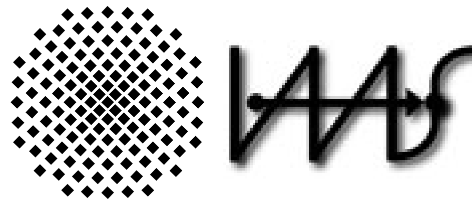
Institute of Architecture of Application Systems
University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master Thesis No. 0838-007

Knowledge Capturing and Usage of Evolving Cloud Application Topologies

Jhonny Vladimir Pincay Nieves



Course of Study: Infotech

Examiner: Prof. Dr. Dr. h. c. Frank Leymann
Supervisor: Santiago Gómez Sáez
Commenced: July 30, 2015
Completed: January 28, 2016

CR-Classification: C.2.4; D.2.8; D.2.11; H.3.3; I.5.3

Abstract

In the last few years, the advent of Cloud Computing has contributed to a considerable increase of the number of service offerings from a variety of providers and every time more applications are being partially or fully deployed in the cloud. Nowadays, experts invest considerable time and effort towards taking the maximum advantage of the benefits that the employment of cloud technologies brings. Such a wide spectrum of cloud offerings, however, increase the number of complex tasks and aspects that must be taken into account when distributing the components of an application such as the quality of service, adaptation to market changes, etc. Furthermore, the distribution of the components and the configuration of cloud resources may evolve over time and there is a lack of tooling support when it comes to assist the developers in the decision-making tasks of selecting an appropriate application distribution topology.

One of the methods that human-beings use to solve problems consists of recalling past experiences and how they solved them at that time. Under this and other considerations the Case-Based Reasoning paradigm was conceived, as a mechanism for solving tasks by recalling past similar problems and adapting their solutions to new situations. This work aims to develop the concepts and mechanisms that enable the capturing and usage of knowledge that the evolution of a system brings along. Specifically, this thesis attempts to identify a set of characteristics that accurately describe a cloud application and to define the models that should be used to identify the cases solved in the past and whose solutions may be useful to solve a new problem. To achieve this, the Case-Based Reasoning with Similarity Retrieval approach is employed, to identify applications with similar characteristics, retrieve their solutions and offer means to refine them in order to obtain a distribution topology that fulfills the requirements of a given application. Furthermore, a prototypical implementation of the approach of this thesis is executed and also employed to validate the concepts and principles that this work follows.

Contents

1 Introduction 1
- 1.1 Problem Statement . . . 1
- 1.2 Motivation Scenario and Research Challenges . . . 2
- 1.3 Outline . . . 4
- 1.4 List of Abbreviations . . . 5

2 Fundamentals 7
- 2.1 Cloud Computing . . . 7
  - 2.1.1 Characteristics . . . 7
  - 2.1.2 Service Delivery Models . . . 8
  - 2.1.3 Deployment Models . . . 9
- 2.2 Cloud Application Topologies . . . 11
  - 2.2.1 Enrichment of Cloud Application Topologies . . . 13
  - 2.2.2 Application Topology Languages and Frameworks . . . 14
- 2.3 Case-Based Reasoning . . . 17
  - 2.3.1 Case Structure . . . 19
  - 2.3.2 CBR Cycle . . . 21
  - 2.3.3 Task Hierarchy . . . 22
- 2.4 Similarity Analysis for Case-Based Reasoning . . . 23
  - 2.4.1 Traditional Similarity Measures . . . 25
  - 2.4.2 The Local-Global Principle . . . 27
  - 2.4.3 Enhancement of Similarity Measures . . . 31
  - 2.4.4 Similarity and Utility Functions . . . 32
- 2.5 Representational State Transfer (REST) . . . 33
  - 2.5.1 RESTful Web Services . . . 34

3 Related Works 35
- 3.1 PatEvol: A Framework for Acquisition and Application of Software Architecture Evolution Knowledge . . . 35
  - 3.1.1 Purpose . . . 35
  - 3.1.2 Approach and Results . . . 35
- 3.2 SMICloud: Framework for Comparing and Ranking Cloud Services . . . 37
  - 3.2.1 Purpose . . . 37
  - 3.2.2 Approach and Results . . . 38
- 3.3 Parallel Cloud Service Selection and Ranking based on QoS History . . . 40
  - 3.3.1 Purpose . . . 40
  - 3.3.2 Approach and Results . . . 40
- 3.4 Graph-Based Analysis and Prediction for Software Evolution . . . 42

3.4.1 Purpose . . . 42
3.4.2 Approach and Results . . . 43
3.5 Evolutionary Algorithm approach for for the Discovery of Software Architectures 45
3.5.1 Purpose . . . 45
3.5.2 Approach and Results . . . 46

4 Concept and Specification 49
4.1 Methodology . . . 49
4.1.1 Topology Modeling - Collection Functional Requirements Data . . . 50
4.1.2 Non Functional Requirements - Data Collection . . . 51
4.1.3 Similarity Measures Calculation . . . 55
4.1.4 Solution Selection and Adaptation . . . 56
4.1.5 Deployment, Monitoring and Knowledge Retrieval and Aggregation . 56
4.2 Formalizing Similarity . . . 57
4.2.1 Similarity of Functional Requirements . . . 57
4.2.2 Similarity of Performance Metrics . . . 59
4.2.3 Similarity of Workload Characteristics . . . 61
4.2.4 Application Similarity Calculation . . . 61
4.3 System Requirements . . . 63
4.3.1 Functional Requirements . . . 63
4.3.2 Non-Functional Requirements . . . 63
4.4 Use Cases . . . 64
4.4.1 Application Developer . . . 64
4.4.2 Domain Expert . . . 64
4.4.3 Use Cases Diagram . . . 65
4.4.4 Use Cases Description . . . 66
4.5 System Overview . . . 78

5 Design 81
5.1 Architectural Overview . . . 81
5.2 Non-functional Aspects Data Model . . . 82
5.3 Modeling Layout Design . . . 84
5.3.1 Performance Requirements Specification . . . 84
5.3.2 Workload Characteristics Specification . . . 84
5.3.3 Discovery of Similar Applications . . . 86
5.4 RESTful API . . . 89

6 Implementation 93
6.1 Modeling . . . 93
6.2 CBR - Similarity Analysis . . . 94
6.2.1 Web Service API . . . 94
6.2.2 Similarity Engine . . . 100
6.2.3 Knowledge Aggregator and Manager . . . 102
6.3 Runtime . . . 102
6.3.1 Provisioning Engine and Monitoring Framework . . . 102

6.3.2 Pricing Knowledge and Cost Calculation Framework 103

7 Validation and Evaluation **105**
7.1 Methodology 105
7.2 Evaluation by Means of Case Study: MediaWiki Application 107
7.2.1 Test Cases Definition 107
7.2.2 Case Retrieval Correctness Evaluation 109
7.2.3 System Behavior after Adaptation Evaluation 114
7.2.4 Domain Expert Operations Validation 116

8 Outcome and Future Work **117**

Bibliography **119**

List of Figures

1.1 Some distribution topologies possibilities for the MediaWiki application [GALS14] 3
1.2 Representation of the approach followed in this work 4

2.1 Cloud computing delivery models including resources managed by the user and examples of each one [ZCB10] 9
2.2 Levels of elasticity and pay-per-use of the different cloud deployment models [FLR$^+$14] 10
2.3 A web shop application topology [AGSLW14] 11
2.4 An extended version *μ-topology* of the web shop application [AGSLW14] . . . 13
2.5 Structure of a Blueprint according to [NLPVDH12] 15
2.6 The TOSCA concepts and their relations [NLPVDH12] 16
2.7 Components of a TOSCA Service Template [top13] 17
2.8 CBR problem-solving approach [Sta03] 18
2.9 An example of an Attribute-Value case representation [Sta03] 20
2.10 The CBR process cycle according to [She03]. 22
2.11 Task decomposition of the processes of CBR [AP94] 23
2.12 A schema of similarity-based retrieval according to [Sta03] 24
2.13 An example of a similarity table [Sta03] 28

3.1 Overview of the processes, activities and repositories of the PatEvol framework [AJP13] 37
3.2 Cloud computing AHP hierarchy as defined by [GVB11] 39
3.3 Overview of the calculations performed on the decision matrix used by the approach proposed by [uRHH14] 42
3.4 Overview of the system developed by [BINF12] 44
3.5 Sample mapping between classes to a tree structure, as suggested by [RRV15] 46

4.1 Cloud application topology enhanced life cycle - CBR Analysis 49
4.2 Processes of the different stages of the methodology [AGSLW14] 50
4.3 Representation of a Case in the proposed system. 51
4.4 Data model of the description of an application 52
4.5 Similarity calculation process. 56
4.6 Use Cases Diagram. 65
4.7 System Architecture Overview 78

5.1 Architectural overview of the system. 81
5.2 Cloud application non-functional aspects data model 83
5.3 Performance specification layout 85
5.4 Workload specification layout 85
5.5 Discover similar applications layout 87

5.6 Refine application and cost calculation layout. In the refinement interface, the white boxes represent the nodes of the depicted α-topology and the gray boxes the γ-topology, together they conform the proposed μ-topology. 88

6.1 Winery Topology Modeler interface extended with the menu Similarity Analysis added 94

7.1 Evaluation methodology and task 106
7.2 MediaWiki Application depicted as a topology 107
7.3 α-topology of the test case modeled in Perfinery 109
7.4 Specification of workload characteristics in Perfinery 110
7.5 Specification of Performance requirements of the category Resource Utilization in Perfinery 111
7.9 Part of the retrieved knowledge from an application 111
7.6 Performance Requirements currently added 112
7.7 Results of the similarity analysis 112
7.8 View of two viable distributions retrieved through the similarity engine . . . 113
7.10 Refining a selected viable topology 114
7.11 Cost calculation interface in Perfinery 115
7.12 Retrieval of similar applications after a new case and solution have been inserted into the knowledge base 115
7.13 Body of the obtained response when invoking the REST function of retrieving similarity tables in Postman 116

List of Tables

4.1 Application QoS performance metrics per category 53
4.2 Attribute-based characterization of the Workload of an application 55
4.3 Sample of a similarity table for *type of application* 58
4.4 Description of Use Case: Model Alpha Topology 66
4.5 Description of Use Case: Specify Performance Requirements 67
4.6 Description of Use Case: Specify Workload Characteristics 68
4.7 Description of Use Case: Specify Hard Constraints 69
4.8 Description of Use Case: Calculate Distribution Cost 69
4.9 Description of Use Case: View Model 70
4.10 Description of Use Case: Refine Application 70
4.11 Description of Use Case: Retrieve Deployment Package 70
4.12 Description of Use Case: Discover Similar Applications 71
4.13 Description of Use Case: Compute Similarity 72
4.14 Description of Use Case: Retrieve Viable Distribution 73
4.15 Description of Use Case: Store Adapted Solution 73
4.16 Description of Use Case: Persist Viable Distribution 74
4.17 Description of Use Case: Persist Knowledge 75
4.18 Description of Use Case: Retrieve Knowledge 76
4.19 Description of Use Case: Update Knowledge 76
4.20 Description of Use Case: Retrieve Similarity Tables 77
4.21 Description of Use Case: Update Similarity Table 77

5.1 REST API summary 89
5.2 Description of REST method: Discover Similar Applications 90
5.3 Description of REST method: Persist Knowledge 90
5.4 Description of REST method: Retrieve Knowledge 91
5.5 Description of REST method: Update Knowledge 91
5.6 Description of REST method: Retrieve Similarity Tables 92
5.7 Description of REST method: Update Similarity Table 92

6.1 Details of URIs supported by the CBR Engine 95
6.2 Mapping of existing attributes in the case base to Nefolog available attributes 103

7.1 Performance metrics for the MediaWiki test case 108

List of Listings

6.1 Specification of non-functional characteristics and solution of an application schema . . . 96
6.2 Workload schema . . . 97
6.3 Performance schema . . . 97
6.4 Solution schema . . . 99
6.5 Similarity table schema . . . 99
6.6 Workload similarity computation . . . 100
6.7 Single local similarity computation . . . 101
6.8 Global similarity computation . . . 102
6.9 Example of a Nefolog Candidate Search query string invoked from the CBR framework . . . 104
6.10 Example of a Nefolog Cost Calculator query string invoked from the CBR framework . . . 104

1 Introduction

Conceived upon the principles of High Performance, Grid and Utility Computing [FZRL08, GLZ$^+$10] and considered a major breakthrough in its beginnings, the terms Cloud Computing have become more than just buzzwords. Cloud Computing is without doubts an efficient way to deliver on-demand resources and capabilities, which in conjunction with its pay-per-use schema, caught a lot of attention and nowadays more and more systems are being migrated to the cloud environment.

The massive expansion of the employment of cloud technologies has contributed to a significant increment of service offerings from different providers and along with that, a raise of the number of options and considerations that should be taken into account when migrating partially or completely an application to an off-premise environment.

The expressed above has led to find appropriate methods to express the architecture of an application in order to visualize all its components and proceed with their distribution without overlooking details. One of them allows to describe cloud application topologies in terms of typed-labeled graphs, whose nodes constitute the components of the application being distributed and the edges depict the relations and interactions among them. Furthermore, multiple applications may have similar functions, objectives, structures and therefore it is reasonable to think that the components of a particular one can be reused for other application. Under this thinking and taking into consideration the approach of expressing the distribution of an application as a graph, nodes could be grouped with the objective of conforming reusable sub-topologies and in that way an application can be described either defining its complete stack of elements or in terms of sub-topologies [AGSLW14].

When depicting the topology of an application in terms of different independent sub-topologies, a number of alternatives is going to be available [AGSLW14]. Those options of distributions are known as viable topologies and the selection of the one that fulfills the existing requirements and needs is a task that could be very complex and that the developers have to face and deal with.

1.1 Problem Statement

Considering that nowadays a big number of clouds offerings from different vendors is available in the market and the fact that the components of an application could be deployed in a number of different ways, the task of selecting the most appropriate services and distribution topology is not a straightforward one. Additionally, it is highly probable that an application has to meet certain requirements of performance and workload, increasing even more the complexity of the work of the developer since those two factors need special attention and they may constitute crucial aspects when choosing a distribution topology.

The performance and workload requirements may change over time and they could imply changes in the configuration of resources and distribution of the application components in order to satisfy these new needs. Market changes demand that the personal behind systems that support different processes react adequately, therefore there is a need for developing the concepts and tooling support to enable and allow a rapid response and adaptation to those changes. Furthermore, the task of designing a topology could be cumbersome and demand a lot of effort and even worse, selecting one that does not completely fulfill the requirements or does not use the assigned resources adequately might cause economic losses, among other issues.

Several works have been conducted towards providing tools that facilitate the decision making process of selecting cloud providers such as PatEvol [AJP13] and SMICloud [GVB11]. Nevertheless, those frameworks consider only characteristics of the offerings of some cloud providers and do not take into account the functional and non-functional requirements, nor the evolution of a system and the aspects that produced those changes and therefore they do not offer the necessary support when it comes to design an appropriate application topology.

1.2 Motivation Scenario and Research Challenges

Human-beings tend to recall past experiences solving problems in order to face new ones [She03], one example of this could be when visiting a doctor. If one is not feeling good, a description of the symptoms is given to the specialist, he recalls past cases of people with the same or similar manifestations and using them he may be able to provide a diagnose and a treatment. Following this conception the paradigm of Case-Based Reasoning with Similarity Retrieval was conceived, remembering past problems and solutions to solve a new situation that have similar characteristics [AP94].

