Project acronym:	**PrEstoCloud**
Project full name:	**Proactive Cloud Resources Management at the Edge for efficient Real-Time Big Data Processing**
Grant agreement number:	**732339**

D5.5 Resources Adaptation & Data-intensive Application Recommenders – Iteration I

Deliverable Editor:	**Dimitris Apostolou (ICCS)**
Other contributors:	**Nikos Papageorgiou, Andreas Tsagkaropoulos, Yiannis Verginadis, Gregoris Mentzas (ICCS)**
Deliverable Reviewers:	**Salman Taherizadeh (JSI), Nenad Stojanovic (Nissatech)**
Deliverable due date:	**30/06/2018**
Submission date:	**30/06/2018**
Distribution level:	**Public**
Version:	**Final**

2018 © Copyright lies with the respective authors and their institutions.

This document is part of a research project funded by the Horizon 2020 Framework Programme of the European Union

Change Log

Version	Date	Amended by	Changes
0.1	05/04/2018	Dimitris Apostolou (ICCS)	Table of Contents
0.2	20/04/2018	Dimitris Apostolou, Nikos Papageorgiou, Andreas Tsagkaropoulos, Gregoris Mentzas, Yiannis Verginadis (ICCS)	First draft version circulated to the consortium
1.0	27/06/2018	Dimitris Apostolou, Nikos Papageorgiou, Andreas Tsagkaropoulos, Gregoris Mentzas, Yiannis Verginadis (ICCS)	Pre-final version ready for internal review
Final	30/06/2018	Dimitris Apostolou, Nikos Papageorgiou, Andreas Tsagkaropoulos, Gregoris Mentzas, Yiannis Verginadis (ICCS)	Final version

2018 © Copyright lies with the respective authors and their institutions.

Table of Contents

2018 © Copyright lies with the respective authors and their institutions.

List of Tables

List of Figures

2018 © Copyright lies with the respective authors and their institutions.

List of Abbreviations

The following table presents the acronyms used in the deliverable.

Abbreviation	*Description*
ACE	Acquisitional Context Engine
DIAFDRecom	Data Intensive Application Fragmentation & Deployment Recommender
AMQP	Advanced Message Queuing Protocol
CPU	Central Processing Unit
CSV	Comma Separated Value
DSL	Domain Specific Languages
HDA	Highly Distributed Applications
JPPF	Java Parallel Processing Framework
JSON	JavaScript Object Notation
MCA	Mobile Context Analyser
MCC	Mobile Cloud Computing
MEC	Mobile Edge Computing
OMG	Object Management Group
OS	Operating System
RAM	Random Access Memory
RARecom	Resources Adaptation Recommender
RPI	Raspberry Pi computer
TOSCA	Topology and Orchestration Specification for Cloud Applications
SDM	Situation Detection Mechanism
UAV	Unmanned Areal Vehicle
UML	Unified Modelling Language
VM	Virtual Machine
XMI	XML Metadata Interchange
XML	Extensible Markup Language
LTE	Long-Term Evolution
GPS	Global Positioning System

2018 © Copyright lies with the respective authors and their institutions.

Executive Summary

This deliverable reports on the work performed under Task 5.3 "Dynamic adaptation of resources allocation" and Task 5.4 "Data-intensive application fragmentation and deployment recommender" with respect to the development of two components included in the PrEstoCloud architecture: (i) Data-Intensive Application Fragmentation & Deployment Recommender (DIAFDRecom) and (ii) Resources Adaptation Recommender (RARecom). Both these components are central elements of the Meta management layer of the PrEstoCloud architecture, and provide the main input to elements of the Control layer, describing the properties of the Cloud Application deployment. The DIAFDRecom and RARecom are responsible for communicating the preferences and constraints of the DevOps to the Control layer and directing the adaptations of the processing topology – and therefore can be considered to be elements of the PrEstoCloud backbone.

The DIAFDRecom module provides the capability of parsing code-level annotations, as well as DevOps preferences and requirements (e.g. cloud provider requirements) expressed in a policy file. These requirements are then grouped and a preliminary ("type-level") TOSCA file is produced. This type-level TOSCA is subsequently pushed to the PrEstoCloud Repository, whence it is retrieved by the appropriate Control Layer components in order to calculate the optimal configuration for the initial application deployment. The RARecom exploits monitoring information with respect to the health status of the deployed application, and triggers the adjustment of the deployed topology properties, in order to better satisfy the requirements expressed by the developer (in the annotations) and the DevOps (in the policy file).

The DIAFDRecom and RARecom provide a solution for the DevOps and the developer to express their requirements in an easily-understandable format (key-value, and annotation-based), and yet be able to communicate over a modern cloud standard (OASIS 2017). They have been built in a way that permits extensions and enhancements (e.g. adding a new requirement or a new TOSCA file section / element). In order to facilitate the understanding of the function of these components, we have included appropriate architectural representations. The relevant figures explain both the internal architecture of the components, as well as their relationships with the rest PrEstoCloud components.

2018 © Copyright lies with the respective authors and their institutions.

1. Introduction

1.1 Scope

This deliverable reports on the baseline implementation of the Data-Intensive Application Fragmentation & Deployment Recommender and the Resources Adaptation Recommender. These two modules of the PrEstoCloud meta-management layer (shown in Figure 1) provide PrEstoCloud the capabilities to recommend application fragmentation, deployment and adaptation actions. Recommendations are propagated to the Control layer, which is responsible for implementing them. The main goal of these two modules is to ensure that the Cloud application will be reliably executed, under non-trivial workloads, while respecting the expressed requirements. The two modules rely on the input of the developer and the DevOps, who provide the guidelines for the initial deployment and subsequent adaptation actions.

The scope of this deliverable includes the architectural design and implementation of the two recommenders. Specifically, we present the internal structure of both the DIAFDRecom and RARecom modules, outlining the main functionalities of each one and the interactions between the constituent sub-components of each module. Moreover, the deliverable includes the specification of the input and the output of the two modules.

1.2 Relation to PrEstoCloud Tasks and Components

The Resource Adaptation Recommender and the Data-Intensive Application Fragmentation & Deployment Recommenders follow the definitions set as part of D2.1 (Scientific and Technological State-of-the-Art analysis), and adhere to the principal layout of the PrEstoCloud platform, as it is expressed in Deliverable 6.1 (Architecture of the PrEstoCloud platform) and initially described in D2.3 (Conceptual Architecture).

Figure 1 shows the logical relations between the two recommenders and other components of the meta-management layer as well as the control layer and the user roles. Technical links reflect the actual flow of information (e.g through network communication) during the operation of the Meta management layer, while logical links reflect the conceptual flow of information among entities.

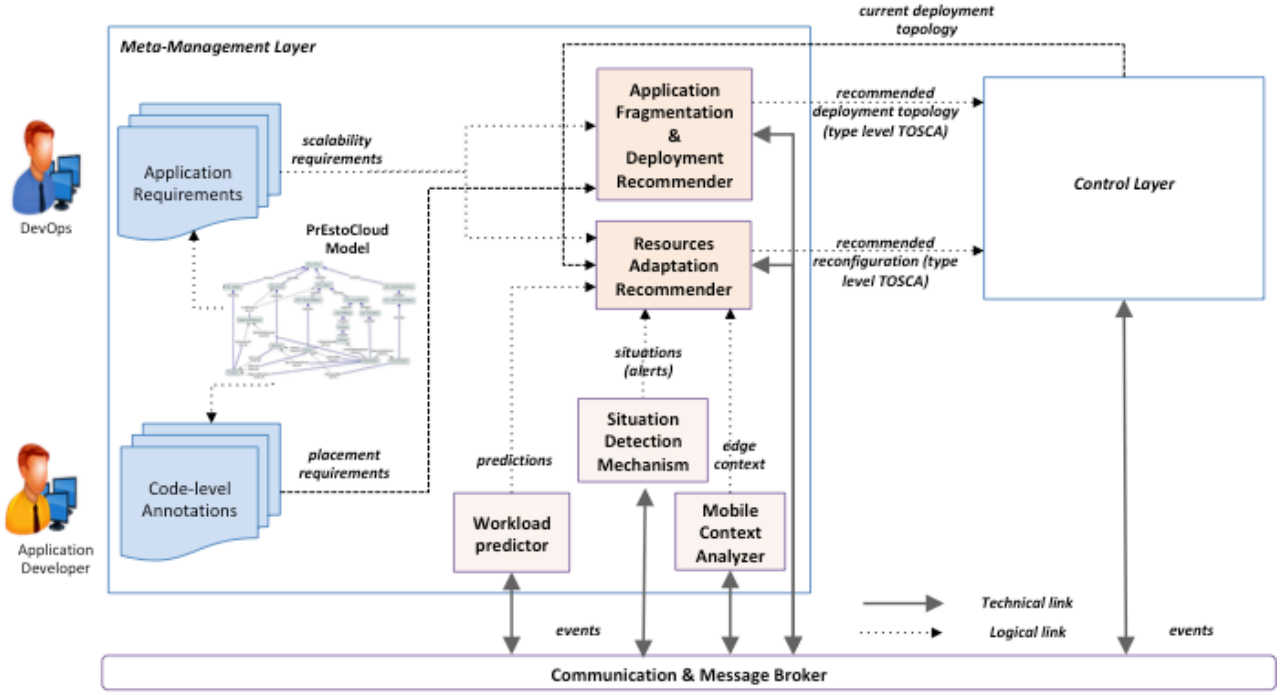

Figure 1: Relations of DIAFDRecom and RARecom with PrEstoCloud components

2018 © Copyright lies with the respective authors and their institutions.

For the DIAFDRecom module, the input is the code-level annotations and a policy file created by the Application Developer and the DevOps, respectively. The output of the DIAFDRecom module is a description of the appropriate topology (according to the expressed preferences and constraints) of the application fragments, expressed in the TOSCA language. The RARecom module receives input from every component of the Meta-management layer – the Mobile Context Analyzer (developed as part of D3.5), the Situation Detection Mechanism (developed as part of D5.1) and the Workload Predictor (D5.3). Moreover, it receives as input from the Control Layer the current deployment of the application. The output of the RARecom module is a description of suggested changes in the current topology of the application, including information about edge devices that cannot be used (according to their context) expressed in the TOSCA language. The output of both modules will be parsable by the components of the Control layer.

1.3 Document Structure

The deliverable is structured as follows: Sections 2 and 3 present the approach, design and implementation of DIAFDRecom and RARrecom, respectively. In Section 4 we formulate our conclusions on the approach and the development of the two modules.Note that related works have been explored as part of deliverable D2.1. Moreover, we include in an appendix, the specification of the type-level TOSCA file which is the output of the recommenders.

