* in RS-*nn* indicates the experiments used for training. The analysis considers both mean memory consumption and peak memory consumption.

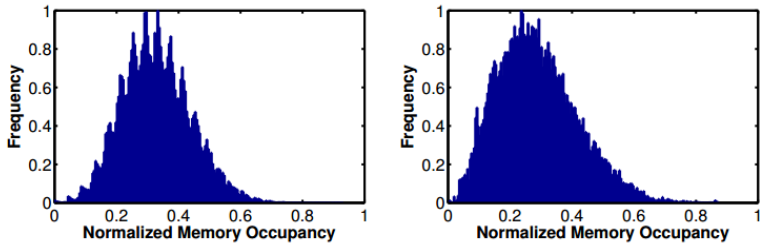


(a) Mean Memory Occupancy (b) Peak Memory Occupancy

Figure 12. Comparison of analytical and empirical memory models

The results indicate that all the approach can produce good results, with the accuracy of simulation being between 20% and 30%, and the accuracy of the response surface methods increasing as more data is fed into the estimation procedure. This suggests that both methods are viable in DICE.

To further verify the applicability of this memory inference approach, we have compared the distribution of memory usage in the real system compared to the one observed during the simulation experiments with JMT. The results are shown below in Figure 13, showing good agreement.



(a) Simulation (b) Traces

Figure 13. Validation of analytical model using simulation

Summarizing, our validation has revealed that memory consumption in DIA neither necessarily require advanced inference techniques to decouple the memory contribution of an individual request from the one of the others requests, nor the development of empirical response surfaces, which require extensive experimentation. Conversely, using simple tests in isolation from each individual requests we were able to obtain good predictive accuracy. As a result, the estimation of this parameter in UML models can proceed directly using the averaged memory consumption collected by DMon.

A.2 Validating applicability of mean execution time estimation in DIA

We have investigated the applicability of DICE-FG to the estimation of demands in an Apache Cassandra case study. In this case study, we have developed a simulation model for Apache Cassandra based on JMT. The model, shown in Figure 14, abstracts the behaviour of a Cassandra node part of a private cloud deployment.

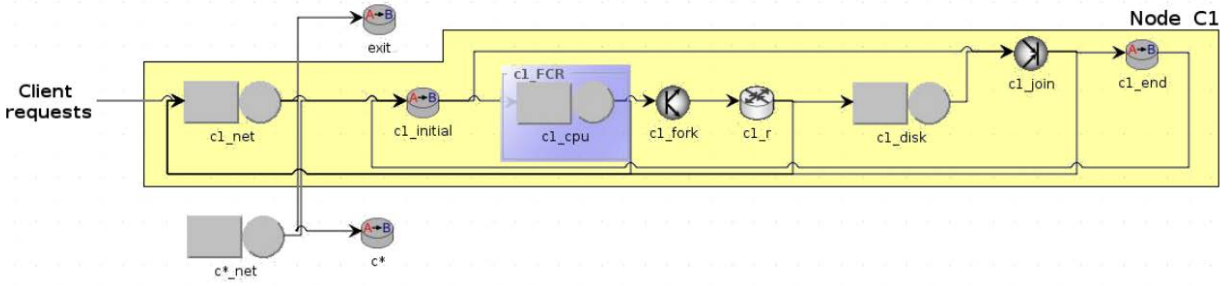


Figure 14. Analytical model of Apache Cassandra used to validate DICE-FG

The simulation considers aspects such as the CPU demand (*c1_cpu*), the disk demand (*c1_disk*), the network demand (*c*_net*), and the forking (*c1_fork*) and joining (*c1_join*) of jobs into tasks that are served either locally or by other nodes of the clusters, until conclusion (*exit*). The "*A->B*" elements denote a change of request type throughout its lifetime, an abstract needed for example to distinguish requests that are received and executed locally to a Cassandra node, from those that are received from remote Cassandra nodes.

One of the goals of this case studies was to apply DICE-FG to the estimation of demands in DIA of this kind, and confirm the quality of the estimated demands by mean of simulation-based prediction. Since the NETF case study relies on Cassandra, this work also goes in the direction of validating the DICE simulation models developed in WP3. To this end, we conducted experiments on a 4-node cluster. Each data item was replicated 3 times across random nodes, and we considered different consistency levels for the requests: ALL (all 3 nodes that hold the data need to retrieve the data), QUORUM (2 out of 3 need to reply), and ONE (a single node needs to reply). In each of these experiments, we varied the number of jobs that act as a client to the DIA, increasing their number up to reaching saturation (about 95% utilization).