When designing the distribution of an application, plenty of deployment alternatives and cloud offerings are going to be available. An example of this is shown in Figure 1.1, where the components of a two-tier application, being in this case the well-known MediaWiki Application, can be distributed in a number of different ways. Finding the optimal distribution, that meets all the existing requirements is a problem of a considerable complexity. The employment of Case-Based Reasoning with Similarity Retrieval can enhance the process of deciding among the different deployment alternatives, since past solutions that have been used to distribute applications with similar characteristics could be adopted as guidelines and in that way it would be possible to reduce the time and complexity that choosing an optimal application distribution takes.

Capturing the conditions that trigger changes in the distribution of an application can be significantly beneficial when performing assessment for comparable applications. If such changes along with the modifications performed to a topology are recorded, they could be used to assist developers and application architects when they are facing a problem with similar characteristics. The employment of Case-Based Reasoning makes sense in this scenario, the aspects that produce changes in the distribution topology constitute requirements that an

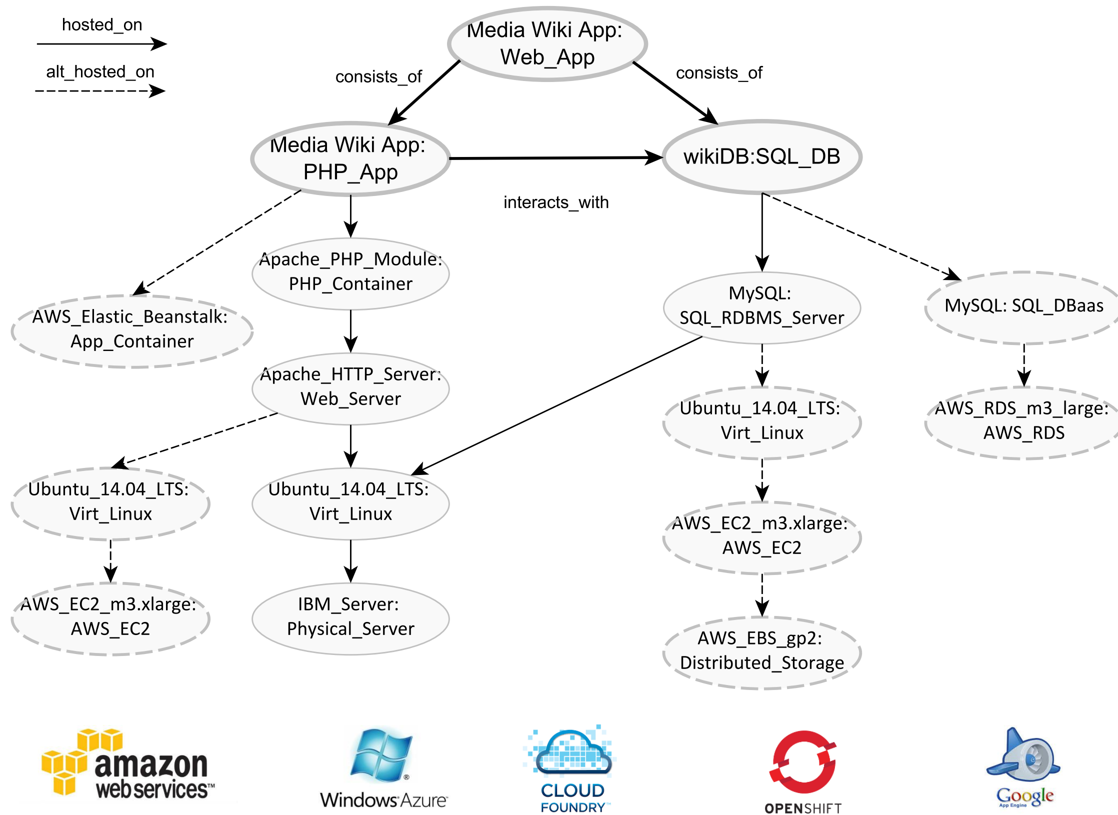


Figure 1.1: Some distribution topologies possibilities for the MediaWiki application [GALS14]

application should meet, meanwhile the resulting distribution is the solution to that problem; therefore, those past experiences can be used to help developers in the tasks related to selecting a viable distribution topology considering not only characteristics of cloud providers, but also the requirements that certain solution accomplishes.

In order to achieve the purpose mentioned above, this thesis focuses on developing means to capture and explode the necessary knowledge to infer and provide potential and valid application distribution alternatives to developers and assist them in the tasks of selecting and refining one appropriate solution that meets all the existing requirements and needs.

Figure 1.2 depicts the approach followed in the development of this work, it consists of creating a knowledge base of past problems and their solutions, accessible to the user through some interface, that also allows him to input a set of requirements and that returns a list of possible solutions from where he can select one, refine it if necessary and store it together with the specified characteristics in order to make them available when solving future problems.

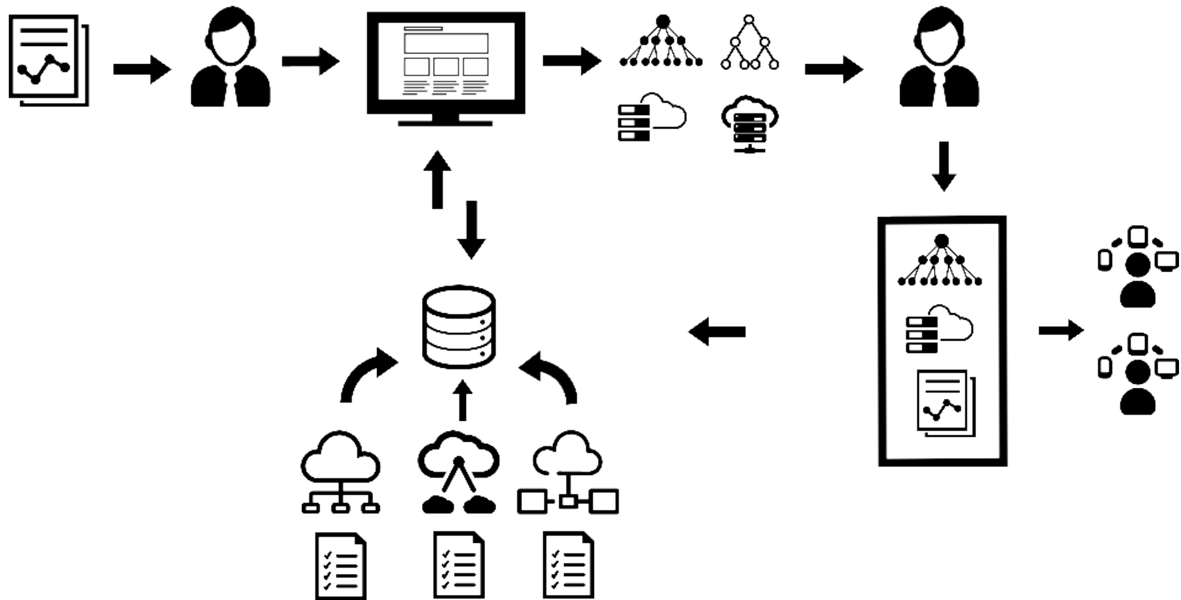


Figure 1.2: Representation of the approach followed in this work

1.3 Outline

The remaining of the present document is structured as follows:

Chapter 2 - Fundamentals: Key concepts and their descriptions as well as required technologies for this work are presented.

Chapter 3 - Related Works: Similar approaches to this thesis, their implementation, findings and issues are described in this chapter.

Chapter 4 - Concept and Specification: The methodology employed in the conception of this work is introduced here. Descriptions of the case representation, data model, use cases as well as the similarity model are described and an overview of the system is analyzed.

Chapter 5 - Design: Details of the architecture of the system are presented, descriptions of the supported data model, RESTful operations and layout design are also provided.

Chapter 6 - Implementation: Well-detailed information of the considerations taken while implementing the different elements of the system are presented in this chapter.

Chapter 7 - Validation and Evaluation: A definition of the methodology used to perform the validation of the system and the results of the evaluation by means of a case study are presented.

Chapter 8 - Outcome and Future Work: The present work is summarized, the obtained results analyzed and a description of future and possible enhancements is provided.

1.4 List of Abbreviations

API Application Programming Interface

AWS Amazon Web Services

CBR Case-Based Reasoning

CB Case Base

CSMIC Cloud Service Measurement Index Consortium

CRUD Create, Read, Update and Delete

CSAR Cloud Service Archive

EC2 Elastic Cloud Compute

DAO Data Access Objects

GENTL GENeralized Topology Language

GUI Graphical User Interface

HATEOAS Hypermedia as the Engine of Application State

HPC High Performance Computing)

HTTP Hyper Text Transfer Protocol

IaaS Infrastructure as a Service

I/O Input/Output

JSON JavaScript Object Notation

JSP Java Server Pages

KPI Key Performance Indicators

MTBF Mean Time Between Failure

MTTR Mean Time To Repair

MVC Model-View Controller

MCDM Multi Criteria Decision Making

NIST National Institute of Standards and Technology

POJO Plain Old Java Object

PaaS Platform as a Service

QoS Quality of Service

REST Representational State Transfer Protocol

RPC Remote Procedural Call

RAM Random Access Memory

RDBMS Relational Database Management System

ROI Return of Investment

S3 Simple Storage Service

SaaS Software as a Service

SBA Service Based Application

SLA Service Level Agreement

SMI Service Measurement Index

SQL Structured Query Language

UML Unified Modeling Language

TOSCA Topology and Orchestration Specification for Cloud Applications

URI Uniform Resource Identifier

URL Uniform Resource Locator

XML Extensible Markup Language

2 Fundamentals

This chapter presents the theoretical fundamentals in which this work is sustained, it introduces basic concepts as well as the employed technologies to develop the proposed solution.

2.1 Cloud Computing

The terms *Cloud Computing* has been around for quite some time by now and they refer to the applications delivered as services over the Internet and also to the hardware, additional software and the infrastructure that allows the delivery of those services [AJP13, AFG+10]. [FLR+14] has stated that Cloud Computing is the logical evolution of Information Technology (IT), considering the current trends of IT outsourcing and division of work. They have compared the nature of its service model as the act of renting a car, where there is a number of different companies offering this service and where the users pay as they use the car. According to [GLZ+10], there are more than twenty different definitions of cloud computing and each one of them seem to only focus on certain aspects instead of providing a full panorama of the technology. For them, cloud computing is based on other existing research areas such as High Performance Computing (HPC), utility and grid computing and the existence of a number of certain attributes that the cloud model has is what differentiates it from the other technologies.

The National Institute of Standards and Technology (NIST), defines Cloud Computing as *"A model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model is composed of five essential characteristics, three service models, and four deployment models."* [BGPCV12].

2.1.1 Characteristics

The NIST enunciates the following characteristics of the cloud model [BGPCV12]:

- *On-demand self-service:* This attribute makes reference to the fact that computing capabilities should be provided as required on an automatic fashion, without the need of interaction between a user and the different provided services [BGPCV12].
- *Broad network access:* The resources provided by a cloud computing system must be available over a high-speed network [FLR+14] and also must guarantee access to a broad variety of clients that may be located on different geographical positions. Furthermore,

a cloud server provider could have data centers around the globe in order to provide high-availability of its services [ZCB10].

- *Resource pooling:* For the sake of dealing with a pay-per-use schema, cloud providers employ large pool of IT resources that is shared among all their customers [FLR+14]. The resource pooling is performed following a multi-tenancy model, where the resources do not belong exclusively to one client, this means that the computing capabilities are assigned and reassigned according to demands of the customers [BGPCV12].
- *Rapid elasticity:* This quality references the fact that resources and computing capabilities are able to scale in or out rapidly, according to the demands of the customer [BGPCV12]. This helps to exploit capabilities of the *economies of scale*, which in the context of cloud computing means that providers offer cloud resources to a large number of clients in order to reduce the cost for individual customers [FLR+14].
- *Measured service:* Another term for this attribute could be pay-per-use and here we could compare cloud resources, such as processing capabilities or storage, to an utility service such as electricity or gas. This means that they are always available and the users pay accordingly to the usage, which should be transparent for both the customer and the provider [GLZ+10].

Additionally to the attributes mentioned above, [GLZ+10] considered that a cloud computing model must be also service oriented, loose coupled, strong fault tolerant, easy to use, TCP/IP based, have high security and virtualization features.

2.1.2 Service Delivery Models

According to the NIST, the different delivery models of cloud computing are [BGPCV12]:

- *Cloud Software as a Service (SaaS):* Access to applications running in the cloud is provided to different users, which share IT resources. The features of self-service, rapid elasticity and pay-per-use are normally granted to them [FLR+14]. It should be pointed that in this model the user does not manage the underlying infrastructure and they may have just access to punctual configurations.

  Some major vendors of SaaS include SalesForce, Google and SAP Business by design [ZCB10].
- *Cloud Paltform as a Service (PaaS):* In this delivery model the provision of platform layer resources, such as operating system support and development framework, is granted to the customer [ZCB10]. On the other hand, the client does not manage the underlying infrastructure including networks, servers, etc. Nevertheless, they may have access to hosting environment configurations [BGPCV12].

  Some examples of providers using the PaaS model are Microsoft Windows Azure, Google App Engine and the Force Platform [ZCB10].

- *Cloud Infrastructure as a Service (IaaS):* In this delivery model the provisioning of on-demand infrastructural resources and capabilities is granted, usually in the form of virtual machines [ZCB10]. Customers are able to deploy and run their own software, including the operating system and applications of some other nature. The capabilities provided to them include processing, network, storage and other computing resources [BGPCV12].

  Examples of vendors of IaaS are Amazon EC2, GoGrid and Rackspace [FLR+14].

Figure 2.1 depicts the different cloud delivery models, the resources that the user is able to manage in each layer and some examples of major providers.

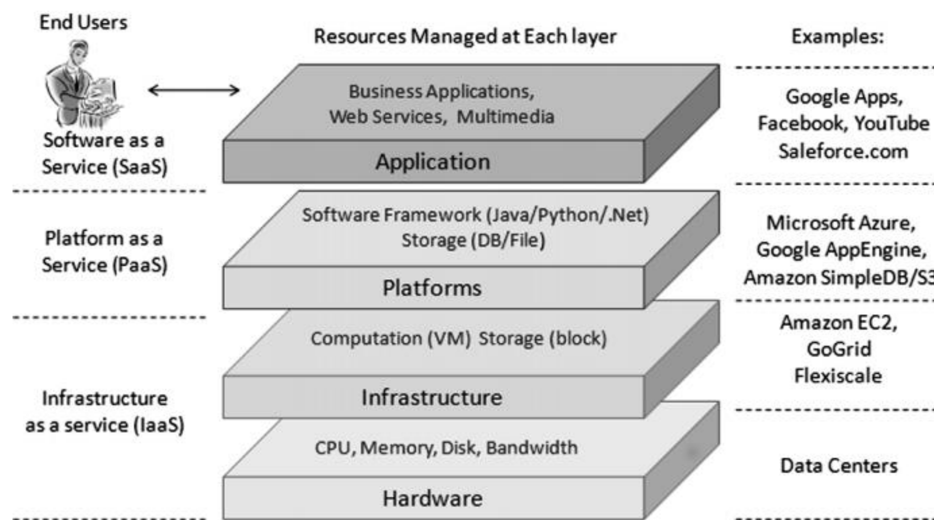


Figure 2.1: Cloud computing delivery models including resources managed by the user and examples of each one [ZCB10]

2.1.3 Deployment Models

According to the NIST definition, there are four major deployment models of cloud computing which are [BGPCV12]:

- *Private cloud:* In a private cloud model the IT infrastructure is provisioned exclusively to a single organization which may have different units [BGPCV12]. This is done with the objective of providing high levels of privacy trust and security, ensuring also a good and elastic use of a static pool of resources [FLR+14].