2018 © Copyright lies with the respective authors and their institutions.

2. Data-Intensive Application Fragmentation & Deployment Recommender

2.1 Approach and Architecture

In PrEstoCloud, we implement a series of components which allow a data-intensive cloud application consisting of independent processing fragments to take advantage of multiple processing entities, from small-factor edge devices such as Raspberry Pi's or UAV's, to large-flavoured resources in public and private clouds. In order to accomplish this, we rely on the input of the developer and of the DevOps in the form of code annotations and of a policy file.

DIAFDRecom aims to undertake the responsibility of describing the appropriate fragmentation of applications into smaller parts in order to be efficiently deployed over cloud / edge resources. Moreover, it aims to associate applications and application fragments with placement constraints and optimization preferences. The input of this mechanism should include the available processing resources as well as the qualitative, quantitative preferences of the DevOp and/or the Application developer. Based on this input, the recommended fragmentation will be serialized in a TOSCA specification that will refer to type-level VMs or Edge resources (as hosting nodes), while the specific number, location and type of the VM or edge device processing instances will be decided based on the advanced optimization mechanism of the PrEstoCloud Control Layer.

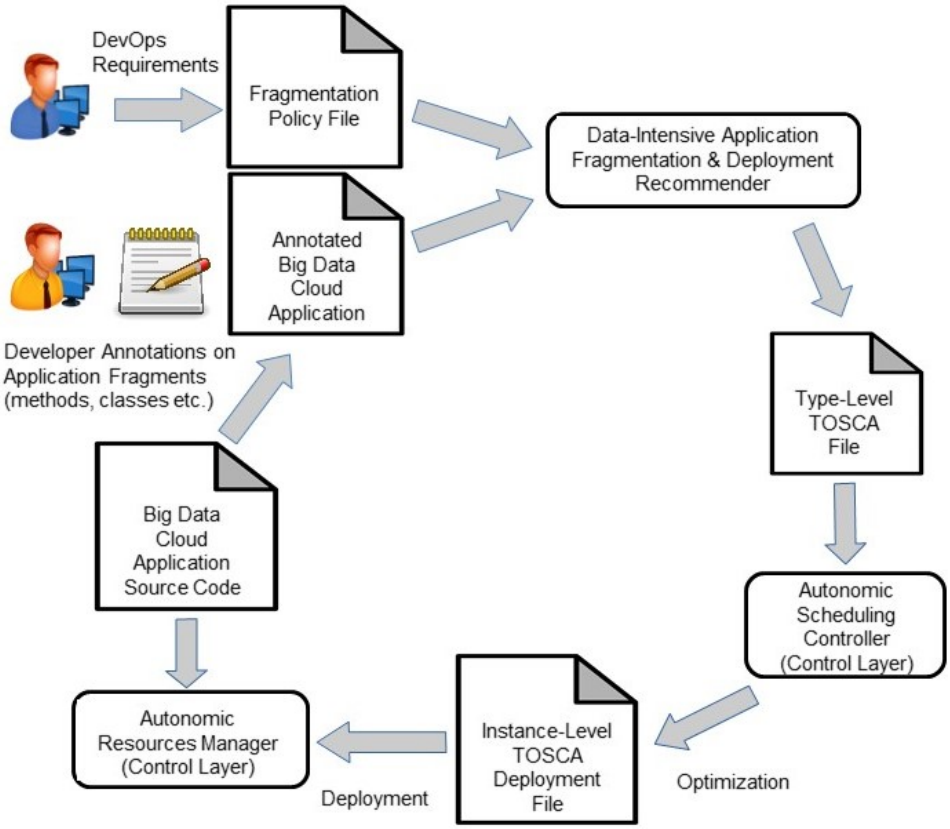

Figure 2: DIAFDRecom during the deployment cycle of an application in PrEstoCloud

In this section, we describe the current design of DIAFDRecom, which can handle code only in the Java programming language, which is widely popular and suitable for multiplatform execution. The input provided can enhance the behaviour of a service as processing requirements can be specified, as well as placement constraints. Additionally, the developer can fine-tune the processing environment of code-fragments through the use of optional java annotations. When no annotation can be found for a fragment, it is expected to be processed locally (legacy mode). The annotations of the developer are completed and

2018 © Copyright lies with the respective authors and their institutions.

can be further enhanced, with the requirements expressed by the DevOps in the Policy file (e.g specifying the Cloud provider to be used). DIAFDRecom processes the requirements included in these files, and produces output in TOSCA format, understandable by the components of the Control layer, which can process it and instantiate the processing topology in an automatic way. Figure 2, presents the DIAFDRecom in the lifecycle of an application deployed using the PrEstoCloud platform.

The DevOps and the Developer send to the module the requirements of the application which are processed and transcribed to a type-level TOSCA file. This file is then retrieved by elements of the Control Layer which optimize the deployment, ensuring the requirements of the application are met. The last step of the initial placement (or the reconfiguration) of an application is the deployment of the application.

The internal architecture of the processing carried out inside the DIAFDRecom is described in the workflow depicted in Figure 3:

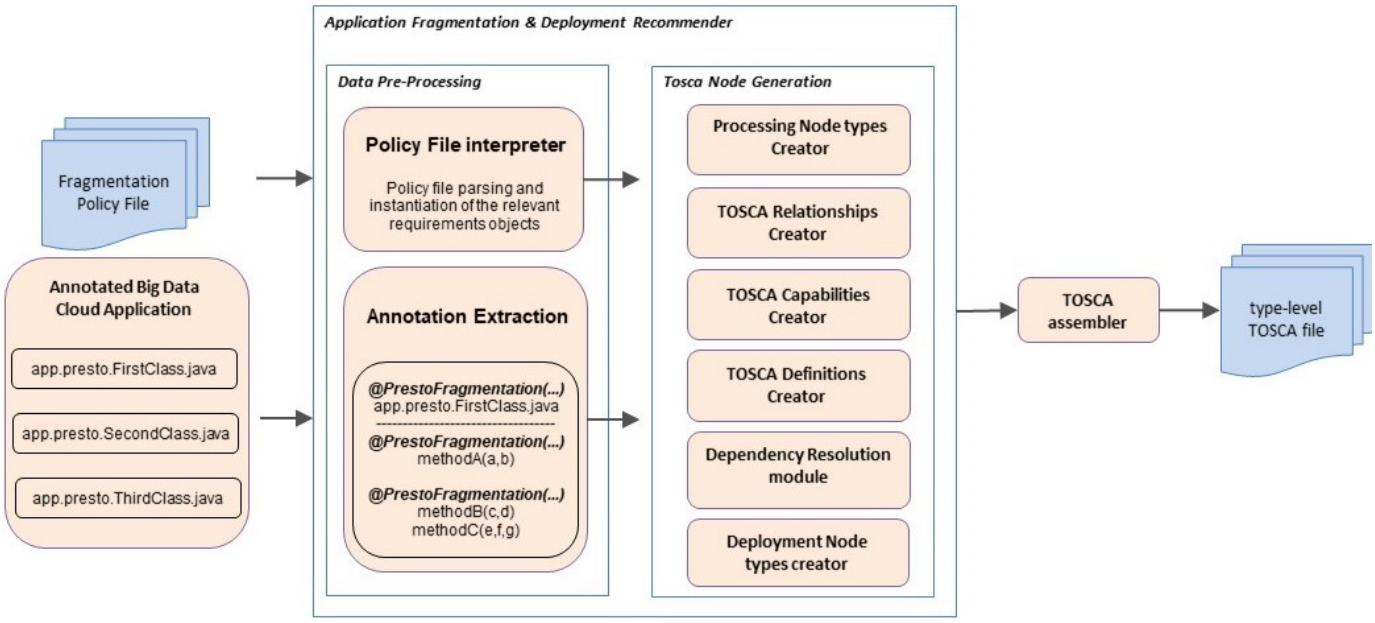

Figure 3: The internal architecture of DIAFDRecom

The necessary input for the application consists of the Fragmentation policy file which is compiled by the DevOps and contains the various requirements of the application from the platform, and the source-code annotations of the developer.

Once all input has been properly entered, the DIAFDRecom module can be triggered, starting the execution of the Tosca Node Generator class, which in turn invokes the Policy File interpreter and the Annotation Extraction modules. The Annotation Extraction module will extract the annotation from each method (expressing the hosting requirements), as well as any dependency requirements or anti-affinity constraints (collocation requirements). The Policy file interpreter on the other hand extracts Business Goals, Deployment Requirements, Budget Requirements, Scalability Requirements, Provider Requirements, and Mapping Requirements.

Following the completion of data pre-processing, the constituent elements of the TOSCA file – the TOSCA nodes – can be constructed. This procedure is performed in the TOSCA node generation stage, which defines new TOSCA nodes and maps the requirements gathered to the appropriate TOSCA constructs. More specifically, the mapping and deployment requirements expressed by the DevOps and the hosting and collocation requirements expressed by the developer are used to create new processing node types and collocation (or anti-affinity) policies. The business goals, budget requirements and provider requirements (are added as metadata, to be considered in the optimization carried out in the Control Layer. Afterwards,

2018 © Copyright lies with the respective authors and their institutions.

the definitions of new property, relationship and capability nodes containing core functionality for PrEstoCloud are appended to the TOSCA file.

Finally, new TOSCA nodes are created for each annotated application fragment which are used to describe the mapping of application fragments to processing nodes, and the TOSCA file is finalized. The output is a "type-level" TOSCA file, because it only contains information concerning the types of the processing nodes, and the relationships among them, rather than naming specific VM flavours, IP ranges or directly mentioning the cloud provider to be used. These latter actions are within the scope of the Application Placement & Scheduling Controller PrEstoCloud component which needs to solve a constraint programming problem in order to calculate an optimal instantiation of the topology used for hosting the data-intensive application.

2.2 Implementation

This section details the implemented system. We have built a component developed using Java 8, which introspects the application code, the java classes of an application (for example placed inside a package in the project directory), retrieves annotations and creates a type-level TOSCA file. The component includes a first, data pre-processing stage which lays the ground for the second, TOSCA generation stage, as described in the previous subsection.

The implementation of DIAFDRecom is available online at: https://gitlab.com/prestocloud-project/application-fragmentation-deployment-recommender

2.2.1 The Data Pre-Processing stage

In the PrEstoCloud semantic model definition, created as part of D2.5 (PrEstoCloud Semantic Model), the application requirements which can (or should be) defined by the DevOps or the developer were specified. In addition to these, we now define two new requirement types: Firstly, mapping requirements - which describemappings between the linguistic valuesused to characterize the load on each processing attribute and the corresponding processing capacity alloted to it (e.g CPULoad.LOW mapped to 2 CPU cores)- and secondly, Deployment Requirements which enable the DevOps to define global topology constraints.