DICE-FG has been applied on each node using the utilization-based regression (UBR) algorithms in order to obtain all required mean execution times needed to specify the model. Based on these estimated values, we have parameterized the simulation and estimated the performance trends as the number of jobs is increased.

The results, shown in Figure 15 below suggest that DICE-FG provides good input parameters to the simulation model, which result in fairly accurate estimates of mean execution times. Some deviations are seen above 30 jobs, but our investigation reveals that these are due to memory trashing effects that are not modelled and that normally do not arise in production systems, where the application is generally tuned to run at lighter loads than 90%.

It is quite interesting to note, in particular, that the parameterization is obtained on an experiment with 1 job and these values are then applied to the prediction of the system with more jobs. Therefore, similarly to what observed in the case of memory consumption estimation, running experiments at low concurrency levels and that stress features one-at-a-time appears to be sufficient to parameterize simulation models of DIA.

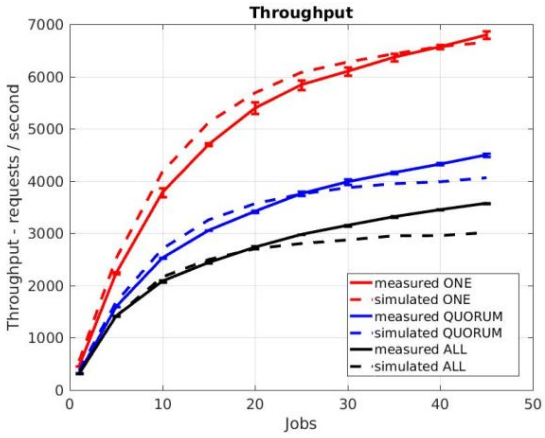


Figure 15. DICE-FG Validation Results on Apache Cassandra

A.3 Validating distribution analysis of execution times

In order to validate our implementation, we have attempted to generate PH distributions fitting the MapReduce data shown at the beginning of this section and subsequently we have used the resulting distributions in a simulation model of Hadoop/MapReduce to validate accuracy. The results of the fitting are illustrated below for the reducer execution times.

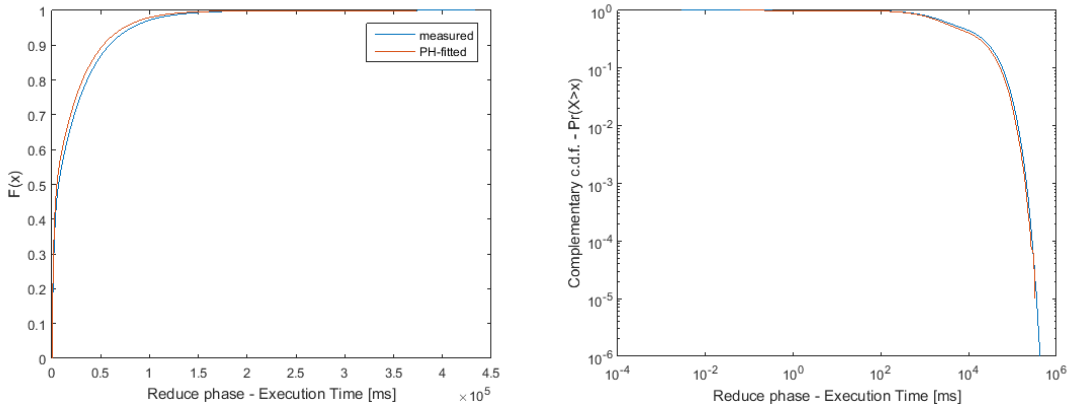


Figure 16. DICE-FG distribution fitting results on MapReduce execution time data

The diagrams indicate that the PH model delivers an accurate fitting of the empirical distribution. We also show the complementary c.d.f, which is here the probability that a reducer execution time exceeds the value shown in the x-axis. The fitting is very accurate, illustrating the capability of the algorithms integrated in DICE-FG to model long-tailed execution time distributions.

The above distributions have been used in the Hadoop/MapReduce case study reported in *D3.8 - DICE Optimization Tools* in Chapter 6. Earlier attempts based on exponential distribution fitting produced incorrect predictions, with large inaccuracies exceeding 60% error on response time. The introduction of phase-type distribution resulting in much more accurate predictions of the order of 15-30% accuracy errors.