  This approach is the one offering the highest degree of control for an organization, nevertheless it may resemble the traditional server farms and may not offer the benefit of cost saving as in other delivery models [ZCB10]. A company offering tools to establish and manage private clouds is VMware, there are other several open source software options which support this delivery model such as Eucalyptus, OpenNebula and OpenStack [FLR+14].

- *Community cloud:* In this model the use of the cloud infrastructure is provided to an exclusive community of organizations that may share similar interests and trust each other [BGPCV12]. Technologies that allow the creation of community clouds are the same ones used to create private clouds, but in this case the users accessing it can belong to different organizations [FLR+14].

- *Public cloud:* In this deployment model, IT resources and computing capabilities are provided to a very large number of users, with the objective of allowing elastic use of a static resource pool [FLR+14]. Public clouds offer several advantages such as the no need of investment and maintenance of infrastructure as well as possible lower tariffs, considering that the resource pool is shared with a big number of other users. On the other hand, there may be some issues regarding security and control over data which could be really important in specific scenarios [ZCB10].

  Some examples of vendors offering their services in public cloud model are Amazon EC2, Google App Engine and Microsoft Windows Azure [FLR+14].

- *Hybrid cloud:* A hybrid cloud is a combination of public, community and private clouds and it tries to take advantage of the benefits of the mentioned types while addressing their limitations [ZCB10]. In other words, the goal is to obtain a homogeneous hosting environment with high levels of data control and security [FLR+14].

  Big challenges that this approach presents are related to the specification of which components should be public, which ones should be private and how should they communicate and interact [ZCB10], but once these challenges are overcome, a organization will count with a highly reliable IT cloud infrastructure.

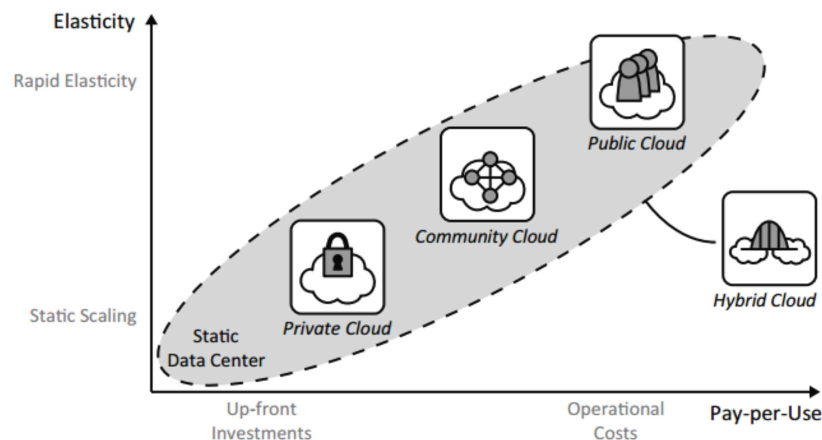


Figure 2.2: Levels of elasticity and pay-per-use of the different cloud deployment models [FLR+14]

One aspect that should be considered is that each of the delivery models will present different levels regarding elasticity and costs and depending on the business needs, these may be influential factors when deciding which model should be adopted. Figure 2.2 shows the level of elasticity and pay-per-use of the deployment models mentioned above.

2.2 Cloud Application Topologies

Multi-tired distributed applications and applications running in the cloud are composed of different elements interacting between them [Reu13]. An application topology can be defined as a typed labeled graph whose nodes represent the components of the application and the edges their different relations, dependences and interactions [Gan15]. [AGSLW14] had provided the following formal definition for the term *application topology*:

An application topology is a labeled graph $G = (N^L, E^L, s, t)$ where N is a set of nodes, E is a set of edges, L a set of labels, and s, t the source and target functions $s, t : E^L \rightarrow N^L$. The topology graph is called typed, if the label set L contains only elements $< name : type >$ (for nodes) and $< type >$ (for edges), in which case the graph is denoted by T.

This approach has been widely adopted for several topology description languages and frameworks such as the Topology and Orchestration Specification for Cloud Applications (TOSCA), Cloud Blueprints and CloudML. The MOCCA framework also employs this graph-based topology approach in order to provide a concrete description of the architecture of an application and its components [AGSLW14, Reu13].

Figure 2.3: A web shop application topology [AGSLW14]

An example of a cloud application topology can be observed in the Figure 2.3. The described architecture corresponds to a web shop, its different tiers, components and relations between them are depicted as well as the type of server where they are deployed. When denoting or describing an application through a topology there may be a number of different possibilities,

an application topology offers just a limited view of all the possible distributions across the cloud solutions [AGSLW14].

Cloud applications have components that are specific to its functional requirements, but there may be also components that can be reused or shared by similar applications. [AGSLW14] in their work introduced the definition of type graph with inheritance, with the objective of modeling and exploring the possibilities previously stated:

A type graph with inheritance TG_I is a triple (TG, I, A) consisting of a type graph $TG = (N, E, s, t)$ (with a set of nodes N, a set of edges E and a target function $s, t : E \to N$), an inheritance graph I sharing the same set of nodes N, and a set $N^A \subseteq N$, called abstract nodes. For each node $n\epsilon I$ the inheritance clan relation is defined by $clan(n)_I = \{n'\epsilon N|\exists pathn' \to n\epsilon I\}$ where $n\epsilon clan(n)_I$ (i.e. the path of the length 0 is included).

In that sense TG_i is a graph whose nodes and edges are types and edges that indicate inheritance or subtype relation type are allowed between nodes. The concept of abstract nodes are used to call generic classes like e.g. operating system, nodes that have only inheritance relations with other nodes. By employing the clan morphism relation $clan(n)_I$ is possible to navigate through the inheritance-types edges in TG_i graphs and considering the application topology as a graph morphism over TG_i, will produce a number of typed topology graphs depending on the existence of sibling nodes in inheritance relations with abstract nodes. Based on this, the concept of viable topology was built [AGSLW14]:

A typed topology T is viable with regards to a type graph with inheritance TG_I, if and only of all elements of T are labeled over the elements of TG_i , for example there exists a graph morphism $m : TG_I \to T$ which uses the inheritance clan relation.

This definition leads to considering the web shop topology, depicted in Figure 2.3, as a *viable topology* under the TG_i graph of Figure 2.4 , this topology contains the same elements as well as relations of Figure 2.3 and additionally it includes more types and subtypes that were not included before and with that an alternative version could be generated. The morphism m that converts TG_I to T can follow a top-down schema, which means that the viable topology T can be generated from a type graph with inheritance TG_I. A bottom-up schema is also possible, meaning that the graph TG_I can be also abstracted from a particular typed topology T. Following this reasonings, the definitions presented bellow were also introduce by [AGSLW14]:

The type graph with inheritance TG_I for a viable application topology T is called μ-topology. The application-specific sub-graph of a μ-topology is denoted by α-topology and by γ-topology the non application-specific and reusable sub-graph of a μ-topology.

In the Figure 2.3 an example of a μ-topology can be observed, the upper part until the dotted line corresponds to the α-topology and the lower part is its γ-topology. It should be pointed that this distinction depends exclusively of the functional nature of the application and the border between them can change according to the application profile and future needs. It is also possible to generate a set of viable topologies v for a determined application, given the α-topology and some generic γ-topology, that can constitute a standard or template topology

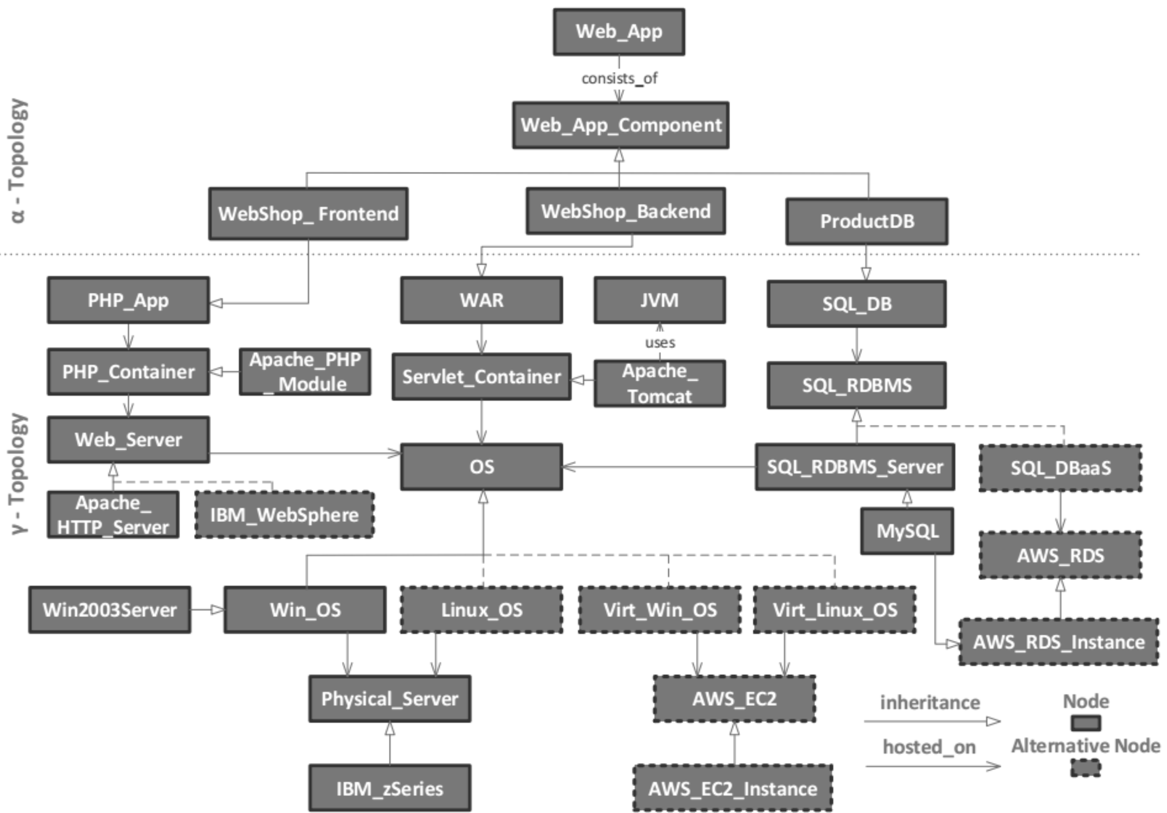


Figure 2.4: An extended version *μ-topology* of the web shop application [AGSLW14]

in a determined context. The results could be assessed by a programmer through some method in order to get the best possible distribution, given requirements of an application.

2.2.1 Enrichment of Cloud Application Topologies

Additional information such as privacy, costs, Quality of Service (QoS), among other non-functional characteristics, are often attached to the application topology. This additional information is called *topology annotations* and they are useful to find out information regarding interoperability, management, discoverability, billing, metering, etc. [Reu13, Gan15]. As described by [DKVR09] , annotations are an effective way to provide a higher level of detail of the microscopic structure of the elements of the graph and their relations and therefore, it is possible to obtain an increase of the description accuracy at the macroscopic level.

According to [Reu13], depending on the usage given to the topology annotations, they can be classified as follows:

- *Discovery:* The annotations used to indicate or describe offerings or requirements of the topology and its elements are known as discovery annotations. Examples of these descriptions can be desired QoS or functional interface requirements [Reu13].

- *Provision and Management:* When an annotation provides information and descriptions regarding the resources needed by an application, the steps that have to be completed to deploy or to execute administration tasks for example, are known as provision and management annotations. They are mostly used to automate the tasks mentioned above [Reu13].
- *Design support:* As its name suggests, this kind of annotations provide support when deciding about the design of a new application topology or the modification of an existing one to adapt it to new environment conditions. They are also very useful in the work of identifying errors or an unfavorable design of a topology [Reu13].

Additionally, if the level of automation in the processing of the information captured in annotations is considered, another classification is possible. If they are entirely processed by machines, they are called automatic processing annotations. In the case that it is intended that the annotations are processed by humans, the type will be human processing and if there is a combination of both, which means that the annotations are processed by machines and also with some human input, then they belong to the group of hybrid processing [Reu13].

2.2.2 Application Topology Languages and Frameworks

Application topology languages are used to represent and describe the elements and relations of the components of a topology, most of them in a graphical fashion. It is also possible to include in the representations the non-functional properties, specified by the annotations [Gan15] some examples include Blueprints [PvdH11], TOSCA [top13] and GENTL [ARSL14]. Furthermore, there also frameworks that support the automatic deployment of composite applications in the cloud such as Cafe and MOCCA [Reu13]. Descriptions of Cloud Blueprints and TOSCA are provided below:

Cloud Blueprints

Cloud Blueprints is an approach that aims to provide flexibility to service-based application (SBA) developers when it comes to customization, since most SaaS, PaaS and IaaS offerings are monolithic solutions that practically leave no options for syndicating the services from multiple Saas, Pass, or IaaS vendors and cases where this is the way to go are very likely to appear. Basically a *Blueprint* is an abstract description of a cloud offering that may cross the different cloud computing levels (SaaS, PaaS, IaaS) [NLPVDH12].

The exposed above is achieved by employing a so-called *Blueprint Template*, that allows to specify and formalize an architecture that combines offerings from different vendors [Reu13].

According to [NLPVDH12], a blueprint has the following structure, depicted also in the Figure 2.5:

- *Basic properties:* They are descriptions that allow to identify the blueprint such as an unique ID, version, etc.
- *Offering:* Description of the different cloud offerings, they may include QoS, elasticity characteristics, etc.
- *Implementation Artifacts:* The necessary elements to implement the offerings such as binary and configuration files.
- *Resource requirements:* This corresponds basically to the required resources to correctly deploy the implementation artifacts.
- *Virtual architecture:* Description of the architecture to be implemented by the developers. This is defined by the interactions and dependences of the different offerings, implementation artifacts and resources.
- *Policy:* Policies essentially describe all the rules and constraints that must be followed by all the elements of the blueprint.

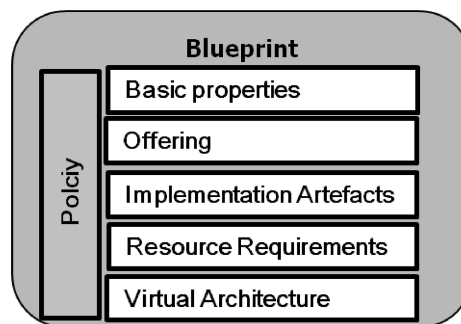


Figure 2.5: Structure of a Blueprint according to [NLPVDH12]

As a result a graph where the nodes are the different parts of the blueprint and the edges correspond to the links and dependences between them is obtained, this is the application topology and it is a representation of the virtual architecture. Reusability is an really important aspect for the blueprint approach, that is why the obtained topologies can be deployed to a repository and could be employed by other developers as base of the architecture of similar applications [Reu13].

Topology and Orchestration Specification for Cloud Applications (TOSCA)

TOSCA is a standard developed by OASIS, its core specification provides a language that allows to describe service components and their relationships using a so-called *Service Topology*. It also provides means for describing management procedures that create or modify services using orchestration processes. Topologies and orchestration descriptions can be combined in a service template that describes the elements that are necessary to keep, in order to perform deployments in different environments [top13]. TOSCA also grants capabilities to

automatically deploy composite application, manage them and performing tasks such as scaling and backing up [BBKL14a].

A core concept of TOSCA is the *Topology Template*, also known as the topology model of a service, it basically defines its structure by describing its components and relationships and also well-defined declarations of its management capabilities. Another important concept of TOSCA are the *Management Plans* which provides higher-level management tasks that are used to create and terminate the service and administrate it during the whole system lifetime [BBKL14a, top13].