In Figure 4, the PrEstoCloud Fragmentation Policy Model developed as part of D2.5(PrEstoCloud Semantic Model) is depicted. The areas which are greyed out represent model entities which are not currently used for the fragmentation of the application (but they will be supported in the second release of the component).

2018 © Copyright lies with the respective authors and their institutions.

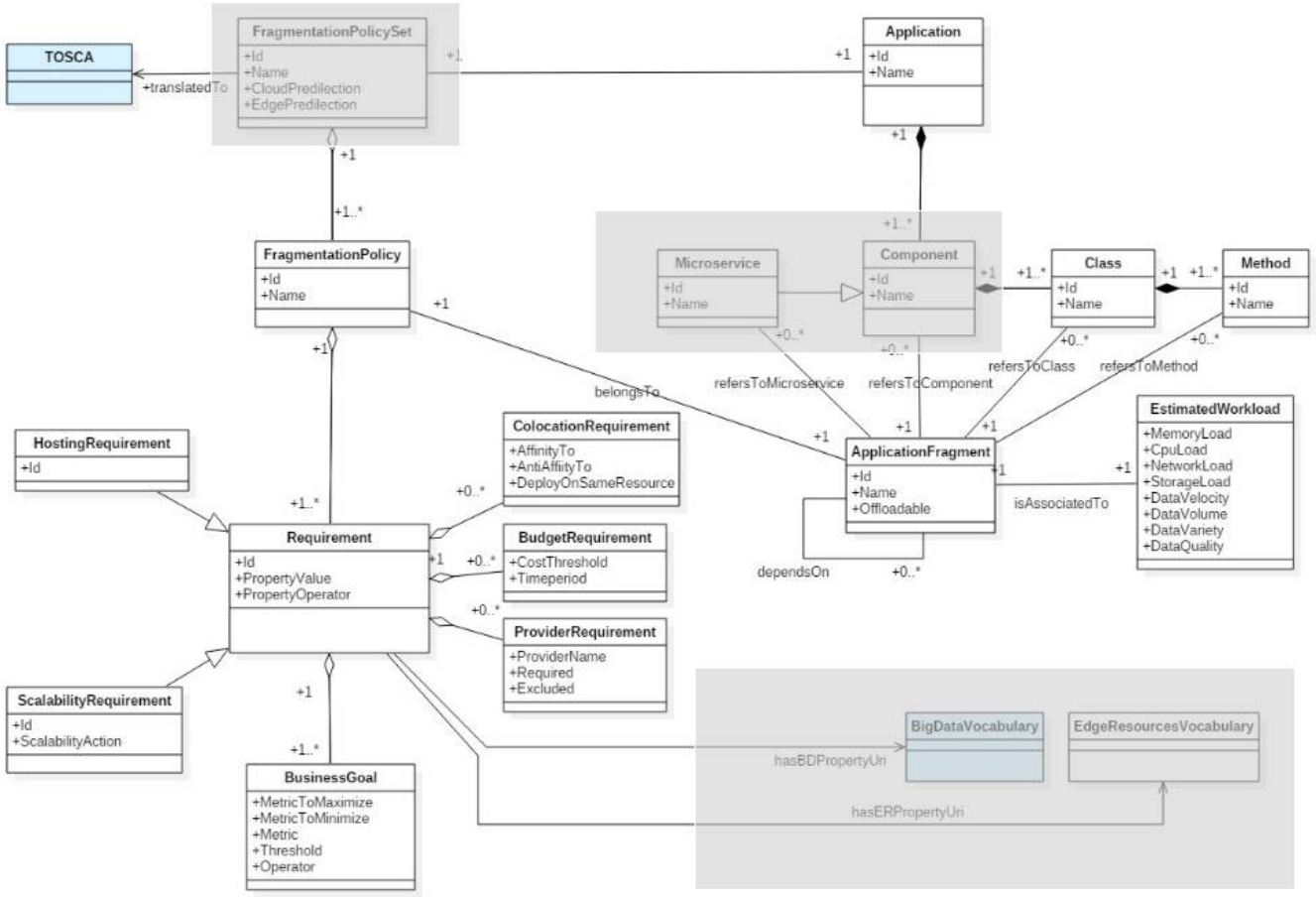

Figure 4: Elements of the Fragmentation Policy model used by the DIAFDRecom

The requirements mentioned above are conveyed through the two input sources of the DIAFDRecom: The annotated source code, and the policy file. The Collocation and the Hosting Requirements are expressed in the annotations, while all other requirements are placed in the PrEstoCloud application policy file. It is the duty of the Data Pre-Processing segment of the DIAFDRecom to parse this information and convert it into a format which can be used for further processing.

2.2.1.1 Annotations Extraction and processing

The DIAFDRecom makes extensive use of annotations, input by the developer inside the processing source code. The annotations specify the hosting requirements and the collocation requirements of the application, using linguistic variables and processing fragment names respectively. The entities which can be annotated are Java methods and Java classes.

The definition of the annotation class is the following:

```
package eu.prestocloud.annotations;

import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Retention(value = RetentionPolicy.RUNTIME)
public @interface PrestoFragmentation {
  enum MemoryLoad {
    VERY_LOW, LOW, MEDIUM, HIGH, VERY_HIGH
  }
```

2018 © Copyright lies with the respective authors and their institutions.

```
  enum CPULoad {
    VERY_LOW,LOW, MEDIUM, HIGH, VERY_HIGH
  }

  enum StorageLoad {
    VERY_LOW,LOW, MEDIUM, HIGH,VERY_HIGH
  }

  String policyFile();

  String overloading_tag() default "";

  boolean onloadable() default false;

  int max_instances() default 1;

  MemoryLoad memoryLoad();

  CPULoad cpuLoad();

  StorageLoad storageLoad();

  String[]  dependencyOn() default {};
  String[]  antiAffinityTo() default {};
}
```

Listing 1. Annotation Class definition

Annotating a Java method signifies that the processing fragment is an independent, explicitly parallel service. Furthermore, it also denotes that it is not affected by race conditions (occuring when parallel processing threads try to simultaneously modify shared data) , and does not require any additional explicit synchronization. Furthermore, annotating a Java class signifies that all of its methods are independent, parallel services in the sense described above. Additionally, when a method of an annotated class is not annotated, it inherits the class annotation.

A complete example of an annotation is the following:

```
@PrestoFragmentation(
policyFile       = "policy_file_name",
onloadable       = true,
memoryLoad       = MemoryLoad.HIGH,
overloading_tag  = "1",
cpuLoad          = CPULoad.HIGH,
storageLoad      = StorageLoad.LOW,
max_instances    = 3
dependencyOn     =
{"eu.prestocloud.application_classes.AudioAnalytics.fragmentA","eu.prestocloud.ap
plication_classes.AudioAnalytics.fragmentB"}
antiAffinityTo   = {"eu.prestocloud.application_classes.AudioAnalytics.fragmentC"}
)
```

Listing 2. A sample annotation

The hosting requirements are determined from the **max_instances** annotation, as well as the processing load of the particular code fragment which is described in terms of CPU, storage and memory usage and is expressed using five linguistic variables: ***VERY_LOW***, ***LOW***, ***MEDIUM***, ***HIGH***, ***VERY_HIGH***. These variables are defined in a suitable enumeration type for each resource, and are used by the **cpuLoad**, **memoryLoad** and **storageLoad** annotations. These variables can be set by the developer based on the estimated workload requirements of the code that he has produced. The collocation requirements are

2018 © Copyright lies with the respective authors and their institutions.

added using the **dependencyOn** annotation. The available annotations, and possible values are presented below. In the example given above, the hosting requirements dictate that a high number of CPU cores and memory GB's should be allotted to the node which will process the particular fragment, while the storage requirements are modest. The fragment can scale out to 3 instances (at most) although it should respect the constraints of collocation and anti-collocation for fragmentA, fragmentB and fragmentC respectively.

The information contained in each annotation, as per the definition provided above is shown in Table 1:

Table 1. The available annotations for a code-level fragment

Annotation Name	Description	Possible values	Notes
Policyfile	The name of the policy file which will be used	"basic_policy_file.policyfile"	Mandatory
overloading_tag	A tag which can be used to distinguish between different instances of overloaded java methods and constructors	[0,Integer.MAX_VALUE]	Optional, however in case of overloaded methods / class constructors there will be no opportunity to distinguish between different instances
Onloadable	A boolean value which determines if the application fragment can be placed for execution on the edge	{ true,false }	Optional, assumed to be false if missing
memoryLoad	An indication for the memory load of the application fragment	{ MemoryLoad.VERY_LOW, MemoryLoad.LOW, MemoryLoad.MEDIUM, MemoryLoad.HIGH, MemoryLoad.VERY_HIGH }	Mandatory
cpuLoad	An indication for the CPU load of the application fragment	{ CPULoad.VERY_LOW, CPULoad.LOW, CPULoad.MEDIUM, CPULoad.HIGH, CPULoad.VERY_HIGH }	Mandatory
storageLoad	An indication for the storage load of the application fragment	{ StorageLoad.VERY_LOW, StorageLoad.LOW, StorageLoad.MEDIUM, StorageLoad.HIGH,Storage.VERY_HIGH }	Mandatory
max_instances	The maximum number of processing fragment instances which can be concurrently used	[1,Integer.MAX_VALUE]	Optional, assumed to be Integer.MAX_VALUE if missing.
dependencyOn	A list of method-level fragments which should be collocated with the fragment which is currently annotated	{ "eu.prestocloud.tosca_generator.AudioAnalytics.detectShout", "eu.prestocloud.tosca_generator.AudioAnalytics.runAlgorithm" }	Optional, assumed to be an empty list if field is not present

2018 © Copyright lies with the respective authors and their institutions.

antiAffinityTo	A list of method-level fragments which should not be collocated with the fragment which is currently annotated	"eu.prestocloud.tosca_generator.AudioAnalytics.detectSilence", "eu.prestocloud.tosca_generator.AudioAnalytics.testMethod2"	Optional, assumed to be an empty list if field is not present

Classes can also be annotated with almost all available annotations (exceptions are the dependencyOn and overteloading_tag annotations) to denote the annotation which should be applied to each of its methods which have not been annotated, implying that all methods in the class satisfy the constraints outlined above. In case a method is annotated inside a class, the class annotation is overriden by the annotation of the method.

2.2.1.2 The policy file

The policy file is placed in the "policyfiles" folder inside the DIAFDRecom project.

The policy file follows a simple JSON-like key-value approach to define application requirements. A sample policy file is provided below:

```
BusinessGoal:
  MetricToMinimize: Cost
BudgetRequirement:
  CostThreshold: 1000
  TimePeriod: 720
DeploymentRequirement:
  MaxInstances: 100
  MaxFragmentInstances: 10
  MaxMasterNodeInstances: 2
ScalabilityRequirement:
  ScalabilityRequirement1: https://example.com/scalabilityAction
  ScalabilityRequirement2: https://example.com/scalabilityAction2
MappingRequirement:
  CPU:
    VERY_HIGH: 32
    HIGH: 16
    MEDIUM: 8
    LOW: 4
    VERY_LOW: 2
  RAM:
    VERY_HIGH: 32
    HIGH: 16
    MEDIUM: 8
    LOW: 4
    VERY_LOW: 2
  DISK:
    VERY_HIGH: 4000
    HIGH: 2000
    MEDIUM: 500
    LOW: 100
    VERY_LOW: 40
ProviderRequirement:
  ProviderName: Amazon
  Required: false
  Excluded: true
```

Listing 3. A sample policy file

2018 © Copyright lies with the respective authors and their institutions.

The mapping requirements introduced in the policy file, bridge the gap between the definition of linguistic variables in the annotations, and the actual processing characteristics expected for a deployment. Each of the five linguistic variables is mapped with the help of the MappingRequirements to a concrete integer value. These integers designate the number of cores which should be used (in CPUload annotations) and the gigabytes of free space required to process the fragment (both in memoryload and storageload annotations).

Additionally, the Deployment Requirements are refined to allow the DevOps to control the behaviour of the application in a more fine-grained way. The rest of the requirements follow the semantics described as part of the PrEstoCloud Semantic Model in D2.5 (PrEstoCloud Semantic Model). While most of the requirements are not implementation-bound, one of the deployment requirements (MaxMasterNodeInstances) is tied to the implementation decision of using JPPF. The particular requirement creates a constraint on the maximum number of JPPF master nodes which coordinate the processing carried out in JPPF agent nodes.

A description of the elements of a policy file follows in Table 2. The elements of a policy file do not need to appear in a particular order.

Table 2. The policy file requirements

Requirement Name	Description	Possible attributes	Notes
BusinessGoal	A Business Goal which the application should try to attain	MetricToMinimize or MetricToMaximize or a (Metric, Operator, Threshold) tuple, which are all String variables.	Mandatory, multiple BusinessGoals permitted
BudgetRequirement	The Budget requirement of the application	CostThreshold and TimePeriod, specifying the number of Euros available for the use of public clouds, and the number of hours for which this amount should last.	Optional, only one BudgetRequirement permitted
MappingRequirement	The mappings between the linguistic processing variables and the actual resources required	For each of the attributes (CPU,RAM,Disk), the desired mappings between the five linguistic variables (VERY_LOW, LOW, MEDIUM, HIGH, VERY HIGH) and their concrete integer values.	Mandatory but also only one MappingRequirement is permitted
ProviderRequirement	A Provider requirement stating the providers that can / should not be used for the deployment of the application	Tuples in the format of (ProviderName,Required), or (ProviderName,Excluded), or (ProviderName,Required,Excluded) . The ProviderName is a String, while the Required and Excluded parameters are boolean variables.	Optional, multiple ProviderRequirements are permitted
ScalabilityRequirements	A List of scalability requirements stating the name of the requirement and a uri of the respective rule	Tuples in the format of (ScalabilityPolicyName: ScalabilityPolicyUri).	Optional, only one ScalabilityRequirement is permitted

2018 © Copyright lies with the respective authors and their institutions.

DeploymentRequirement	A Deployment requirement stating scaling parameters which should guide the deployment and reconfiguration of the application	MaxInstances, or MaxFragmentInstances, or MaxMasterNodeInstances. All of these fields accept integer values and denote the maximum number of processing instances globally, the maximum instances which can be started per fragment (overrides the value in the annotation if less), and the maximum instances of JPPF Master nodes in the topology	Optional, only one DeploymentRequirement permitted

2.2.2 The TOSCA generation stage

A detailed description of the function of each component in the TOSCA node generation of the DIAFDRecom follows.

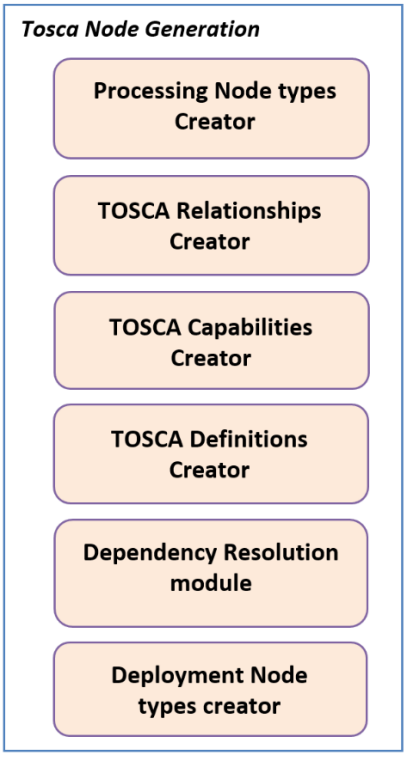

Figure 5 The Tosca Node Generation subcomponents

2.2.2.1 Processing Node Types Creator

The processing node types creator component creates a TOSCA node for each application fragment. These nodes contain the hosting requirements which should be met for an application fragment to be able to be executed. Initially, for each annotation describing the resource workload of a fragment (the CPU load or the memory load or the storage load), linguistic variables are mapped to a range of integers (reflecting the number of cores and GB's), using the mapping requirement contained in the policy file. For example, if the mapping requirement indicates that HIGH CPU usage requires 16 processing cores and that VERY_HIGH CPU usage (the linguistic value after HIGH) requires 32 processing cores, the resulting range for a fragment annotated with the CPULoad.HIGH annotation will be [16, 32] cores. Here, [16, 32] means that the

2018 © Copyright lies with the respective authors and their institutions.

number of cores can be from 16 to 32, inclusive. Providing a range instead of a single value allows for flexibility and optimization of the deployment in the Control Layer. In general, we consider that if two linguistic variables **z1** and **z2** of successively increasing magnitude (for example VERY_LOW and LOW) are mapped to integer values **x** and **y** respectively, and that a fragment has been annotated with a processing load of **z1** for a resource type, the respective integer range will be [**x** , **y**].

The processing characteristics of the host node are then complemented by more generic properties pertaining to the OS (Operating System) and the architecture of the processing device. Then the requirements of the host node in terms its location (cloud or edge) are set to cloud and edge or cloud only, according to the onloadable annotation of the fragment. Finally, in the case that the produced TOSCA file is a reconfiguration of the topology, a list of excluded edge devices can be attached to deny execution on devices which have low battery, do not have a quality data connection etc.

2.2.2.2 TOSCA Definitions Creator

This component consists of specific classes which model the TOSCA structure of a JPPF fragment, a JPPF Master and a JPPF Agent node. The JPPF Agent TOSCA representation includes the fundamental requirement of a JPPF Master to which it can connect, while the JPPF Master TOSCA representation specifies the capability of being connected to. The JPPF fragment representation includes the requirement of being processed by a JPPF Agent node.

2.2.2.3 TOSCA Relationships Creator & TOSCA Capabilities Creator

These two components consist of utility methods which can be used to express a new Relationship (i.e a dependency between TOSCA nodes) or a Capability (a specific trait of a node, e.g to execute a processing fragment) in the TOSCA format. They are used to create all capabilities and relationships which are not covered by standard TOSCA, yet are required by PrEstoCloud. Currently, there are two new capabilities (prestocloud.capabilities.jppf.endpoint and prestocloud.capabilities.jppf.fragmentExecution) and two new relationships (prestocloud.relationships.jppf.ConnectsTo and prestocloud.relationships.ExecutedBy) defined once and used throughout the type-level TOSCA document. The two capabilities are used by the TOSCA Definitions Creator, while the relationships are used both by the TOSCA Definitions Creator and the Processing Node Types Creator.

2.2.2.4 Dependency Resolution Module

The Dependency Processing module is a separate class inside the DIAFDRecom, which can resolve dependencies amongst fragments. For the purpose of modelling the dependency problem we consider that the cloud application can be represented by a graph G(V,E), where the set of vertices V represents the application fragments, and the set of edges E represents the relationship amongst the fragments (dependency or anti-affinity constraint).

a) Forward dependency processing

The Forward dependency processing is the simpler of the two algorithms which can be used to infer dependencies for a processing fragment (the other one being the Bidirectional dependency processing algorithm described below). Firstly, the annotation which describes other fragments on which the fragment is dependent on is extracted. Then, a new entry is created in a Java Map data structure, keyed by the name of the fragment and having a set of dependent fragments as its value.

Recursively, each of the fragments *f* in the fragment dependency set of a fragment *F* is evaluated for any dependencies it may have itself. If any dependencies are found, these are marked and added to the dependency set of *F* while *F* is marked as "explored". Once this BFS (Breadth-First-Search) – like traversal of graph G is finished, every fragment in the Map is associated to a set of dependencies possibly expanded in comparison to the original. The result of this process is that each fragment possesses its dependency group, and some fragments have been marked as dependencies of other fragments.

2018 © Copyright lies with the respective authors and their institutions.

In the final stage, the dependency group of every fragment not marked to be a dependency of another fragment, is converted to a TOSCA collocation group. These groups are subsequently added to the type-level TOSCA file.

```
function forward_dependency_processing():
 initialize_dependencies()


 foreach processing_fragment in annotated_source_code_classes:
  if (not(processing_fragment.isVisited)):
   processing_fragment.isVisited ← true
   additional_dependencies_found ← additional_dependencies(processing_fragment.getDependencies())
   processing_fragment.getDependencies().add(additional_dependencies_found)
   dependency_dictionary.add_pair (processing_fragment, processing_fragment.getDependencies()
   processing_fragment.isExplored ← true
 return

function additional_dependencies(dependent_fragments_list):

 additional_dependencies_found← EMPTY

 foreach processing_fragment in dependent_fragments_list:
  if (not(processing_fragment.isVisited)):
   processing_fragment.isVisited ← true
   processing_fragment.isIncluded ← true //marked as included in the dependencies of another node
   additional_dependencies_found.add(processing_fragment.getDependencies())
   additional_dependencies_found.add( additional_dependencies(processing_fragment.getDependencies()))
   fragment_dependencies.add(additional_dependencies_found)

   processing_fragment.isExplored ← true
  else if (processing_fragment.isExplored):
   processing_fragment.isIncluded ← true //marked as included in the dependencies of another node
   additional_dependencies_found.add(processing_fragment.getDependencies()) //add the dependencies of the node nevertheless

 return additional_dependencies_found

function initialize_dependencies():
 global dependency_dictionary ← EMPTY

 foreach processing_fragment in annotated_source_code_classes:
  processing_fragment.isVisited ← false
  processing_fragment.isIncluded ← false
  fragment_dependencies ← parse_annotation_for_dependencies()
  dependency_dictionary.add_pair (processing_fragment, fragment_dependencies)

return
```

Listing 4. The forward dependency algorithm pseudocode

For example let us consider an example application composed of fragments A,B,C… M, with the following dependencies:

Table 3. Example fragmented application

Fragment name	Fragments dependent on
A	B, E
B	-
C	E, D

2018 © Copyright lies with the respective authors and their institutions.