Figure 2.6 provides a TOSCA-based description of an hypothetical application and how the TOSCA concepts relate.

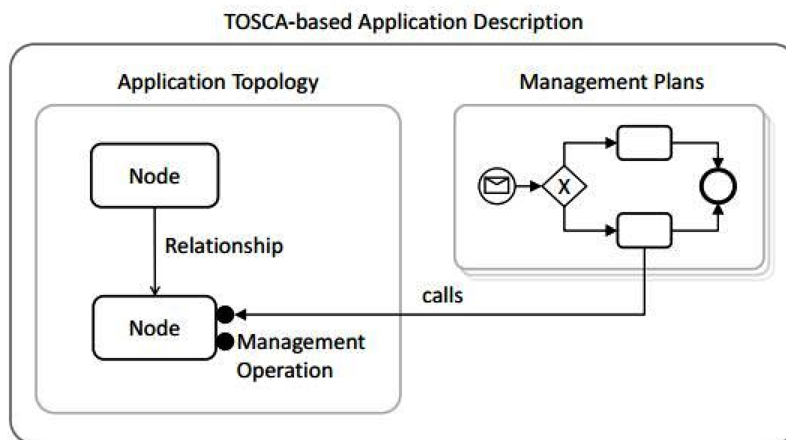


Figure 2.6: The TOSCA concepts and their relations [NLPVDH12]

As previously mentioned, TOSCA employs a service template where the information of the topology of a service as well as the orchestration of its processes are combined in order to provide the required definitions for interoperable deployment of cloud services [Reu13]. The different components of the service template are depicted in the Figure 2.7 and are described bellow [top13]:

- *Node type:* It defines the properties of a component of a service and the operations that exists to manipulate such a component, it is possible to reuse the node type definitions too. One example of a property defined on the node types could be the IP address of an instance of this type.
- *Node template:* This component references a node type and adds constraints and restrictions to it, for example the number of times that a component can occur. Following the example of the previous point, in this case the node template can specify a range of IP addresses for the instances.
- *Relationship type:* It specifies semantics and properties of a relationship.

- *Relationship template:* This component specifies how the relationships among nodes in a topology occur, this means that here the elements that certain relationship connects, its direction and constraints are defined.
- *Topology template:* A graph is formed when considering elements of the service as nodes and their relationships as edges. This graph is represented by the topology template as a set of nodes and relationship templates.
- *Plans:* They are defined as process models and for the sake of their specification an appropriate language, like BPEL or BPMN, should be used . Plans basically describe the management aspects of the service instances and they provide guidelines when creating and terminating them.

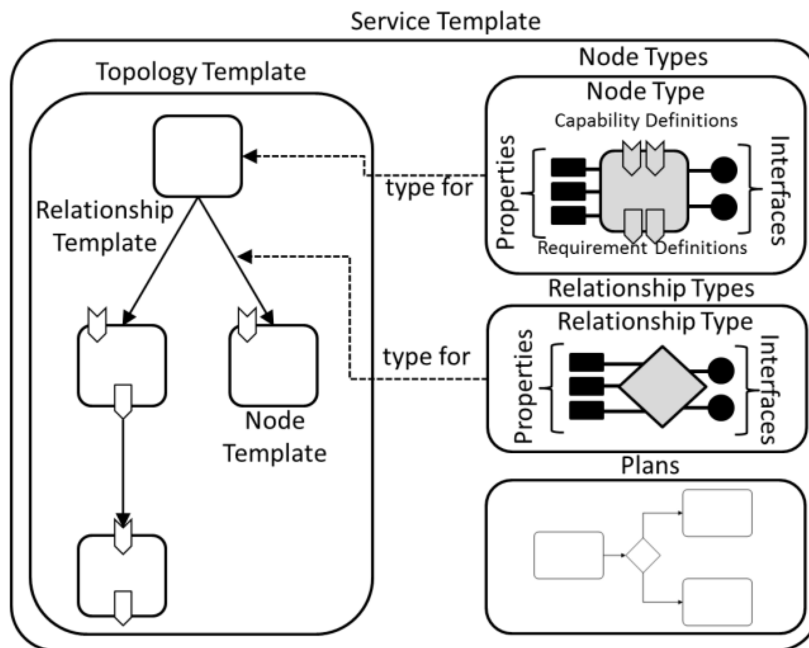


Figure 2.7: Components of a TOSCA Service Template [top13]

TOSCA also makes use of *Policy Types* and *Policy Templates* to describe and specify non-functional behaviors and QoS of a node type and a node template respectively [Reu13].

The services and cloud applications in TOSCA are packaged in a Cloud Service Archive (CSAR) file, which contains a hierarchical list of folders. Inside the folders several files are located and they specify and implement the different templates, nodes and relationships of the service [top13].

2.3 Case-Based Reasoning

Case-Based Reasoning (CBR) is a problem solving paradigm first formalized in the 80s, when it emerged as one alternative to solve problems of medical decision-making systems. This

approach basically uses the knowledge acquired from previous experiences in order to try to solve new problems with similar characteristics. This reasoning comes from the fact that human-beings almost all the time solve problems by remembering past experiences (cases) and adapting them to the new conditions they are facing [RAS+14]. In other words CBR states that similar problems are best solved with similar solutions [She03].

As previously exposed, the main idea of the CBR paradigm is learning from the experience of previous similar experiences. According to [She03], the most important point of CBR lies in the term *similar* which means that not necessarily a fully identical problem had to be solved in the past, but one with some characteristics in common. Another important aspect of CBR is that it is an incremental-sustained learning approach, this means that new experiences obtained when solving a problem are retained and they can be used immediately when facing new situations [AP94].

According to [She03], *"CBR is a technique for managing and using knowledge that can be organized as discrete abstractions of events or entities that are limited in time and space"*, those abstractions are the so-called *cases*. In the field of software architecture, cases are denoted by vectors of features that correspond for example to number of interfaces, size, development method, etc. Systems that solve problems following this approach normally make use of a repository where all the cases are stored and when a new problem arises, the cases and their solutions are retrieved, some analysis are conducted and finally one or several alternatives to solve the problem are proposed.

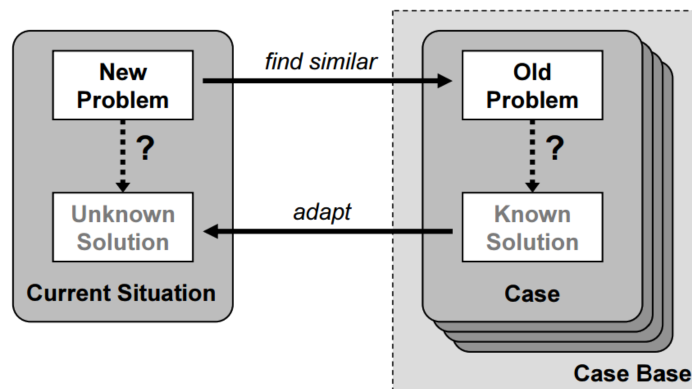


Figure 2.8: CBR problem-solving approach [Sta03]

Figure 2.8 shows in a nutshell how the problem-solving approach of CBR works. In the coming chapters, details about how the CBR paradigm is employed to find solutions from past experiences are provided.

CBR offers different advantages in comparison with other knowledge management techniques [She03]:

1. Many problems related to knowledge codification and elicitation are avoided.

2. Failed cases are also handled, this allows to identify high risk situations.

3. Helps to deal with problems that are part of poorly understood domains because the solutions obtained by CBR are based on past experiences, which is not followed by other hypothesized models.

4. Supports collaboration, in the sense that users using this method to deal with a situation are willing to accept solutions from others and they may collaborate with their found solution as well.

2.3.1 Case Structure

Traditionally a case is only constituted of a problem and a solution part. Nevertheless, a broader structure can also be employed in order to use the cases in a more extensive sense. Considering this, two components of cases can be identified [Sta03]:

- *Characterization:* This part has the necessary information that will help to determinate if a case can or cannot be reused in a certain situation.

- *Lesson:* Basically this part consists only of additional information that might be useful to reuse. This part can also be empty, meaning that only the characterization part is enough to reuse a case.

Making an analogy to the traditional approach of CBR, the characterization part is the problem while the lesson part corresponds to the solution. Furthermore, there are more concepts and formalisms used to describe the components stated above. [Sta03] proposed the use of an *Attribute-Value based representation* of cases, whose basic elements are the attributes and are defined as follows:

"An attribute A is a pair (A_{name}, A_{range}) where A_{name} is an unique label out some name space and A_{range} is the set of valid values that can be assigned to the attribute, also called the value range. Further, $a_{name} \epsilon A_{range} \cup \{undefined\}$ denotes the current value of a given attribute A identified by the label A_{name}".

If there exist attribute values that are unknown or irrelevant, it is possible to use the special value of $undefined$. The value range can contain a collection of elements of a basic type, for example numeric integer, numeric real, symbolic, dates and times. It is also possible to assign a valid rage of values per each attribute, by following one of the undermentioned alternatives:

- Specifying only a basic value type, for example by defining an attribute type as integer, its range of valid values is be taken.

- Specifying intervals for numeric types for example integer values of the interval [0,100].

- Explicitly defining or enumerating the allowed values, in the case of symbolic types an enumeration of colors can be use red, blue, green.

It is possible now to formally define case characterization and lesson part [Sta03]:

"A case characterization model is a finite, ordered list of attributes $D = (A_1, A_2, ..., A_n)$ *with* $n > 0$. $\widehat{D}$ *denotes the space of case characterization models"*

"A lesson model is a finite, ordered list of attributes $L = (A_1, A_2, ..., A_n)$ *with* $n \geq 0$. $\widehat{L}$ *denotes the space of lesson models"*

The concepts stated above allow to introduce a formal description of a *case* using the attribute-value representation [Sta03]:

"A case model is a pair $C = (D, L) = ((A_1, A_2, ..., A_n), (A_{n+1}, A_{n+2}, ..., A_m) \epsilon \hat{D} \; x \; \hat{L}$ *with* $m \geq n$. $\widehat{C}$ *is used to denote the space of case models."*

"A case according to a given case model $C \epsilon \widehat{C}$ *is a pair* $c = (d, l)$ *where* $d = (a_1, a_2, ..., a_n)$ *with* $n > 0$ *and* $l = (a_{n+1}, a_{n+2}, ...a_m)$ *with* $m \geq n$ *are vectors of attribute values and* $a_i \epsilon A_{irange} \cup \{undefined\}$ *is the value of the attribute* A_i. *Further, the vector d is called the case characterization and the vector l is called the lesson of c"*

"The set of all valid cases according to a given case model $C \epsilon \widehat{C}$ *is the case space* $\mathbb{C}_c$ *of C. Moreover, the symbol* $\mathbb{D}_D$ *denotes the case characterization space according to a case characterization model* $D \epsilon \widehat{D}$ *and the symbol* $\mathbb{L}_L$ *denotes the lesson space according to a lesson model* $L \epsilon \widehat{L}$*"*

As example of a case model could be a class in the field of object-oriented programming, being an instance of that class a single case. Since a CBR system has a number of cases, it is necessary to define the concept of *Case Base* too [Sta03]:

"A case base CB for a given case model C is a finite set of cases $\{c_1, c_2, ..., c_m\}$ *with* $c_i \epsilon \mathbb{C}_C$.*"*

There is also a special kind of cases called queries:

"Given a case model $C \epsilon \widehat{C}$, *a query is a special case* $q = (d, l) \epsilon \mathbb{C}_C$ *with an empty lesson part l"*

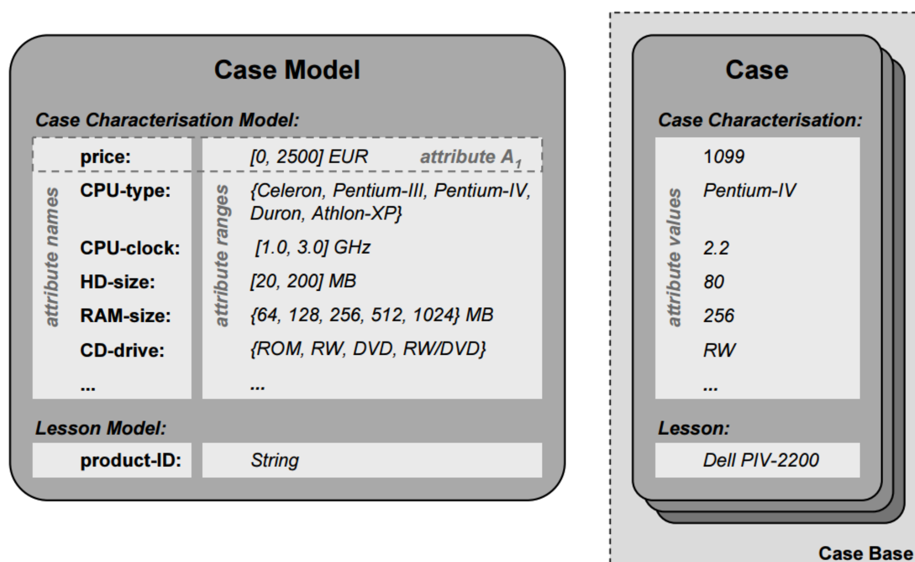


Figure 2.9: An example of an Attribute-Value case representation [Sta03]

An illustration of the definitions introduced in this section are presented in the Figure 2.9. There, the *case base* is constituted of different kinds of computers, they are described on terms of a set attributes and values such as CPU-type, CD-rive, etc. and that correspond to the *characterization model*. The *lessons* are the IDs of the products that meet certain technical properties. When a user inputs into the system the desired set the properties that a computer he is looking for should meet, he is giving values to the attributes that form the characterization model, the CBR system processes that information and looks for similar cases in the knowledge base and returns the lesson of the products that meet the depicted characteristics.

2.3.2 CBR Cycle

[AP94] has identified four processes that constitute the CBR technique, they are also known as the R^4 model:

- *Retrieve* similar cases.
- *Reuse* the information and previous knowledge used to solve that problem.
- *Revise*, check and adapt the proposed solution to fit the new problem.
- *Retain* the problem and the new found solution.

Figure 2.10 illustrates the different processes of the CBR approach and their interactions. A new problem has to be codified into a vector of features or characteristics as first step, those features are the key to retrieve and find the similar cases and their solutions. Once an appropriate solution for the problem is found, it can be reused and revised to adapt it to the current situation. An important process is to retain the new solved case in order to enrich the knowledge base and make it available in the future.

A real life example of application of CBR, proposed by [She03], is the prediction of the number of resources to allocate in the development of a software project; experience in the administration of previous projects is the key to predict the future effort in this new situation. A case is a software component and as example of features we could have the programming language, time frame to develop it, number of interfaces, among others. The features can be of different types such as categorical, discrete or continuous.

The choice of features is arbitrary and the election of the best ones that characterize the problems depends on the problem context and domain. After a new case (target case) arises and it has been described in terms of a feature vector, similar cases from the knowledge base have to be retrieved and also the known effort values which will serve as base to predict the new value for the target case. The value predicted may be modified and adapted by an expert or by the application of rules. Once the process has been completed and the true-effort value is found, it must be added to the knowledge base so it can grow and adapt to new tendencies.