D	-
E	-
F	G, H
G	-
H	-
I	J
J	K
K	-
L	C, M
M	-

Considering this application as input to the forward dependency algorithm, three dependency groups will be created, as presented in Figure 6 below:

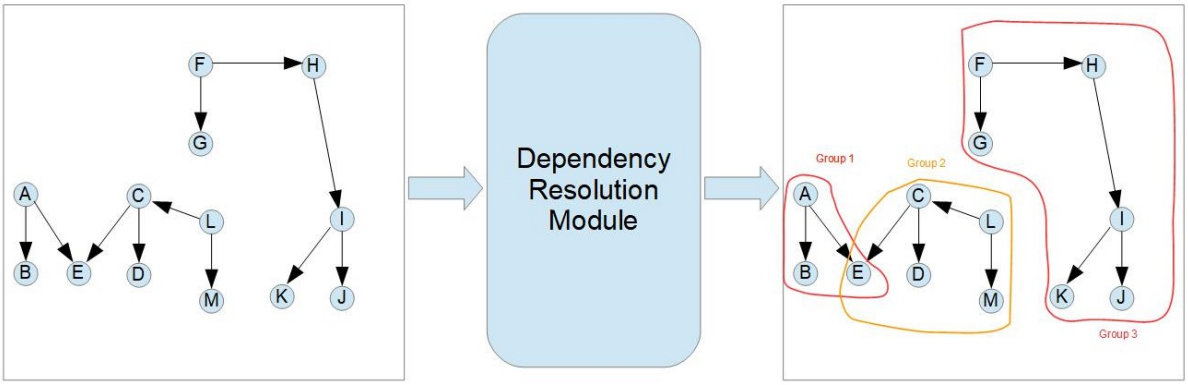

Figure 6: Resolution of dependencies using forward dependency processing

b) Bidirectional dependency processing

A shortcoming of the approach described above is that when two fragments *F1,F2* request another fragment *F3* as a dependency, two separate groups are created; one will contain {F1,F3}, while the other will contain {F2,F3}. Obviously, this means that all three fragments should be collocated, but the component optimizing the deployment (the Autonomic Placement & Scheduling Controller, developed in WP4) should include explicit business logic to handle this case. Alternatively, the developer should annotate only one of *F1* and *F2* with a dependency on *F2* or *F1* respectively, and F3. However, in order to alleviate the developer from having to handle a multitude of method dependencies, we include processing logic that can determine that {F1,F2,F3} should all be collocated.

For this reason, for every edge $e \in E$, describing a connection (V1→V2) we also add the reverse edge (V1←V2), and then apply the forward dependency algorithm described above.

```
function bidirectional_dependency_processing():
  initialize_dependencies()
  assign_backward_dependencies()

  foreach processing_fragment in annotated_source_code_classes:
    if (not(fragment.isVisited)):
      processing_fragment.isVisited ← true
```

2018 © Copyright lies with the respective authors and their institutions.

```
    additional_dependencies_found ← additional_dependencies(fragment.getDependencies())
    processing_fragment.getDependencies().add(additional_dependencies_found)
    dependency_dictionary.add_pair (processing_fragment, processing_fragment.getDependencies())
    processing_fragment.isExplored ← true


function additional_dependencies(dependent_fragments_list):

additional_dependencies_found← EMPTY

  foreach processing_fragment in dependent_fragments_list:
   if (not(processing_fragment.isVisited)):
    processing_fragment.isVisited ← true
    processing_fragment.isIncluded ← true //marked as included in the dependencies of another node
    additional_dependencies_found.add(processing_fragment.getDependencies())
    additional_dependencies_found.add( additional_dependencies(processing_fragment.getDependencies()))
    fragment_dependencies.add(additional_dependencies_found)

    processing_fragment.isExplored ← true
   else if (processing_fragment.isExplored):
    processing_fragment.isIncluded ← true //marked as included in the dependencies of another node
    additional_dependencies_found.add(processing_fragment.getDependencies()) //add the dependencies of the node nevertheless
return additional_dependencies_found


function assign_backward_dependencies():
  foreach processing_fragment in dependency_dictionary.entries:
   foreach dependency in processing_fragment.getDependencies()
    dependency_dictionary.add_pair(dependency,processing_fragment) //the inverse of the already existing (processing_fragment,
dependency) relationship.


function initialize_dependencies():
  global dependency_dictionary ← EMPTY

  foreach processing_fragment in annotated_source_code_classes:
   processing_fragment.isVisited ← false
   processing_fragment.isIncluded ← false
   fragment_dependencies ← parse_annotation_for_dependencies()
   dependency_dictionary.add_pair (processing_fragment, fragment_dependencies)
```

Listing 5. Bidirectional Dependency processing algorithm pseudocode

A graphical representation of the result of this dependency processing is Figure 7:

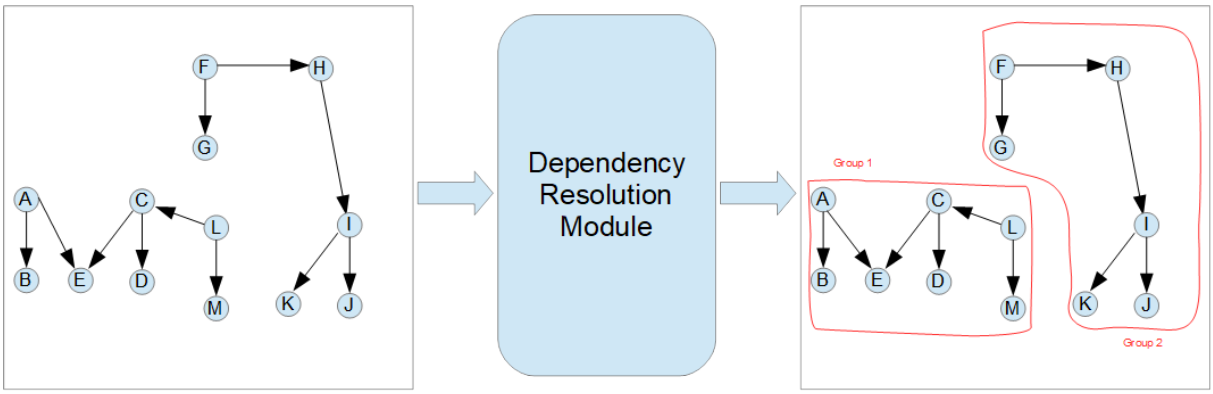

Figure 7: Resolution of dependencies using bidirectional dependency processing

By default, bidirectional dependency processing is enabled in order to address the shortcomings of the forward dependency processing. However if more post-processing flexibility is required (e.g. a module considering different dependency categories), the forward dependency mode is also available. The final

2018 © Copyright lies with the respective authors and their institutions.

output of these two dependency processing algorithms is a TOSCA collocation policy which includes the names of the fragments which should be collocated.

The anti-affinity statements are processed in a slightly different way: First, the list *L* which contains fragments which should not be collocated with a fragment *F* are determined from the annotations. Then for each fragment $f \in L$ the dependency set is determined using one of the above-described algorithms. The final output of this processing is a TOSCA anti-affinity policy which includes a list, containing as its first element the name of the fragment which should not be collocated with the fragments which follow in the list. Once the dependencies are finalized, the relevant collocation and anti-affinity TOSCA groups are created. The entry point for the collocation group are the nodes which have not been marked as dependencies of other nodes.

2.2.3 The TOSCA assembler

In the final stage of the deployment, the main sections of the file have already been created and are assembled in a full type-level TOSCA document. Additionally, information on TOSCA dependencies is added to the type-level document in order to enable the correct parsing of the document. The completed document is now ready to be stored to the PrEstoCloud Repository, and a new event is sent to the PrEstoCloud Communication & Messages Broker (D6.1) so that the document can be further processed in the PrEstoCloud lifecycle.

2018 © Copyright lies with the respective authors and their institutions.

3. Resources Adaptation Recommender

3.1 Approach

The Resource Adaptation Recommender (RARecom) is the software tool that aims to reason about adaptation actions within the PrestoCloud infrastructure. Today many distributed systems utilize cloud and edge resources in order to achieve goals such as cost minimization, high availability, lower response time, lower energy consumption or combinations of multiple, often competing, goals. In order to achieve those goals it is important to implement mechanisms that observe the current state of the system, the utilization of the different resources and the context in order to take decisions for adaptation at the right time.

In deliverable D2.1 (Scientific and Technological State-of-the-Art Analysis), chapter 3, we investigated the main resource adaptation and reconfiguration types that have been used in the related bibliography (horizontal and vertical scaling, VM and container migration, offloading and onloading). The RARecom will be designed to implement adaptation in environments that combine cloud and edge resources. It is an open issue whether the existing research and industrial approaches for cloud resource adaption can be applied in cloud-edge environments. According to a whitepaper published by OpenStack[1], cloud computing has several similarities but also important differences with edge computing. For example although both cloud and edge environments can benefit from the use of virtualized resources (CPU, memory, storage) in edge computing we have the capability to use resources closer to the end-user when the required network quality characteristics cannot be met due to technical or financial limitations. On the other hand in edge devices we have more restrictions in resource or energy consumption. So, in order to achieve specific goals in edge and cloud environments that PrEstoCloud aims to support, the resource adaptation recommender must be designed appropriately.

As described next, the first version of the RARecom is a rule-based and context-driven, flexible adaptation mechanism that will serve as the basis on top of which we will later develop more advanced adaptation strategies and functionalities by utilizing additional input (i.e. predictions from the Workload Predictor) and feedback capabilities in order to continuously improve the adaptation triggering.

We have identified the following top-level requirements for the RARecom:

1) Distributed architecture, compatible with the overall PrEstoCloud architecture. PrEstoCloud is an event-driven software platform. The distributed Message Broker (implemented using RabbitMQ) serves as the messaging infrastructure of PrEstoCloud which interconnects the different components in an agile and loosely-coupled manner. The RARecom should communicate with the other components of PrEstoCloud through the Message Broker .
2) Easy deployment in edge and cloud resources. As PrEstoCloud aims to support application and microservice deployment in non-uniform environments that consist of edge and cloud resources with very different software and hardware configurations and capabilities, the RARecom should be developed with technologies (such as multiplatform programming languages and libraries, virtualization and containerization technologies) that can support multiple execution platforms (processor architectures and operating systems).
3) Flexibility and reconfiguration capabilities. The RARecom should be able to follow the evolution of the environment that it supports, as edge devices join or leave constantly and cloud resources are changing following the trends of the workload, by adapting itself.
4) Easy integration with other systems, utilization of technologies adopted in PrEstoCloud. Several PrEStoCloud components such as the MCA and the SDM use the AMQP or MQTT protocols for messaging, JSON format for the encoding of event payload and REST interfaces. The RARecom

[1] https://www.openstack.org/assets/edge/OpenStack-EdgeWhitepaper-v3-online.pdf

2018 © Copyright lies with the respective authors and their institutions.

should follow common standards for the interconnections with the other components of PrEstoCloud.

In terms of WP5 work, we will develop and release the RARecom mechanism in two iterations. This document describes an initial design of both iterations along with the implementation details of the first one.

3.2 Architecture

The RARecom (Figure 8) aims to engage the run-time operation of our platform since it provides context-aware, edge and cloud adaptation recommendations that may include changes to the already used resources and reconfigurations with respect to where each application fragment has been hosted.

The RARecom receives as input the current processing topology and placement (i.e. resources used and hosting location of each application fragment), the detected situations, predicted workloads along with the respective context of the used and the available edge devices. Based on this, it generates as output the recommendation to reconfigure the processing topology, e.g., to introduce new processing nodes, replicate nodes for failover purposes, remove redundant or underused processing nodes and move application fragments among the available cloud and edge hosting nodes. For example, assume multiple streams coming from a variety of different cameras (i.e. CCTVs, mobile phones). Based on the data volume and velocity of these streams, the RARecom can recommend adaptations that will affect the processing topology. Such adaptations may involve moving away or closer to the edge certain application fragments (e.g. video transcoding, face detection) and/or instructing the use of additional instances of the same application fragments on different virtual hosts. We note that the RARecom is responsible for detecting the appropriate time for triggering a new reconfiguration while it dictates the minimum reconfiguration actions (e.g. add 2 more instances of a certain resource) and indicates which edge devices should be excluded in the new application topology (based on their context). As in the initial deployment flow, the final decision about what it will actually be deployed/reconfigured (i.e. expressed in an instance-level TOSCA) is made by the Application Placement & Scheduling Controller who is responsible for solving a constraint programming problem that expresses the optimization goals set by the DevOps. In any case the Application Placement & Scheduling Controller should respect the minimum reconfiguration actions set by the RARecom and extend them if it is necessary, according to the optimization goals (e.g. add 3 instances of the resource that the RARecom indicated).

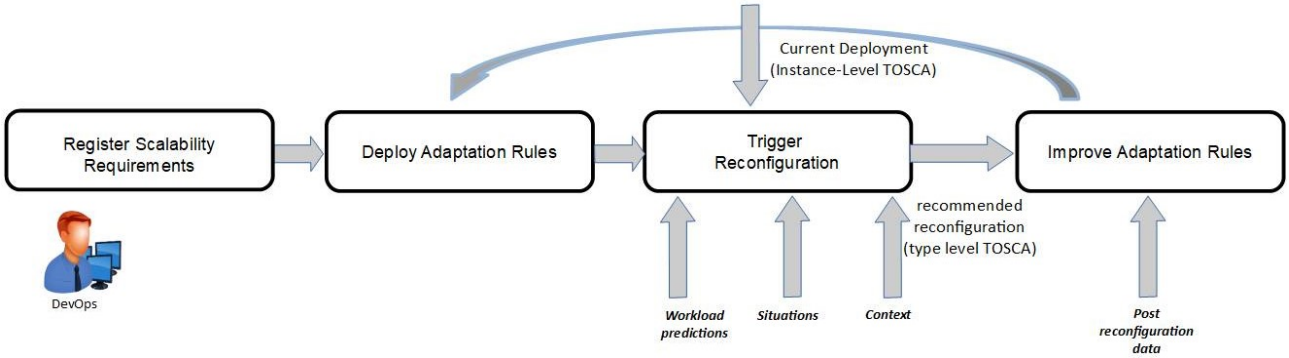

Figure 8. The RARecom Business Logic

To facilitate the operation of the RARecom, the DevOps need to register the scalability requirements of the deployed applications and services. Based on these requirements and constraints, adaptation rules are created and deployed. Adaptation rules are at the core of our approach. They enable the triggering of reconfiguration recommendations based on the knowledge about the current deployment topology as well as the predicted workloads and context of computing resources. Adaptation actions are triggered by detected situations. The actual adaptation result is determined by a series of "meta-rules" which are created based on the scalability requirements expressed by the DevOps (as explained below). Meta-rules

2018 © Copyright lies with the respective authors and their institutions.

may specify for example the number of instances which are actually used to implement the adaptation prescribed in a scalability rule. A further feature of the proposed approach is the capability to improve the adaptation meta-rules based on analyses of post-reconfiguration data, that is, data that shows how well a recommended and eventually implemented reconfiguration has proved in reality.

The architecture of the RARecom is shown in Figure 9. At the core of the RARecom lays a rule engine (Drools 7.7.0), comprising the production and working memories as well as an inference engine. The inference engine is responsible for matching active rules to triggering conditions (situations and context) and for managing multiple, active rules. Production memory is used to maintain and configure adaptation rules while the working memory contains the facts that will be used by the inference engine when firing the rules. Asynchronous communication with other Meta-Management Layer components (Mobile Context Analyser, Workload Predictor and Situation Detection Mechanisms) is done through the broker and its pub-sub mechanism. The Control Layer feeds the RARecom with the current deployment of the applications and services at the cloud / edge infrastructure. The RARecom generates as output a new deployment configuration (i.e. new type-level TOSCA), which is fed to the Control Layer for optimizing it and physically deploying it on the cloud / edge infrastructure. Additionally, the architecture contains a Feedback mechanism, which is used to improve the adaptation rules.

Adaptation rules are provided by the DevOps in the form of scalability requirements, which are subsequently translated to scalability rules, e.g "If Average(CPU)>80% then Scale_out". In the previous example, 80% is a threshold provided by the DevOps who also defines abstractly the required adaptation action (Scale_out). This data is then used by adaptation meta-rules which determine the number of additional instances to be allocated to the processing service.

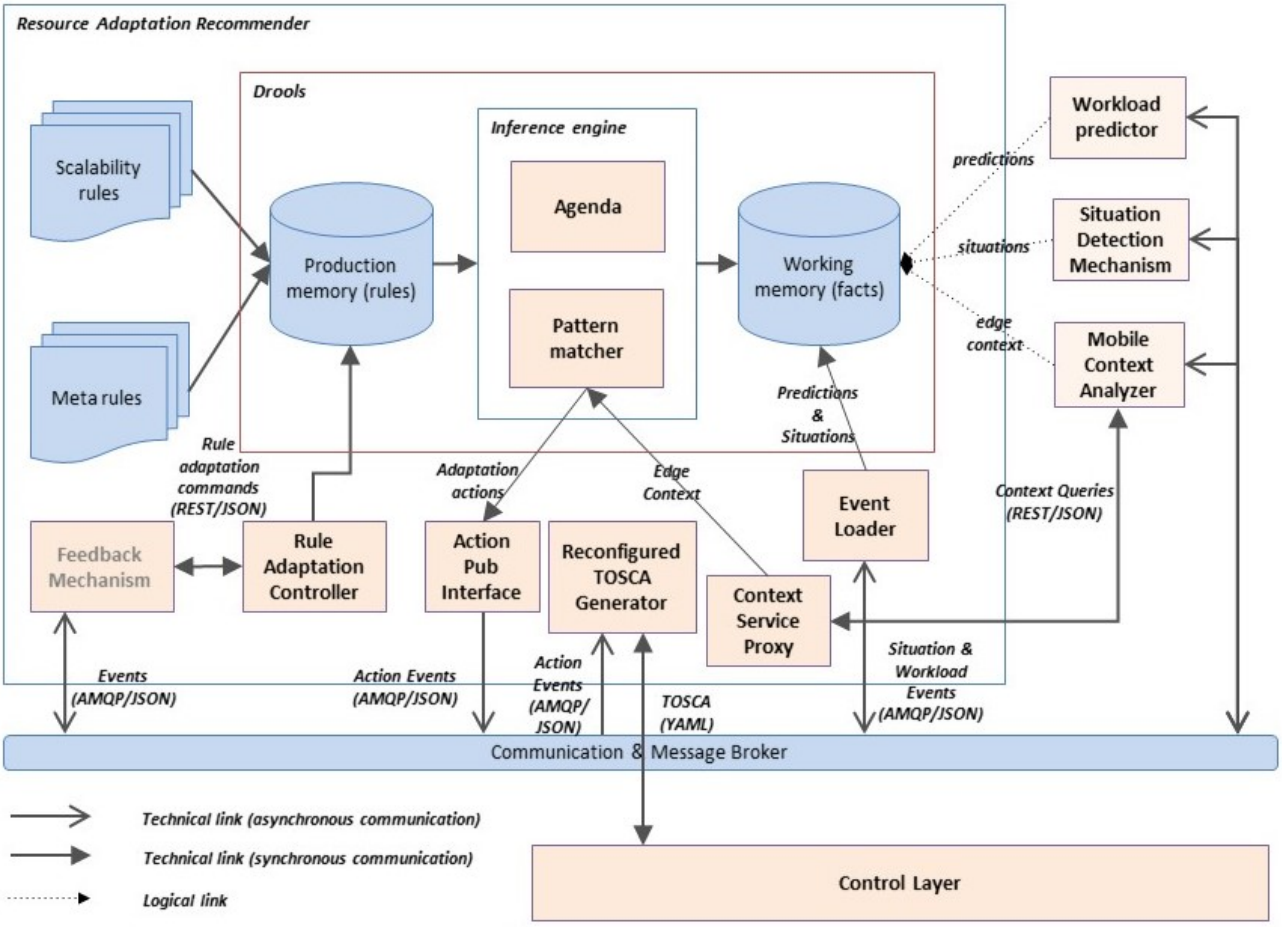

Figure 9. The RARecom Architecture

The Rule Adaptation Controller, which communicates with the active Drools engine session, handles the rule management commands (such as rule addition, rule deletion, rule list fetching) in runtime. The

2018 © Copyright lies with the respective authors and their institutions.