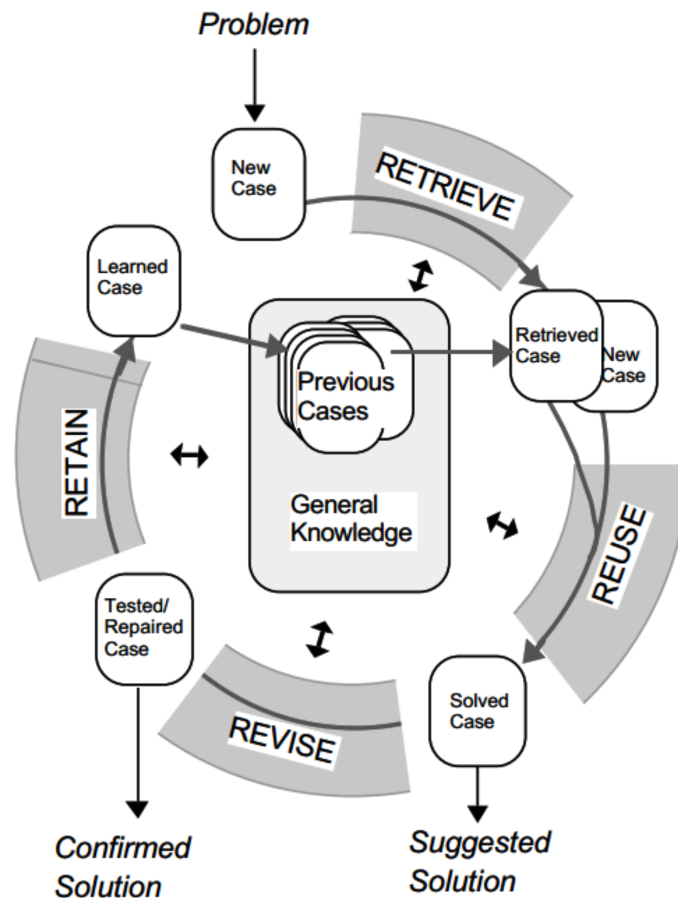


Figure 2.10: The CBR process cycle according to [She03].

2.3.3 Task Hierarchy

The R^4 model previously described provides a general overview of the different processes that CBR follows. Furthermore, those processes can be decomposed in a set of different tasks and activities which describe the detailed mechanism that help to achieve the desired goals.

This task-oriented model follows the principle that a system description can be done from three different perspectives: Tasks, methods and domain knowledge model. The definition and objectives of the different tasks are determined by the system goals and tasks are executed by using or applying a set of methods. At the same time, the employed methods need information about the general domain of the problem as well as information of the current context [She03].

Based on the reasoning stated above, a division of tasks of the four processes of CBR was conceived. Figure 2.11 shows the decomposition of the CBR approach in tasks. the words in bold indicate the name of a task which is split in more low-level tasks that need to be completed in order to complete the top-level task. Dotted lines denote a relation between a task and a method, the methods specify an algorithm that access knowledge and controls

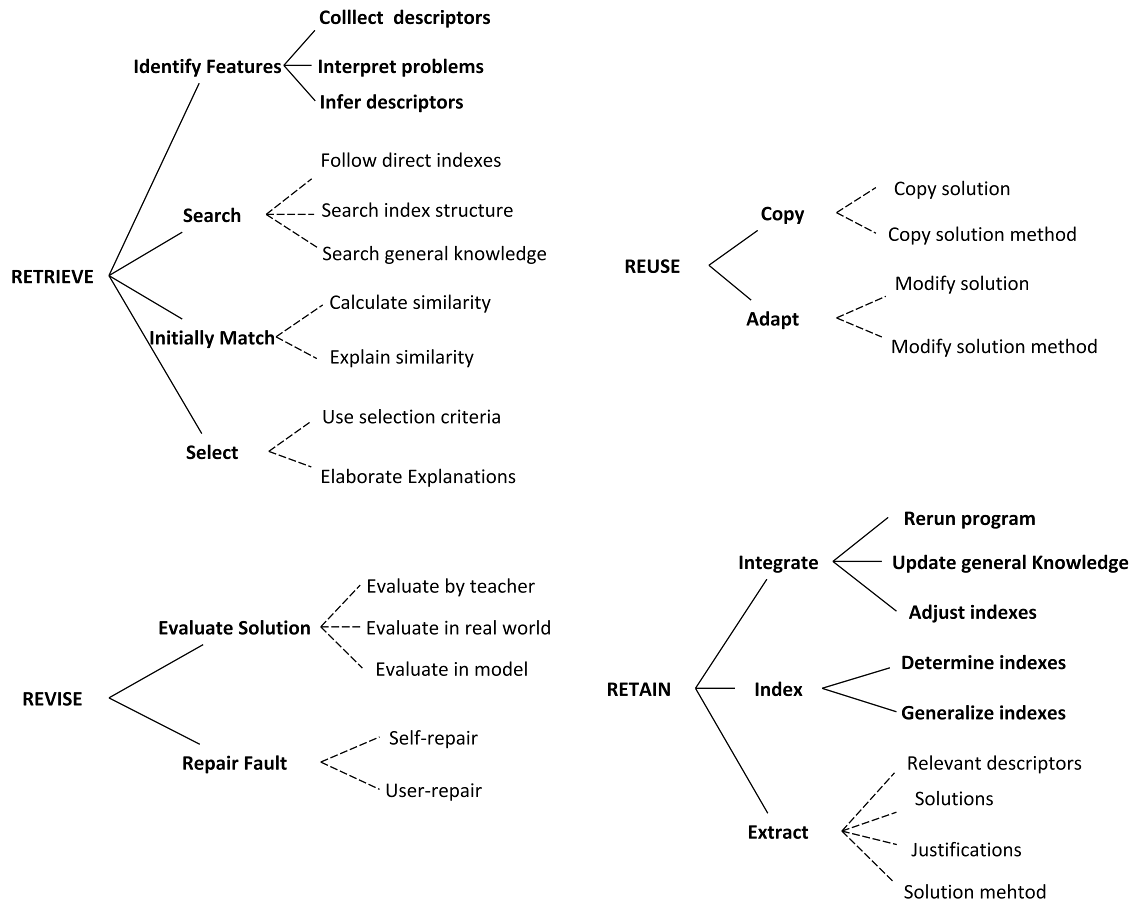


Figure 2.11: Task decomposition of the processes of CBR [AP94]

the execution of a task. At the same time, the methods depicted in the picture could contain further functions or sub-methods necessary to complete the one in the higher level. It should be pointed that a task may be solved just by one method or by a combination of them, not necessarily all of them need to be completed in an specific order, this mostly depends on the domain of the problem. Additionally, no control tasks structures are depicted since the actual control must be specified and be part of the solving problem method [AP94].

2.4 Similarity Analysis for Case-Based Reasoning

Case-based reasoning theory makes a high frequent use of the word *similar* and this is really a key term in this topic. Given an input case, it is expected that the most similar cases are the ones retrieved, therefore the key question at this time is how to determinate them. In order to achieve that, it is necessary to count with a metric that indicates the degree of similarity and based on that the retrieval of the similar cases should performed. This metric is known as *Similarity Measure* and according to [Sta03] it could be defined as follows:

"A similarity measure is a function $SIM = \mathbb{D}x\mathbb{D} \longrightarrow [0,1]$*"*

This means that any similarity measure must be a function whose range of results are between 0 and 1, where 0 indicates that two cases are completely different whereas 1 their equality. Figure 2.12 describes the process of retrieving cases using similarity measures. Given a query q, a case base CB and a similarity measure SIM, similarity values between q and the cases contained in CB have to be calculated in order to identify a list of cases retrieved by a retrieval mechanism and ordered by the value of their similarity measures. The number retrieved cases will depend on the maximal desired number of cases and a value specifying a similarity threshold that defines the minimum similarity that a case c must meet in order to be part of the retrieved list of cases.

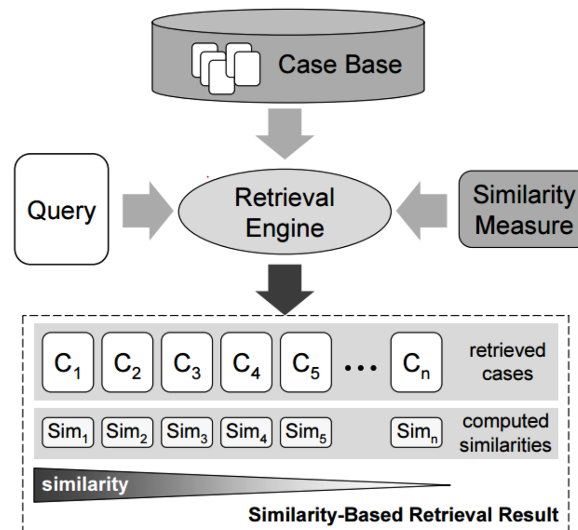


Figure 2.12: A schema of similarity-based retrieval according to [Sta03]

According to the research work conducted by [LZM98], there are two major case retrieval approaches:

- *Distance-based or computational methods*: Which employs some concept to determinate the distance between an input case and a number of cases and the most similar one is determined through the evaluation of such a distance.
- *Representational approach:* Also known as indexing and where the similar case is coded into the structure of the knowledge base itself and the cases are connected by indexing structures, the indexing structures can be traversed to look for the similar case.

Several studies have been conducted in this area and therefore there is a wide range of different approaches, they can belong to one of the groups stated above or even be a combination of them [LZM98, Sta03].

2.4.1 Traditional Similarity Measures

One of the most widely used similarity measures by CBR systems is the *inverse of weighted normalized Euclidian distance* or the equivalent *Hamming distance*. The inverse of the Euclidian distance is given by the expression 2.1 while the the Hamming distance is given by the Equation 2.2, where a weight w_i is normalized since it denotes the importance of the *ith* element, $i = 1, 2, ..., n$ being n the total number of attributes in the cases [MB02].

$$SIM(X,Y) = 1 - DIST(X,Y) = 1 - \sqrt{\Sigma_i \, w_i^2 \, dist^2(x_i, y_i)} \quad (2.1)$$

$$SIM(X,Y) = 1 - DIST(X,Y) = 1 - \Sigma_i \, w_i \, dist(x_i, y_i) \quad (2.2)$$

The normalized $dist(x_i, y_i)$ is represented as in the Equation 2.3.

$$dist(x_i, y_i) = \frac{|x_i - y_i|}{|max_i - min_i|} \quad (2.3)$$

The values of max_i and min_i correspond to the maximum and minimum values of the *ith* attribute. From the same equation can be observed that if $x_i = y_i$, $dist(x_i, y_i) = 0$; otherwise $dist(x_i, y_i) = 1$. From the Equations 2.1 and 2.2 when $DIST = 0$, the value of the similarity SIM reaches its maximum possible value that is 1 and that means that two cases are identical. On the other hand if $DIST = 1$, the value of the similarity is its minimum 0, indicating that two cases are completely different.

According to [She03], other widely used approach based on *nearest neighbor* principles is the one described by the Equation 2.4. As in the previous method, the features must first be standardized with the objective of not considering the influence of the units chosen.

$$SIM(C_1, C_2, P) = \frac{1}{\sqrt{\Sigma_{j \epsilon P} \, Feature_dissimilarity(C_j, C_{2j})}} \quad (2.4)$$

Where P is the set of n features, C_1 and $C2$ are cases and the $Feature_dissimilarity$, which is a functions that expresses how different the cases are and can take the following values:

$$Feature_dissimilarity(C_{1j}, C_{2j}) = \begin{cases} (C_{1j} - C_{2j})^2 & \text{if the features are numeric} \\ 0 & \text{if the features are categorical and } C_{1j} = C_{2j} \\ 1 & \text{if the features are categorical and } C_{1j} \neq C_{2j} \end{cases}$$

Another frequently used similarity measure is based on the ratio mode and was proposed by Tversky in 1977 [LZM98]. It is presented in the Equation 2.5.

$$SIM(X,Y) = \frac{\alpha \times common}{\alpha \times common + \beta \times different} \tag{2.5}$$

Where $common$ and $different$ denote the number of attributes whose values are classified as similar or dissimilar respectively, between the old case and the input case. If the value of similarity between them is above some threshold, then they are classified as similar, otherwise they are dissimilar. α, β represent the weights for $common$ and $different$ respectively.

In 1993, Sebag and Schoennauer proposed a *similarity measure based on rules*. Consider Ω as a problem domain, given two cases X and Y in Ω, the similarity between them $SIM(X,Y)$ is considered to be the number of rules that are fired by both or by none of X and Y. Each rule R_i learned from the case base is assigned a weight $w(R_i)$ depending upon the number of training examples it covers, therefore the similarity is defined as in the Equation 2.6.

$$\forall\ X, Y \epsilon\, \Omega \ \ SIM(X,Y) = \Sigma_i\, w(R_i) \tag{2.6}$$

There are several situations where case matching is not a symmetrical operation. Taking that in account Weber proposed the use of the *contrast model*, where the similarity between a new case X and a target case Y is given by the Equation 2.7 [LZM98].

$$SIM(X,Y) = \theta f(X \cap Y) - \alpha f(X - Y) - \beta f(Y - X) \tag{2.7}$$

The intersection $(X \cap Y)$ denotes the set of attributes that are common to X, Y; $(X - Y)$ and $(Y - X)$ are complement sets that describe the attributes observed only in the target case and in the base case respectively, while θ, α, β represent weighting parameters that assign a different degree of importance to each term. f corresponds to an operation or algorithm that computes the matching score of the relate sets of attributes.

According to [She03], there are many other similarity measure techniques and they include the following:

- *Manually guided induction:* An expert identifies key features in a manual fashion.
- *Template retrieval:* In this case the user inputs values or ranges for a subset of a problem description and all the cases that match are retrieved.
- *Specificity preference:* Cases that match exactly the desired features have more preference than those that do not.
- *Frequency preference:* Cases that have been retrieved the most in the past are given a higher preference.
- *Object-oriented similarity:* Since there may be situations where comparison between cases with different structures need to be performed, this approach proposes to represent the cases as collection of objects where each object has a set of feature and value pairs and organized in a hierarchy.

- *Fuzzy similarity:* This method uses concepts like at-least-as-similar and just-noticeable-difference.

The measures mentioned above present different disadvantages. For example symbolical or categorical features might be problematic, even though some of them attempt through different methods to solve those deficiencies, they tend to follow a Boolean approach. This means that the features either match or not, without considering middle points. Another contra point is the fact that they do not take into account information that can be derived from the structure of the data and this can provoke that higher order relationships are not fully considered. In the case of object oriented similarity, that helps in situations where the cases present different structures, it is necessary to take in account intra- and inter-object similarity. Intra-object similarity is based on common properties, nevertheless the differences between two cases may reside in their class structure rather than in their shared features, therefore it is necessary to take into account the inter-object similarity. Other difficulties of these metrics are related to validation and the fact that in general their approaches are more complex and not so intuitive as an Euclidean distance calculation [She03].

2.4.2 The Local-Global Principle

The representation of a case could be very complex. For example, a problem can be described in terms of attributes whose values can have different types and because of that the similarity measures presented above may not be enough to obtain good results, since they can be used when a set of numeric characteristics is available. An approach that allows to tackle the previously described issue is known as the *Local-Global Principle*, which basically states that the computation of the similarity can be decomposed on a local and a global part [GG15, Sta03].

The local part comprises the calculation of *local similarities* between single attribute values and the global part consists on the aggregation of all those individual local similarity values, in order which is performed to obtain an unique *global similarity* measure which helps in the task of determining whether two cases are similar or not. This decomposition simplifies the problem of having attribute values of different types, since a particular similarity measure can be used for a specific data type and using that is also possible to define well-structured measures for very complex case representations [Sta03].

Local Similarity Measures

Local similarity measures are used to take advantage of very low and detailed data that describes a case, they allow to take into consideration the influence of every single attribute when determining the similarity of two cases. In CBR systems, local similarity measures are defined according to the data type of an attribute [Sta03]. A formal definition of local similarity measure is presented below and it corresponds to a general one, in practice the representation of a local similarity measure depends on the basic value type of the attribute [Sta03].

"A local similarity measure for an attribute A is a function $sim_A : A_{range} \times A_{range} \longrightarrow [0,1]$*, where* A_{range} *is the value range of A."*

Local Similarity Measures for Discrete Value Types

Is a common situation that CBR systems have to deal with attributes whose values are in categorical and discrete ranges. According to [Sta03], the only possible way to represent local similarity for that kind of attributes is through the use of a lookup table, better known as *Similarity Table*.

"Let A be a symbolic attribute with the range $A_{range} = (v_1, v_2, ..., v_n)$*. A* $n \times n$*-matrix with entries* $s_{i,j} \epsilon [0,1]$ *representing the similarity between the query value* $q = v_i$ *and the case value* $c = v_j$ *is called a similarity table for* A_{range}*"*

A similarity table is a very powerful representation for symbolic data types, since is possible to assign values for every single combination of attribute values and in this way is also possible to increase the knowledge quality of the CBR System. On the other hand, having similarity values per each single combination may increase the necessary effort to maintain consistency among all the entries of the table. This approach can be also used for discrete value types where the value range is defined by an explicit enumeration of a finite set of values.

Figure 2.13 shows an example of a similarity table defined for different types of cases of personal computers following some criteria, the main diagonal of the matrix has only values of 1 since they express the similarity of two cases that are equal.

q \ c	laptop	mini-tower	midi-tower	big-tower
laptop	1.0	0.2	0.1	0.0
mini-tower	0.3	1.0	**0.9**	0.5
midi-tower	0.2	**0.7**	1.0	0.7
big-tower	0.1	0.4	0.6	1.0

Figure 2.13: An example of a similarity table [Sta03]

Local Similarity Measures for Numeric Value Types

The similarity table approach is not suitable in this case since the values that the attributes can take is infinite, the approach followed in this situation is based on the difference of the two values being compared and is known as *Difference-Based Similarity Function*, a formal definition is presented below [Sta03].

"Let A be a numeric attribute with the corresponding value range A_{range}*. Under a difference-based similarity function we understand a function* $sim_A : \mathbb{R} \longrightarrow [0,1]$ *that computes a similarity value* $sim_A(\delta(q,c)) = s$ *based on some difference function* $\delta : A_{range} \times A_{range} \longrightarrow \mathbb{R}$*"*

Some of those difference functions include the so called linear difference $\delta(q,c) = c - q$ and the logarithmic difference:

$$\delta(X,Y) = \begin{cases} ln(X) - ln(Y) & \text{for } X, Y \epsilon \mathbb{R}_+ \\ -ln(-X) + ln(-Y) & \text{for } X, Y \epsilon \mathbb{R}_- \\ undefined & \text{else} \end{cases}$$

The difference-based similarity functions follow the assumption that a decrease of the similarity measure implies an increase of the difference between two values, therefore the correct selection and application of them is crucial to obtain high quality results. Some functions typically used by CBR systems are threshold, linear, exponential and sigmoid and their models are presented below [Sta03]:

Threshold function:

$$sim(\delta(q,c)) = \begin{cases} 1 & \delta(q,c) < \Theta \\ 0 & \delta(q,c) \geq \Theta \end{cases}$$

Linear function:

$$sim(\delta(q,c)) = \begin{cases} 1 & \delta(q,c) < min \\ \dfrac{max - \delta(q,c)}{max - min} & min \geq \delta(q,c) \geq max \\ 0 & \delta(q,c) > max \end{cases}$$

Exponential function:

$$sim(\delta(q,c)) = e^{\delta(q,c).\alpha}$$

Sigmoid function:

$$sim(\delta(q,c)) = \frac{1}{e^{\dfrac{\delta(q,c) - \theta}{\alpha}} + 1}$$

This approach is a very convenient way to model similarity of numerical values and it fits in most all domains. However, depending on the problem and specific needs, it may be necessary to define own functions in order to fulfill the distinct conditions of a problem domain. This is not an easy task but it might be necessary to do it if a high quality of results is the objective and the typical difference functions cannot satisfy this demand [Sta03].

Global Similarity Measures

Global similarity measures are used to aggregate all the previously computed local similarity values and obtain an unique value that depicts the similarity between two cases. A formal definition is presented below [Sta03].

Let $D = (A_1, A_2, ..., A_n)$ be a characterization model, $\vec{w}$ be a weight vector and sim_i be a local similarity measure for the attribute A_i. A global similarity measure for D is a function $Sim : \mathbb{D}_D \times \mathbb{D}_D \longrightarrow [0,1]$, of the following form:

$$Sim(q,c) = \pi(sim_1(q.a_1, c.a_1), ..., sim_n(q.a_n, c.a_a), \vec{w})$$

Where $\pi : [0,1]^{2n} \longrightarrow [0,1]$ is called aggregation function that must fulfill the following properties:

- $\forall \vec{w} : \pi(0, ..., 0, \vec{w}) = 0$
- π is increasing monotonously in the arguments representing local similarity values.

The aggregation functions can be very complex, like the ones presented in Section 2.4.1, or very simple as the ones presented below [Sta03], once again the selection of the function depends mostly on the problem domain.

Weighted average aggregation:

$$\pi(sim_1, ..., sim_n, \vec{w}) = \sum_{i=1}^{n} w_i.sim_i$$

Mikownski aggregation:

$$\pi(sim_1, ..., sim_n, \vec{w}) = (\sum_{i=1}^{n} w_i.sim_i^p)^{\frac{1}{p}}$$

Maximum aggregation:

$$\pi(sim_1, ..., sim_n, \vec{w}) = max_{i=1}^{n}(w_i.sim_i)$$

Minimum aggregation:

$$\pi(sim_1, ..., sim_n, \vec{w}) = min_{i=1}^{n}(w_i.sim_i)$$

2.4.3 Enhancement of Similarity Measures

There are some techniques that can be used in order to improve the results obtained by applying any of the similarity measure models previously presented [LZM98].

Case Typicality

When collecting cases to form a case base is usual that some cases contain more information that others and according to [LZM98], cases with more knowledge are more typical cases in the case space. The Equation 2.8 proposed by Agre considers this case typicality:

$$DIST(X,Y) = W_y \times Dist(X,Y) \tag{2.8}$$

In the equation presented above, $DIST(X,Y)$ corresponds to the distance between a new case X and an old case Y without taking into account the case typicality; W_y represents the weight of the old case Y and that weight is calculated as the reciprocal of the typicality, which indicates that if more than one case is found, the one preferred is the more typical. Normally the case typicality is calculated by an expert or by obtaining the average distance between some test case and selected typical cases [LZM98].

Transforming Attributes

Another possible situation when comparing attributes of a case is that its measures come expressed in different scales, this requires the application of transformation in order to have all the problems expressed in equivalent units and later apply CBR. To tackle this issue, Wess and Globing proposed the transformation of a simple symbolic learning algorithm into an equivalent case-based variant. Symbolic learning approaches denote concepts learned explicitly and case-based approaches describe concepts by a pair (CB, SIM), being CB the case base and SIM a measure of similarity. An important aspect of this approach is to identify relevant and irrelevant attributes. An attribute is classified as relevant if it is part of the target concept $X = (x_1, ..., x_n)$. The value of a function f_i is $fi(yi) = 1$ if $x_i = y_i$ and $f_i(y_i) = 0$ otherwise [LZM98].

The functions f_i are combined in one $f : U \longrightarrow \{0,1\}_n, f[(x_1, ..., x_n)] = [f_1(x_1), ..., f_n(x_n)]$. The distance between two cases X and Y is given by a metric called city-block and is defined as in 2.9.

$$DIST(x,Y) = |f_i(x_1) - f_1(y_1)| + ... + |f_n(x_n) - f_n(y_n)| \tag{2.9}$$

The functions fi are learned by an algorithm through selected positive and negative cases. When the concepts are learned, both the function f_i and the CB can be used for classification.

Missing Attributes

Is highly probable that a new case or a stored case will contain missing or null attributes, therefore there are methods that help handling such situations in order to not affect the similarity measurement. The calculation method presented in 2.10 was proposed by Agre and it helps to measure the distance of an attribute with missing values [LZM98]:

$$dist^2(x_i, yi) = \frac{1}{L_i} \times (1 - \frac{1}{L_i}) \tag{2.10}$$

L_i represents the number of possible values of the i_{th} attribute. Ricci and Avesani proposed the model presented in 2.11 to compute the similarity of two cases X and Y for a normalized attribute i with missing values.

$$SIM(x_i, y_i) = 0.5;\ if\ x_i\ or\ y_i\ is\ unknown \tag{2.11}$$

Other approach proposed by Surna and Vanhoof stated that the distance of both missing values is $dist(x_i, y_i) = 0$ and in the case that one is known and the other is not, the distance must be 1. It should be pointed that deciding the value of similarity for two cases with missing attributes also depends on the problem domain and could be defined by experts in order to obtain the most coherent results.

2.4.4 Similarity and Utility Functions

As introduced in Section 2.3, CBR basically consists of retrieving past cases and solutions and use that knowledge in order to solve a new problem with similar characteristics. A key question arises, *how useful are the retrieved cases in the task of providing alternatives of solution to a problem?* and without doubts this is something critical because if at the end of the day the retrieved solutions are not helpful or are inaccurate, then there is not point in using a CBR system . This leads to introduce the concept of *utility* [Sta03, BRS+01].

"A function $u : \mathbb{D}_D \times \mathbb{C}_C \longrightarrow \mathbb{R}$ is called utility function on the case space $\mathbb{C}_C$"

In essence, an utility function assigns a real value to a case C and a characterization D, which represents the utility of c with respect to d. In other words is a measure that indicates how useful is a case stored in the case base to solve a new case or problem. The value of a utility function depends on the particular situation and it is also possible that a number of different utility function exists, this conducts to define the concept of *preference relation induced by utility function*.

"Given a characterization d, a utility function $u(x)$ induces a preference relation $\succeq_d^u$ m on the case space $\mathbb{C}_C$ by $c_i \succeq_d^u c_j$ if and only if $u(d, c_i) \geq u(d, c_j)$."

The utility of the cases as well as the underlying utility function could be affected by several factors [HZ11, Sta03]:

- The domain of the application and current situation.
- The knowledge contained in the CBR system.
- Preference of users.

The points mentioned above are only a few of the many aspects that could affect to utility functions, moreover the main problem is that they are partially known and cannot be calculated beforehand. That is why similarity measures are hugely important for CBR, they allow to approximate the utility of a case and therefore the performance and effectiveness of a CBR system relies almost entirely on the quality of the similarity measures used to retrieve cases. A case base can contain really useful cases but if they are not retrieved, because the similarity measures do not approximate the utility values adequately, then the results provided by such a system will be simply insufficient and unreliable [Sta03].

2.5 Representational State Transfer (REST)

REST is the abstract architecture of the web, is an abstraction of the architectural elements within a distributed hypermedia system [Fie00]. The core idea behind REST is to apply the web architectural style to more sophisticated interactions without caring on details of component implementation and protocol syntax, but focusing on the roles that each component of a system plays and the constraints between their interactions [RR08].

Among the characteristics of the web architecture are [RR08]:

- Interaction with Uniform Resource Identifier (URI) addressable resources.
- A determined number of generic interactions.
- It uses an standardized data format.
- Stateless interactions.
- Context is fully understandable from messages.

The key elements of REST are [RR08]:

- The resources, which are identified by an URI.
- The representations of those resources, which can be of different types such as HTML, XML, CSS, etc.
- The interactions, that must be held by protocols that understand URIs and their representations. Two examples are the HTTP and FTP protocols.

Web services that follows all the principles of the REST architectural style are called RESTful and the considerations that must be taken into account when developing them are presented in the following section.

2.5.1 RESTful Web Services

Integrating application components require that those component employ communication mechanism, one of them is the use of an Application Program Interface (API). The API exposes the set of operations that a client could use in order to interact with a service and those interactions can be designed by following the principles of the REST architectural style. A web API that has been designed following the REST style, is called a REST API [RR08].

As previously mentioned, one of the key components of a REST API are the resources and in order to access them a URI must be available meanwhile the actions performed over those resources are made through some protocol like HTTP. In order to correctly design the URIs, some rules need to be followed and are presented as follows [Mas11]:

- A slash separator(/) must be used to indicate hierarchical relationships and they cannot be used to name resources.
- Lowercase letters should be preferred when naming URI paths.
- File extensions should not be included in URIs. Methods used by HTTP to specify formats should be employed.
- When describing resources, a singular noun should be used for name of documents, a plural noun for collection names and for store names.
- Create, retrieve, update and delete (CRUD) function names should be avoided in URIs.
- The query string component of a URI may be used to filter collections or stores.
- GET and POST methods should not be used to tunnel other methods.
- GET must be used to retrieve a resource, PUT must be used to insert and update a resource, POST only to create a resource and DELETE to remove a resource.
- The HTTP location response header must indicate the URI of a newly created resource.
- Custom HTTP headers must not be used to change the behavior of standard HTTP methods.

3 Related Works

This chapter presents previous approaches and proposal of solutions for the task of capturing and using knowledge of application topology architectures. A total of five works are discussed, from each of them the purposes, methodologies and obtained results are presented. Among the analyzed frameworks are PatEvol [AJP13] which employs concepts of software repository mining and software evolution, SMICloud [GVB11] that based on a set of user specifications returns a list of possible cloud services where he can deploy an specific application and an evolutionary algorithm approach for discovering software architectures. Furthermore, analysis of similarities and differences of the design and implementations of those approaches with the ones accounted in this work are also analyzed.

3.1 PatEvol: A Framework for Acquisition and Application of Software Architecture Evolution Knowledge

3.1.1 Purpose

The Patter-Driven Architecture Evolution (PatEvol) framework, is a proposal developed by researchers of the Dublin City University and it aims to unify the concepts of *Software Repository Mining* and *Software Evolution* in order to enable a continuous acquisition and application of architecture evolution knowledge, to address frequent changes in software architectures in a systematically fashion [AJP13].

The software repository mining process is used to capture knowledge, specifically evolution-centric knowledge, through the analysis of post-mortem evolution histories, meanwhile software evolution concepts are used to enable the employment and application of the discovered knowledge. Furthermore, the PatEvol framework intents to support the reuse of discovered knowledge to enable knowledge application that facilitates the process of evolution-off-the shelf in software architectures

3.1.2 Approach and Results

The authors of this research work have mainly focused on these points:

- *Knowledge acquisition:* Achieved through the use of architecture evolution mining which is a sub-domain of software repository mining and enables the extraction of not obvious predictive information.
- *Knowledge application:* Achieved with architecture evolution which refers to the mapping among the problem-solution views and the application of discovered solutions.

PatEVol was defined as a conceptual framework that describes a set of processes and activities that allows the discovering and reusing of evolution knowledge. The processes of the framework basically point what has to be done and the activities indicates how to do it. The processes and activities defined by the researchers where:

1. *Acquisition of Evolution Knowledge:* As mentioned before, this process consists on architecture evolution mining application which in this context is aimed to employ a set of automated methods for extraction of architecture change instances from log files that are the main source of knowledge. The change logs should be formalized through a notation and the one chose by the authors was a graph-based notation. Activities conducted in this process are:
   a) *Taxonomical classification of architecture change and operational dependencies,* this activity basically consists on categorizing the different changes of an architecture over the time. This classification can be based on the complexity, dependency, etc. of the changes.
   b) *Discovery of architecture evolution patterns,* after completing the previous activity a taxonomical classification of architecture changes is obtained and observing he frequency of the changes among some other techniques, allows to identify some frequently occurring changes that may represent evolution patterns.
   c) *Template-based specification of evolution patterns,* in this activity a catalog of architecture evolutions patterns is created, this allows to share and reuse the evolution patters.
2. *Application of Evolution Knowledge:* This is defined as the systematic implementation of architectural changes to modify an existing architecture. Since most applications tend to evolution over time, the use of discovered knowledge comes very handy to complement and guide the evolution execution. Activities conducted in this process are:
   a) *Specification of architecture evolution,* the source architecture, constraints and the elements that need to be modified, added or removed to reach the desired evolution have to be specified as first step.
   b) *Selection of architecture change patterns,* As result of the previous activity, a document specifying the architectural changes is defined, in this activity the pattern catalog is queried in order to provide a solution based on the evolution context. According to the authors, this is not an easy process and they suggested the use of a methodology Question-Opinion-Criteria to achieve the desired effects.
   c) *Pattern-based evolution of Architectures,* consists on the usage of the obtained pattern in the architectural change execution.