Feedback mechanism uses the Rule Adaptation Controller in order to modify the RARecom adaptation rules in runtime. The ActionPub Interface converts adaptation actions produced by the rules in the Pattern Matcher in JSON format and publishes them to the Message Broker at the desired topic. The Context Service Proxy is responsible for the communication of the Drools engine with the repository (ElasticSearch) of the Mobile Context Analyzer. The Event Loader subscribes to (situation) events produced be the Situation Detection Mechanism (through the Broker) receives them and inserts them into the working memory as JsonObjects (thus it performs the deserialization of the JSON situation payload). Finally, the Reconfigured Tosca Generator receives adaption events produced by the adaptation rules which are published by using the ActionPubInterface, communicates with the Control Layer to receive the current instantiated TOSCA topology and produces a new TOSCA topology which expresses the desired adapted topology. The Reconfigured TOSCA generator sends the new TOSCA topology to the Control Layer for implementation.

3.3 Implementation

The current implementation of the RARecom is available online at: https://gitlab.com/prestocloud-project/resource-adaptation-recommender

The general format of the adaptation logic is that is executed through Drools rules is the following:

- When
  - a situation (or a combination of situations) occurs
- Then
  - examine if the context conditions are true
  - If they are true produce an event that will trigger adaptation actions

The following example (extract from Rules.drl) shows how the different components of the RARecom are used in the context of adaptation rules.

```
global gr.iccs.presto.RARecom.ActionPubInterface ap;
global gr.iccs.presto.RARecom.ContextServiceProxy ctx;

rule "adaptation-rule-1"
when
   $s: Situation(message == "cpu low") // eventLoader subscribes to Broker and inserts
the received events in Drools working memory
then
   String qryTemplate = ctx.getContextConditionQuery("context-query2"); //context-
query2 : context queries can be abbreviated by short names in the file context-
mappings.properties
   String qry = qryTemplate.replace("~#IP#~",$s.getSource()); // if the context query is
a template we can instantiate it at runtime with data that can come from Situation events
by replacing parameters (i.e. "~#IP#~") with values (i.e. $s.getSource())
   JsonObject $c = ctx.evaluateContextQuery(qry);   // ContextServiceProxy executes the
query and converts the result in an instance of the class JsonObject
   Integer hits = $c.get("hits").getAsJsonObject().get("total").getAsInt(); // Here we
read the value of the parameters hits from the deserialized JSON string that MCA
(ElasticSearch) returned

   if ( hits > 0 ) {   // decision to publish an event that denotes a new adaptation
action
       Action $a = new Action(); // create a new Action object
       $a.setSource($s.getSource());
       $a.setMessage("scale_up");
       $a.setDate(new Date());
       publishAction(ap, "rar-actions", $a); //ap = ActionPublisher Service, "rar-
actions" : the Broker topic
   }
end
```

Listing 6. File RULES.drl

2018 © Copyright lies with the respective authors and their institutions.

The EventLoader subrcibes to situations produced by the Situation Detection Mechanism. When a situation event is detected with the desired properties (for example a situation occurs with message "CPU low") then the system can evaluate one or more context conditions by using the ContextServiceProxy API. The Context Service Proxy executes context queries by communicating with the ElasticSearch Query API of the Mobile Context Analyzer. The results are JSON data which are converted to instances of JsonObject by the ContextServiceProxy. The JsonObject classes provide JSON parsing and deserialization methods which can be used within a rule to retrieve the desired fields returned by the context queries. The most common way to decide whether a context is true will be to examine the number of results returned (total hits). We can retrieve the number of results with the following command:

```
Integer hits = $c.get("hits").getAsJsonObject().get("total").getAsInt();
```

where $c is a JSON object retrieved by the ContextServiceProxy upon executing a query as the one shown in Listing 7. Then the mechanism constructs an Action event which may contain field values retrieved by Situations or context queries or context query templates and publish it using the ActionPubInterface. Names can be given to context queries by using the configuration file context-mappings.properties, as presented in Listing 7. Listing 7 presents a context-query template. Templates are queries that contain parameters that must be defined in runtime. In the specific example a rule that uses this template must replace in runtime the string ~#IP#~ with the source IP address of an event by using the method replace as shown in Listing 6 (command : `qryTemplate.replace("~#IP#~",$s.getSource()))`.

```
context-query2 = { \
  "query" : { \
    "match" : { \
      "src_ip" : { \
        "query" : "~#IP#~" \
      } \
    } \
  } \
}
```

Listing 7. File context context-mapping.properties

The RARecom runs as a Docker service. Upon startup it initializes the Drools production memory by reading and compiling the rules contained in configuration files placed under the folder resources/rules.

These rules can be modified at runtime by the **RuleAdaptationController**. This component provides APIs for the retrieval of a list of the currently running rules (per package), the removal of a specific rule, and the addition of a new rule (by providing the rule name, the package that this rule belongs and the rule text). The **RuleAdaptationController** informs the user if a rule cannot be added because there are (syntactic) errors. This API can be used by the **Feedback Mechanism** in order to modify the Meta-rules of the **RARecom** at runtime and thus improving its effectiveness.

The adaptation action events must be transformed to actual cloud-edge resource adaptation workflows. This process starts with the ReconfiguredToscaGenerator which communicates with the PrEstoCloud control layer in order to retrieve the current TOSCA topology deployed. For example when the adaptation action is "Scale_in" the ReconfiguredToscaGenerator will generate a new type-level TOSCA file to be considered for instantiation by the Application Placement & Scheduling Controller.

3.4 Adaptation rules

A central element in the operation of the RARecom is the creation and enforcement of scalability rules. While existing solutions such as Amazon AWS or Google Cloud Compute also offer to the DevOps the opportunity to set automatic reactions based on the workload monitored, the adaptation policies need a

2018 © Copyright lies with the respective authors and their institutions.

lot of input, are not optimal, and usually need periodic manual evaluation by the DevOps to determine whether they are still efficient and relevant.

We propose instead a different approach, which requires from the DevOps a single value for each attribute relevant to the adaptation, called a threshold value. Optionally, the DevOps can also provide PrEstoCloud with a cooling-down period between two adaptations, which determines the minimum interval between two adaptations allowed.

3.4.1 Scalability Rules

The main input of the DevOps to the RARecom are scalability rules, which abstractly define the adaptation actions needed, in relation to specific metrics thresholds. These rules contain the threshold value mentioned above, and are used by appropriate "meta-rules" to determine the actual adaptation action.

The general format of a scalability rule is the following:

$\boldsymbol{if}\ Metric > Threshold\ \boldsymbol{then\ Scale\_out/Scale\_in}$, or $\boldsymbol{if}\ Metric < Threshold\ \boldsymbol{then\ Scale\_in/Scale\_out}$

An instantiated example of a scalability rule would be the following:

$\boldsymbol{if}\ AverageCPU_{cluster} > 80\%\ \boldsymbol{then\ Scale\_out}$

Such scalability rules may contain additional attributes to be defined by the DevOps. For example for the above-mentioned $AverageCPU_{cluster}$ metric, the DevOps can define the time period over which the average value of the metric is calculated.

3.4.2 Meta-rules

Meta-rules describe the concrete parameters of adaptation actions. They use the adaptation threshold value input by the DevOps to create a series of rules for horizontal scaling. These rules take into consideration the observed value of the metric, the threshold input by the DevOps and the maximum/minimum values of the metric (assumed to be known). Meta-rules are evaluated only if a relevant scalability rule has been triggered.

The format of the i_{th} meta-rule is the following:

$\boldsymbol{If}\ LowerBound_i < MetricValue \leq UpperBound_i\ \boldsymbol{then\ Scale\_out/Scale\_in\ by\ f(i)\ instances},\ where\ 0 < i < n$

In both cases, n is the number of meta-rules introduced for each scaling rule, and i is the integer identifier of a meta-rule. Initially, we set n equal to 3, but this value is subject to updates by the Feedback Mechanism. The number of instances to be added or removed, is determined by a function $\boldsymbol{f}$, which is a simple linear function of i . For example $f(i) = i$ can be used for defining concrete increasing number of instances while $f(i) = CurrentNumberOfInstances \cdot \frac{i}{n}$ can be used for expressing the number of instances to be scaled out or scaled in as a proportion of the already deployed cluster (e.g. add 50% more instances).

In the case of scaling out, the upper and lower bounds for the i_{th} (starting from i = 1) meta-rule are determined as follows:

2018 © Copyright lies with the respective authors and their institutions.

$$LowerBound_i = Metric_{threshold} + (Metric_{max} - Metric_{threshold}) \cdot \frac{i-1}{n} \quad , where\ 0 < i < n \text{ and,}$$

$$UpperBound_i = Metric_{threshold} + (Metric_{max} - Metric_{threshold}) \cdot \frac{i}{n} \quad , where\ 0 < i < n$$

In the case of scaling in, the upper and lower bounds for the i_{th} (starting from i = 1) meta-rule are determined as follows:

$$LowerBound_i = Metric_{threshold} - (Metric_{threshold} - Metric_{min}) \cdot \frac{i-1}{n} \quad , where\ 0 < i < n \text{ and,}$$

$$UpperBound_i = Metric_{threshold} - (Metric_{threshold} - Metric_{min}) \cdot \frac{i}{n} \quad , where\ 0 < i < n$$

To better illustrate this, let us consider that the DevOps has provided the following scalability rule as part of the initial deployment requirements:

$$\boldsymbol{If}\ Average(CPU) > 70\%\ \boldsymbol{then\ Scale\_out}$$

Let us also assume that the initial number of instances in the topology is 12, and that the more complex function $f(i) = CurrentNumberOfInstances \cdot \frac{i}{n}$ is used to determine the number of instances in the meta-rules. Since it is an initial deployment n is set to 3, and so the meta-rules which will be initially created are the following:

$$\boldsymbol{Rule\ 1}: \boldsymbol{If}\ 70\% < Average(CPU) \leq 80\%\ \boldsymbol{then\ Scale\_out\ by}\ 4\ \boldsymbol{instances}$$

$$\boldsymbol{Rule\ 2}: \boldsymbol{If}\ 80\% < Average(CPU) \leq 90\%\ \boldsymbol{then\ Scale\_out\ by}\ 8\ \boldsymbol{instances}$$

$$\boldsymbol{Rule\ 3}: \boldsymbol{If}\ 90\% < Average(CPU) \leq 100\%\ \boldsymbol{then\ Scale\_out\ by}\ 12\ \boldsymbol{instances}$$

The meta-rules created initially, can be dynamically updated based on the current number of instances in the topology, and the observed results after an adaptation has been implemented. For example, if it is observed that on the 80% of the scaling out adaptation cases (based on rule 3), a second scaling out action follows (shortly after, based on rule 1) then the rule 3 can be modified by the *Feedback* component as follows:

$$\boldsymbol{Rule\ 3}: \boldsymbol{If}\ 90\% < Average(CPU) \leq 100\%\ \boldsymbol{then\ Scale\_out\ by}\ 16\ \boldsymbol{instances}$$

These improvements on the meta-rules based on prior knowledge will be part of the Feedback Mechanism subcomponent, which will be implemented in the second iteration of the RARecom. Furthermore, while in the approach outlined above only horizontal scaling is used, meta-rules can be enhanced to additionally allow the expression of vertical scaling adaptations.