The authors also suggested not to overlook the role of the repositories, since they are the main source of knowledge and storage of identified patterns. Figure 3.1 shows a schema of all the process and activities performed as well as the repositories used by the PatEvol framework. As conclusions, the researchers claimed that PatEvol allows to explode architecture change logs, provides support for pattern specification and instantiation, provides also an evolution

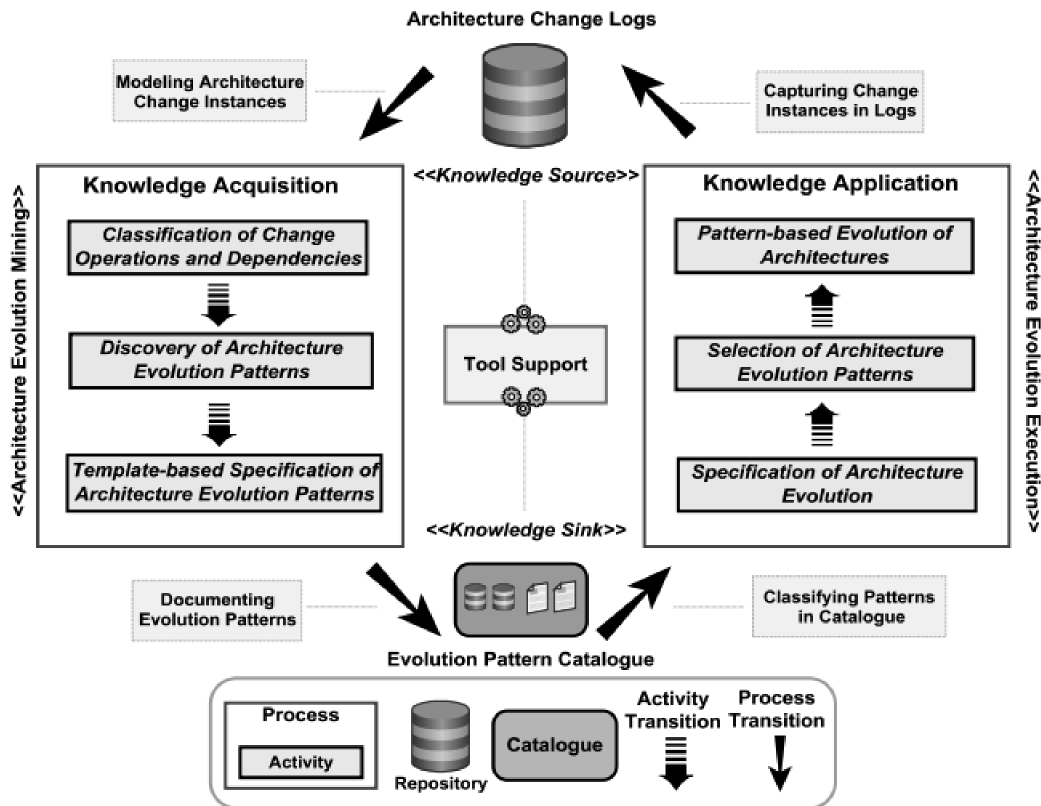


Figure 3.1: Overview of the processes, activities and repositories of the PatEvol framework [AJP13]

application framework that enables pattern reuse and also the discovery of evolution patterns to continuously feed the catalogs.

PatEvol makes use of a knowledge base created from log files and categorizes them in order to obtain patterns that indicate how certain software architecture evolves. In contrast with this thesis, the knowledge that forms our case base comes from descriptions of applications and their solutions depicted as a set of characteristics. When trying to find a solution for a new application instead of looking for patterns, mathematical models are applied to find similar problems and their solutions. Moreover, PatEvol was designed to capture and apply knowledge of software architectures, meanwhile this thesis aims to use the knowledge of distribution of components, not focusing on details of software implementation.

3.2 SMICloud: Framework for Comparing and Ranking Cloud Services

3.2.1 Purpose

This work conducted by [GVB11] aims to assist users in the tasks of evaluating and selecting cloud providers though the use of a framework. Service Measurement Index (SMI), which

consist of a set of Key Performance Indicators (KPI's) that provide a standard method for measuring and comparing a business service and defined by the Cloud Service Measurement Index Consortium (CSMIC) [GVB11], were employed and extended in this approach to let users compare the different cloud offerings, according to their priorities and considering a range of dimensions. Basically, SMICLoud is a decision support system that takes as input user specifications and application requirements and it returns a list of cloud services where the customer can deploy the application.

Among the challenges faced by the researchers were how to measure the variety of SMI attributes, since those attributes may change over time and without the existence of precise measurement model for each attribute, it resulted very difficult to compare them . Another challenge was how to rank the cloud services based on those SMI attributes since there are two types of QoS requirements: Functional and non-functional and some of them are very hard to measure because of the nature of the cloud services. Furthermore, the task of selecting the service that best suits all functional and non-functional requirements was a big problem for them, considering the presence of multiple criteria and the interdependences among them.

3.2.2 Approach and Results

The first issue previously mentioned, was tackled by using historical values and combining them with offered values by cloud providers in order to get the real value of an attribute.

The second issue led the authors to categorize it as a Multi Criteria Decision Making (MCDM) problem and adopt the Analytical Hierarchical Process (AHP) approach to solve it, since each parameter has influence depending on its priority, and this is a based ranking mechanism that among other things assigns weights to features considering interdependence between them.

As previously mentioned, the developed framework was based on SMI and using them SMICloud proposed an holistic view of QoS that a customer needs to consider when selecting a cloud service provider:

- Accountability
- Agility
- Assurance of service
- Cost
- Performance
- Security and Privacy
- Usability

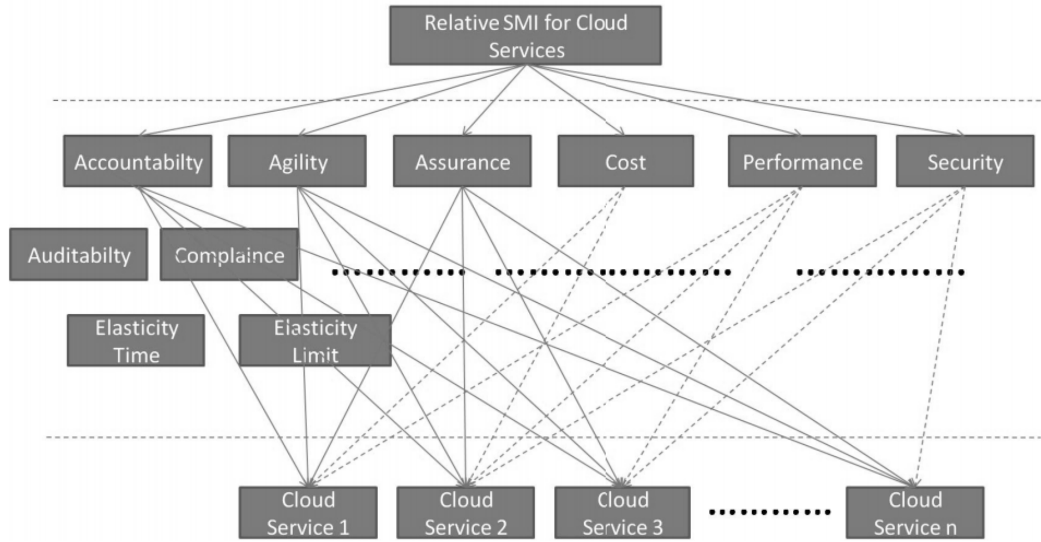


Figure 3.2: Cloud computing AHP hierarchy as defined by [GVB11]

Ranking the cloud services is one of the most important features of this framework and it was computed considering the requirements of the customer and features of cloud services. The ranking mechanism employed by the authors was based on AHP because it provides a flexible way to solve problems with multi-attribute characteristics. To overcome the ranking problem, the researchers decomposed it into four phases:

1. *Hierarchy structure for Cloud Services based on SMI KPIs:* Figure 3.2 shows the hierarchy developed by the authors of this work. First level corresponds just to the goal of the analysis, second layer contains QoS attributes previously described and the bottom most layer presents the cloud services and the QoS attributes that they have.

2. *Computation of relative weights of each QoS and service:* The use of weights to indicate the importance of a service was necessary in order to compare two cloud services. The authors considered two types of weight:

    - User assigned weight: The user assigns weights to the attributes based on a scale provided by the AHP system.

    - Arbitrary user assigned weights: Customers can give weight values in their own scale.

3. *Relative value-based weights for ranking cloud services:* Since the data type of the attributes may have a variety of forms, the researchers developed a ranking model where the dimensional units must be of the same type. An specific model has to be applied depending on the data type and the user must specify if the attribute is essential or not. As a result, a matrix of NxN where N is the number of registered services is obtained and the relative ranking of a service, given an specific attribute, is determined by calculating the eigenvector of the matrix.

4. *Aggregation of relative ranking for each SMI attribute:* This step takes into consideration the weights of each attribute and is executed for all of them. As result, the final ranking of the cloud services is obtained.

As final part of this work, the authors applied the designed framework to a real case situation and according to them the results obtained were good and concluded that the framework effectively faces key issues such as ranking cloud services when having different dimensional units of various QoS attributes and that SMICloud represents a significant next step towards accurate QoS measurement.

SMICloud offers a solid methodology when it comes to choose cloud providers that satisfy QoS requirements of applications. It employs a set of metrics and the MCDM approach to find the candidates that better suit the business needs. Both SMICloud and this thesis, define a set of metrics which are the factors deciding which solution to pick. Nevertheless, SMICloud only offers support when selecting an offering, not when it comes to choosing an appropriate distribution of components among the different providers, which is one of the aspects that the present work attempts to support.

3.3 Parallel Cloud Service Selection and Ranking based on QoS History

3.3.1 Purpose

The purpose of the work developed by [uRHH14] was to assist users in the task of selecting a cloud service provider, but not only considering real-time QoS characteristics but also the historical QoS performance of the services . The authors considered indispensable to take into consideration the QoS history of a service, because the capturing employing only real-time data may lead to select a service at one local maxima performance and therefore it may not be the most appropriate. On the other hand, there are approaches that consider the history of the QoS but they execute just an average calculation that is not enough to capture the frequent variation of the performance of the cloud services.

Considering the facts previously stated, this approach takes into account the multitude of available cloud services, variations in QoS performance, variation of price and also the user's criteria to rank the service and help them to choose the most suitable.

3.3.2 Approach and Results

In order to tackle the problem of selecting a cloud service, considering the historical performance as well as a number of criteria, the MCDM method was employed. The performance of a service in different time slots was considered, the MCDM process in one particular time slot is independent of others and is executed in parallel. The obtained individual results are

later combined using an aggregation method that will help to obtain an overall service rank in the total time period.

The MCDM methods used to rank the services were the Technique for Order Preference by Similarity to Ideal Solution (TOPSIS) and Elimination and Choice Expressing Reality (ELECTRE). In the TOPSIS method, the services are ranked on the basis of the Euclidean distance, previously introduced in Section 2.4.1, of a possible solution with the ideal and the a anti-ideal one, being the closest alternative to the ideal solution and farthest from the anti-ideal the one selected. The ELECTRE method on the other hand determines the pairwise dominance relationship between alternatives.

The architecture of the developed framework relies on integrated QoS information collected from multiple sources such as service specification, a monitoring system and feedback from existing cloud service users. It consists of several modules described bellow:

- *Cloud service discovery:* Searches for cloud environments and their specification. Additionally, it looks for new services and keeps the data up to date.
- *Cloud service monitory:* This module collects QoS data by means of execution of a benchmark test.
- *QoS information repository:* Stores data collected by the service discovery and monitoring modules as well as information from users.
- *MCDM cloud service selection module:* This module employs the information stored in the QoS repository and the criteria from the users in order to perform a multi-criteria analysis on this information and obtain a ranking of the services.

The period of time when the QoS is observed was divided in two parts: pre-interaction and post-interaction phase, additionally a time spot was defined and it corresponds to the period of time when the selection of the cloud service has to be made. The pre-interaction phase is the period before the time spot and the post-interaction corresponds to the period after and here the real-time QoS is monitored in order to determinate if the selected service truly satisfies the needs of the application. In order to perform the MCDM selection process, several steps need to be completed.

1. The pre-interaction phase is divided into different time-slots. Criteria $C_1, C_2, C_3, ...C_n$ are selected by the user and retrieved by the MCDM method from the QoS repository.
2. Since the QoS criteria may not by equally important, the system allowed to input a weight $w_{c1}, w_{c2}, w_{c3}, ...w_{cn}$ per each criterion.
3. The data of the QoS of all available services form a decision matrix and MCDM criteria is applied to get a top service per each time slot.
4. This system also considered as influential the freshness of the data in a determined time slot, and therefore each time slot got a weight assigned and the slot that is closest to the selected time slot gets the highest weight value of 1.0 while the older gets lower value until a minimum of 0.4 is reached.

5. An aggregation process is performed as last step, obtaining as a result a service rank in the pre-interaction phase that is the mean to help the user to make a decision.

An overview of the decision matrix used by the MCDM process as well as of the steps mentioned above are depicted in the Figure 3.3.

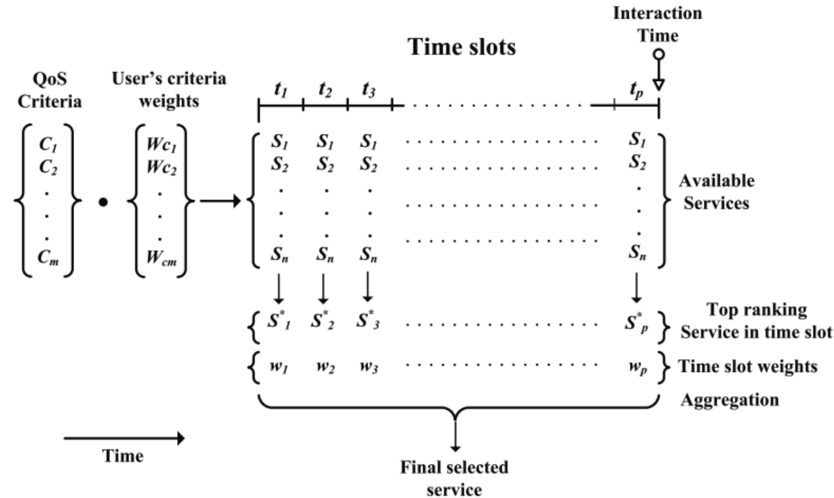


Figure 3.3: Overview of the calculations performed on the decision matrix used by the approach proposed by [uRHH14]

The aggregations were performed following the TOPSIS and ELECTRE methods. Experimental validations of the framework were also performed and the data was obtained from five Amazon EC2 IaaS cloud services during 300 days. The obtained results allowed the authors to conclude that the proposed approach was effective and capable of capturing variations in performance of the services, giving more importance to the most recent data but without discarding older information.