2018 © Copyright lies with the respective authors and their institutions.

4. Conclusions and Future Work

This deliverable presented the first version of the Data-Intensive Application Fragmentation & Deployment Recommender and the Resources Adaptation Recommender which allow the Control layer to perform a deployment of a cloud application at the level of granularity desired by the developer and following the requirements set by the DevOps. The two recommenders use the type-level TOSCA file as their output in order to send both the initial topology as well as any required reconfiguration.

The annotations system and the policy file used for the DIAFDRecom were designed in a way that permits the definition of new types of requirements and attributes. This enables any adopter of PrEstoCloud to define new metrics inside their annotations scheme, as well as to define new classes of requirements inside the policy file. All of these requirements can be transcribed in a type-level TOSCA file, which can be interpreted by the Control layer.

Our next objective will be the refinement of the Meta-management layer components, as part of the second iteration of T5.1, T5.3 and T5.4. Concerning the DIAFDRecom we will work on improving the policy file input and the type-level TOSCA file output, in order to make the learning curve of the component as low as possible, while maintaining its expressivity. With regards to the RARecom, we will focus mainly on implementing the feedback mechanism and extending the capabilities for facilitating the creation and refinement of adaptivity rules.

2018 © Copyright lies with the respective authors and their institutions.

5. References

OASIS, 2017. Topology and orchestration specification for cloud applications version 1.2, Committee Specification Draft 01, OASIS Standard. Available online: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.2/TOSCA-Simple-Profile-YAML-v1.2.pdf .

2018 © Copyright lies with the respective authors and their institutions.

APPENDIX I: Specification of the type-level TOSCA file

This appendix contains information on the internal structure of the type-level TOSCA file, which is the primary output of the two recommenders. The main sections of the type-level file are the following:

Table 4. Description of the TOSCA segments present in the type-level TOSCA file

Type-level TOSCA segment	Description
Metadata	General requirements, described below in-depth
Description	A textual description of the TOSCA file
Imports	A list of csar (renamed zip compression) files which should be searched for additional node declarations
Node types	Definitions of processing nodes and property nodes, described below in-depth
Relationship types	Definitions of TOSCA relationships
Capability types	Definitions of TOSCA capabilities
Topology template / Policies	A topology_template sub-segment describing various policies which should be followed during deployment, described below in-depth
Topology template / Node templates	A topology_template sub-segment describing the mapping of annotated code fragments to processing nodes, described below in-depth

I.1 The Metadata Segment

```
metadata:
   template_name: ICCS generated types definition
   template_author: ICCS
   template_version: 1.0.0-SNAPSHOT
   CostThreshold: 1000.0
   TimePeriod:  720
   ProviderName_0: Amazon
   ProviderRequired_0: false
   ProviderExcluded_0: true
   ProviderName_1: Google_Cloud_Compute
   ProviderRequired_1: true
   ProviderExcluded_1: false
   MaxInstances: 100
   MetricToMinimize: Cost
```

Listing 8. The TOSCA metadata segment

The metadata segment includes information which characterizes the overall deployment of a TOSCA file, and cannot be included in the subsequent segments. It contains the following fields:

2018 © Copyright lies with the respective authors and their institutions.

Table 5. The TOSCA metadata fields

Metadata Field	Description
template_name	Fields reserved for internal use and documentation
template_author	
template_version	
CostThreshold	The materialization of the Budget Requirement
TimePeriod	
ProviderName_id	Fields reflecting a Provider Requirement. The suffix contains the underscore and the integer id of the requirement
ProviderRequired_id	
ProviderExcluded_id	
MaxInstances	The maximum number of processing instances that can be concurrently used at any single time
MetricToMinimize	The Business Goal set in the policy file

2018 © Copyright lies with the respective authors and their institutions.

I.2 The Node Types segment

```
prestocloud.nodes.jppf.Agent:
   description: A basic JPPF-agent node
   derived_from: tosca.nodes.Root
   capabilities:
      - fragments:
         type: prestocloud.capabilities.jppf.fragmentExecution
   properties:
      component_version:
         type: version
         required: false
   requirements:
      - master:
         capability: prestocloud.capabilities.jppf.endpoint
         node: prestocloud.nodes.jppf.Master
         relationship: prestocloud.relationships.jppf.ConnectsTo

processing_node_eu_prestocloud_application_classes_AudioAnalytics_getData_0:
   description: A TOSCA description of a node
   derived_from: prestocloud.nodes.jppf.Agent
   requirements:
      - host:
         capability: tosca.capabilities.Container
         node: prestocloud.nodes.compute
         relationship: tosca.relationships.HostedOn
         occurrences: [ 1,4 ]
         node_filter:
            capabilities:
               - host:
                  properties:
                     - num_cpus: { in_range: [ 4,8 ] }
                     - mem_size: { in_range: [ 4,8 ] }
                     - disk_size: { in_range: [ 500,2000 ] }
               - os:
                  properties:
                     - architecture: { valid_values: [ x86_64, i386, arm64, armel, armhf ] }
                     - type: { equal: linux }
                     - distribution: { equal: ubuntu }
               - resource:
                  properties:
                     - type: { valid_values: [ cloud, edge ] }
               - excluded_devices:
                  properties:
                     - identifier: { valid_values: [ acfdgex98, kdsfk31fw, f2553fdfs, bd5fgdx32 ] }
```

Listing 9. A sample from the node_types segment

The Node Types segment includes information which is used to correctly implement the hosting requirements of the application. Two types of nodes are contained: Firstly, property nodes which are used to describe concepts specific to PrEstoCloud (e.g. the first node of the two illustrated above describes a JPPF agent). Each of these nodes has unique attributes, defining core elements of the PrEstoCloud platform (described in greater detail in the TOSCA Definitions Creator subsection). Secondly, this segment includes processing nodes which denote the resources and attributes a compute node should possess, in order to undertake the processing of a certain type of a fragment. Processing nodes follow a very specific structure containing the following fields:

Table 6. The fields of a TOSCA processing node

Processing node field	Description
description	Fields which describe the function of a node, indicate possibly inherited properties from a parent node and specify
derived_from	

2018 © Copyright lies with the respective authors and their institutions.

capability	that the node which they describe should be able to perform processing at container-level.
node	
relationship	
occurrences	A range containing the minimum and the maximum number of nodes which can be employed to process the hosted application fragment.
host-num_cpus	A range containing the minimum and maximum number of CPUs which can be used for the processing of the fragment.
host-mem_size	A range containing the minimum and maximum number of GB's that the host should have in its RAM, which can be used for the processing of the fragment.
host-disk_size	A range containing the minimum and maximum disk space (in GB's) which should be available for storage purposes of the fragment.
os-architecture	The acceptable processor architectures that can be used for deployment.
os-type	The family of the operating system (e.g. Windows, Linux, Solaris etc.)
os-distribution	The particular operating system which will be used in the processing node.
resources-type	The permissible types of hosting environments for the particular processing node, which are cloud and edge or cloud (only).
excluded_devices-identifier	The identifiers of the edge devices which cannot be used to host the particular processing node.

2018 © Copyright lies with the respective authors and their institutions.

I.3 The Policies segment

```
topology_template:
   policies:
      - collocation_policy_group_0:
         type: prestocloud.policies.Collocation
         targets: [ eu_prestocloud_tosca_generator_AudioAnalytics_detectActiveIncidents,
 eu_prestocloud_tosca_generator_AudioAnalytics_getData,
 eu_prestocloud_tosca_generator_AudioAnalytics_runAlgorithm ]

      - anti_affinity_group_0:
         type: prestocloud.policies.AntiAffinity
         targets: [ eu_prestocloud_tosca_generator_AudioAnalytics_runAlgorithm,
 eu_prestocloud_tosca_generator_AudioAnalytics_detectShout ]
```

Listing 10. A sample from the policies segment

The policies segment as well as the following node_templates segment, is located inside the topology_template. It contains the collocation and the anti-affinity groups, which reflect the collocation requirements of the PrEstoCloud model. The fragments comprising a collocation group should be collocated if this is possible. On the other hand, the first fragment mentioned inside an anti-affinity group should never be collocated with the rest of the fragments. The groups reflect the collocation requirements which are originally expressed by the developer in the form of dependency and anti-affinity annotations.

Table 7. The fields of a policy node

Policy node field	Description
type	The type of the group, which can either be a collocation group or a group signifying the anti-affinity of a fragment with the rest.
targets	The names of the processing fragments which comprise the group.

I.4 The Node_templates segment

```
 getData_0:
    type: processing_node_eu_prestocloud_application_classes_AudioAnalytics_getData_0

 eu_prestocloud_application_classes_AudioAnalytics_getData:
    type: prestocloud.nodes.fragment
    properties:
       id: 0
       name: eu.prestocloud.application_classes.AudioAnalytics.getData
       onloadable: true
    requirements:
       - execute: getData_0
```

Listing 11. A sample from the node_templates segment

The node templates segment contains information on the assignment of processing fragments to the processing node types defined in the Node Types segment. There are two types of TOSCA nodes which are included in this segment: Fragment nodes, which describe the annotated fragment, and mapping nodes which define where a particular fragment should be processed.

2018 © Copyright lies with the respective authors and their institutions.

Table 8. The fields of a fragment node

Fragment node field	Description
type	The TOSCA type of the node.
id	The id of the property node, which is a monotonically increasing integer.
name	The name of the fragment, more precisely reflecting the actual code hierarchy and the naming of java components.
onloadable	A Boolean variable specifying whether the fragment can be executed on edge devices or not.
execute	The name of the mapping node which will map the current fragment to a processing node type.

Table 9. The fields of a mapping node

Mapping node field	Description
type	The TOSCA type of the node, which should match with one of the processing nodes defined in the Node Types segment.

2018 © Copyright lies with the respective authors and their institutions.