In a similar way as SMICloud, presented in the previous section, this approach provides support to select a cloud provider based on QoS characteristics, not only considering values at some point in time but also taking into account the measures in several time slots in the past. In this sense this approach is similar to the ones followed by this thesis, since both of them consider the historic information of QoS metrics. However, this approach is limited only to rank cloud offerings from different providers and does not help in the task of selecting a possible distribution of application components in the cloud.

3.4 Graph-Based Analysis and Prediction for Software Evolution

3.4.1 Purpose

This work was conducted by researchers of the University of California. It basically consisted on identifying whether a graph-based method, through the use of an accurately constructed graph model of a software system, can help in the tasks mentioned bellow [BINF12]:

- Improvement of software maintenance by identifying which are the components that need to be debugged, tested or refactored first.
- Prediction of the defect count of an upcoming release, based on historical records of previous releases.

Furthermore, the goals of this study were to demonstrate that by deriving information of a set of different graph metrics and topological analysis it is possible to understand how the software evolves, how to construct predictors for software engineering metrics as well as the points exposed bellow:

1. Topological analysis of software-based graphs can reveal properties about software processes.
2. Graph metrics capture significant events in the software life cycle
3. Graph metrics can be used to estimate bug severity, prioritize maintenance effort and predict defect-prone releases.

3.4.2 Approach and Results

In order to achieve the goals of this work, the authors needed to define the methodology on how to build the graph of a system and select the graph metrics to use. Regarding the graph construction, the source of information for its composition were the source code repository where the commit logs, historic source code versions, patches and source code-based developer interactions were stored. The bug tracking of a system constituted another source of data of bug records and bug fixing-developer interactions.

With the information obtained from the two sources mentioned above, three graphs were made:

1. *Source Code-Based Graphs*: This graph helped to capture information regarding function calls, in other words caller-callee relationships.
2. *Module Collaboration Graph:* This one showed information regarding the communication between modules.
3. *Developer Collaboration Graph*: This graph was built to understand how developers collaborated and communicated as the software was evolving. Two kinds of subgraphs were also derived, a bug-developer collaboration graph and a commit-based developer collaboration graph which basically helped to trace the behavior of the developers.

The exposed above is illustrated in the Figure 3.4, where the data sources and the obtained graphs are clearly identifiable. It should be pointed that for the sake of this work, eleven popular open source applications were studied, including Firefox, Blender, VLC, etc. Common characteristics of them are that they have a long release history, a considerable size, a large number of users reporting bugs and a large number of developers solving and patching them.

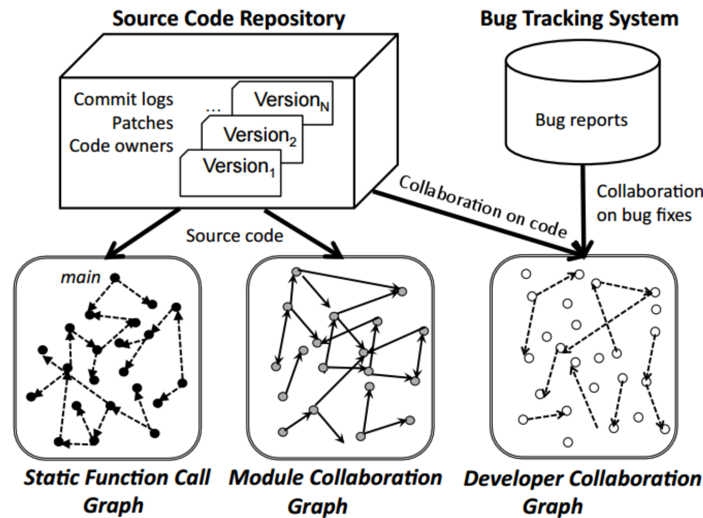


Figure 3.4: Overview of the system developed by [BINF12]

The chosen metrics as well as the software engineering concepts employed by the users were:

- Average degree
- Clustering coefficient
- Node rank
- Graph diameter
- Assortativity

The metrics mentioned above provided an overview of a single software release, in order to find out how a program structure changes over time further metrics were necessary. These metrics basically captured the number of changes in edges between two graphs and they were the edit distance and the modularity ratio. The researchers proceeded then to calculate all the selected metrics for the first and last releases of all of the systems they were studying and proceeded to analyze them to find out if they really can detect non-obvious pivotal moments in a program evolution and if there were evolution trends across all the programs they studied. After analyzing the results, they concluded that graph metrics are good measures that can reveal events in software evolution.

In order to accomplish the bug severity objective of this work, the authors proceeded to use the metric Node Rank to identify critical functions and modules that when buggy are most likely to span high-severity bugs. The hypothesis they managed to prove were:

- Functions and modules of higher Node Rank will be prone to bugs of higher severity.
- Modules with higher Modularity Ratio have lower associated maintenance effort.

- An increase in edit distance in bug-based developer-collaboration graphs will result in an increase in defect count.

At the end of this work, the researchers concluded that graph-based metrics effectively can capture and describe the structure and evolution of software products and processes. Furthermore, they can also help to predict bug severity and identify components that need higher effort in order to reduce the number of bugs of high severity of future releases.

The Graph-Based Analysis conducted in this research provides support when designing the architecture of a software system, similar to the approach followed by this thesis, even more considering that the distribution topologies are in the end directed graphs. Nevertheless, the Graph-Based Analysis is targeted to the structures used to implement the software of certain system and not to give support when designing the distribution of components, unlike this thesis aims.

3.5 Evolutionary Algorithm approach for for the Discovery of Software Architectures

3.5.1 Purpose

Since in the last few years the combination of meta-heuristic approaches and software engineering have converged into a problem domain called Search Based Software Engineering (SBSE) and being Evolutionary Computation (EC) the most widely used heuristic in the field of assisting software engineers in the improvement of their architectural designs, the authors of this work propose the use of EC as a search technique to extract the underlying software architecture of a system [RRV15].

This research work aimed to answer the following questions:

- Can a single Evolutionary Algorithm (EA) help the software engineer to identify an initial candidate architecture of a system at a high level of abstraction?
- How does the configuration of the algorithm influence both the evolutionary performance and the quality of the returned solution?

As determined by the authors of this work, the identification of architectural models is required during early stages of software conception, when requirements are still changing or the developers are asked to check the correctness of their designs. At this stage, source code artifacts are still not available, therefore the use of other sources of information to discover the intended architecture is necessary and an initial class diagram may be a good starting point for the architecture discovery.

3.5.2 Approach and Results

The approach proposed by the authors consists on taking class diagrams of a system as the artifacts to abstract the software architecture and encode them using a tree structure. Once that is done, apply a genetic operator and use definitions like cohesion and coupling to guide and validate the search.

One crucial aspect of using evolutionary algorithms is the representation of the input information, in this case the authors proposed the use of a tree structure in order to represent classes and their relationships. To construct the tree some definitions were introduced: A component is a cohesive group of classes, a directed relationship between classes belonging to different models are considered as a candidate interface and a connector is the linkage between a pair of interfaces interconnecting different components.

Figure 3.5 shows an hypothetical example of how to represent classes and components in a tree structure.

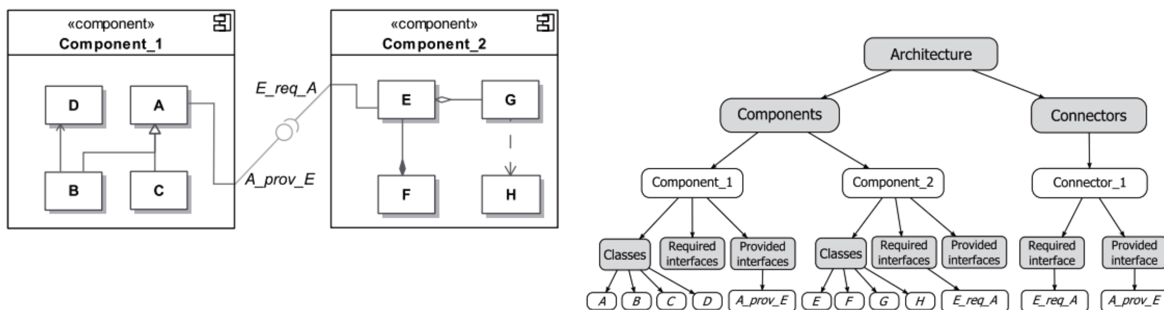


Figure 3.5: Sample mapping between classes to a tree structure, as suggested by [RRV15]

Two important aspects were the initial population and the fitness function. The initial population corresponded to the early versions of the class diagram of the system, while for the fitness function the strength and independence of the inner functionality of each component was considered, thus the fitness function was calculated as an aggregation of rankings. These rankings belonged to a specific metric related to desirable characteristics of the architectural design. Among those metrics were:

- Intra-modular coupling density, which helped to determinate trade-offs between cohesion and coupling.
- External ranking penalty, that basically applies a penalty if some relations are not specified through interfaces. The optimal value of this metric was 0.
- Groups/component ratio, that specifies the ratio between the elements of a component and the total number of components in the whole architecture.

The fitness function was defined as the aggregation of those metrics, being the lowest values the best ones.

Genetic operators were also defined, they allowed the creation of new solutions from other ones, they facilitate the tasks of exploring design alternatives. In this case, five mutations were considered and they resembled the transformation that software architects do during the discovery process. The operations were add, split, remove a component, merge two components and move a class. The definition of those genetic operators allowed to define a mutation operator, which consisted of a probabilistic roulette built for each parent and composed of the mutation procedures that can be applied to them. Once the roulette is completed, a mutation procedure is selected and if the resulting individual does not satisfy the constraints, a new generation is produced until a valid individual is obtained. All the steps previously described led finally to the definition of the EA.

The authors conducted several experiments and defined some tests to define how accurate their approach was. The results showed that this attempt was relatively good in the task of discovering good solutions in the majority of the problem instances they used to experiment. Nevertheless, they also evidenced some contradictions between metrics, specially for systems of high complexity but they considered that the results were acceptable in general, since the solutions obtained were composed of well-connected groups of classes and components that correctly matched the intended architecture, even though the result may not be the optimal one. As final toughs, the researchers considered that this approach can be extended to other scenarios, such as service oriented architectures, by including in the models factors like cost and response time and that the inclusion of opinions from experts in the evolutionary search may improve results.

This approach based on evolutionary algorithms is focused to discover software architectures, it assists developers when designing the architecture of a system in a similar way as the present thesis attempts. However, the support provided by this work is limited to software structures and not to distribution of components in the cloud.

4 Concept and Specification

The main focus of this chapter is to present the methodology and conceptual foundations for a system that allows the capturing and usage of knowledge of evolving cloud application topologies. The specification of the system is described and the concept of its design is explained.

4.1 Methodology

The usage of case-based reasoning techniques in the design and execution of cloud applications enables the possibility for handling existing problems and solving new ones with similar characteristics. The purpose of this work is to create a knowledge base by gathering and collecting data from applications that are running in the cloud and employing the CBR approach in a system that assists the application architects and developers when selecting viable distribution alternatives for a new application.

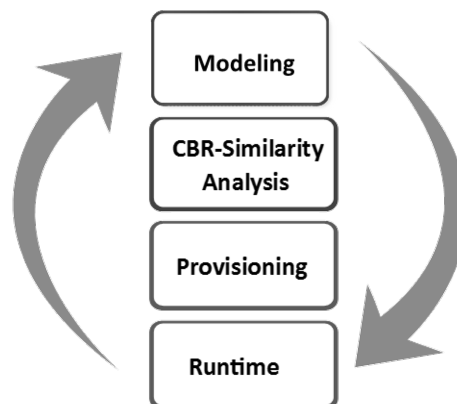


Figure 4.1: Cloud application topology enhanced life cycle - CBR Analysis

During the early stages of a system distribution design, a developer possesses a set of functional requirements as well as non-functional requirements, e.g. performance needs, business rules and workload characteristics, of an application system. The developer can use all this information and input it into the proposed system in order to get similar cases from a knowledge base as well as their distributions topologies. Based on the results, the developer can pick one of the past solutions and adapt it to the current situation. Moreover, the adopted topology is recorded in the knowledge base and its performance in runtime is monitored, in this way the case base is enriched with more solutions every time the system is used and more accurate characteristics describing the past applications are obtained.

The actions described above can be expressed in terms of different stages which are depicted in the Figure 4.1. These stages represent the step-by-step process that our methodology follows. It starts with the modeling of the topology, followed by the similarity analysis and the provisioning of resources and runtime phase. Moreover, these phases can be modeled as a cycle since the needs and requirements of applications that are running may change and therefore a new distribution of their elements might be necessary. Each of the stages of the life cycle has different processes. they are presented in the Figure 4.2 and explained in detail in the sections bellow.

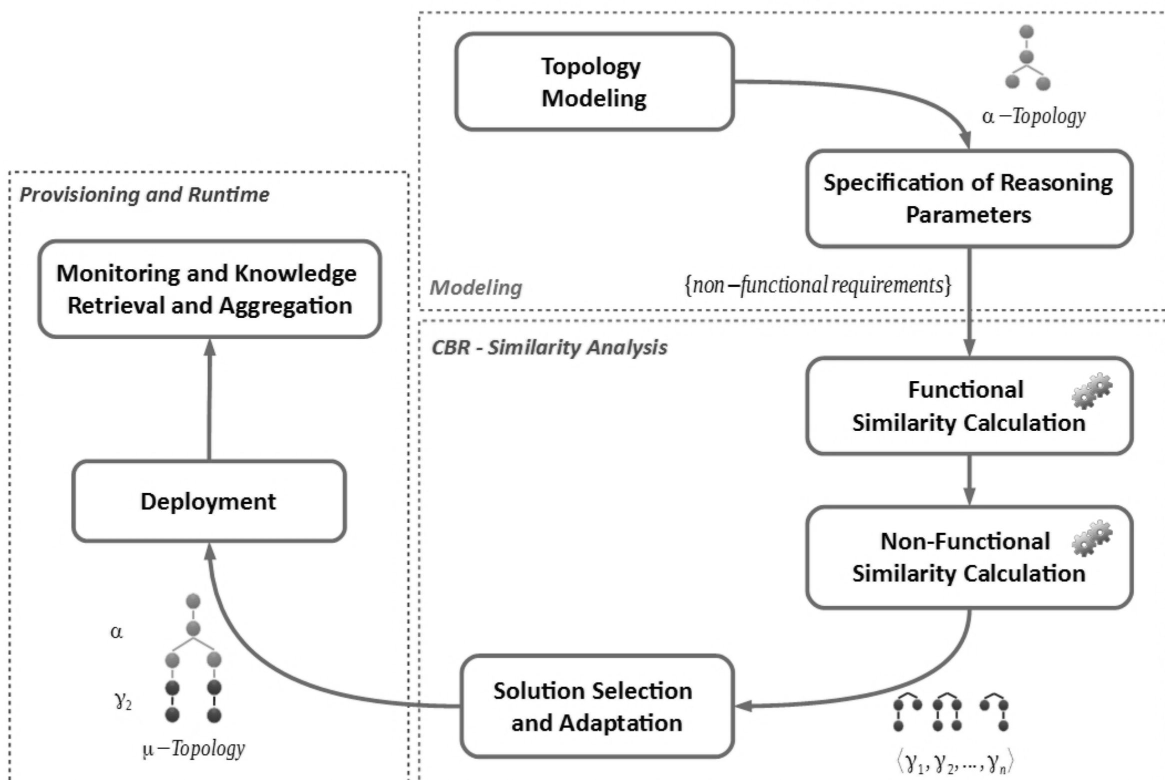


Figure 4.2: Processes of the different stages of the methodology [AGSLW14]

4.1.1 Topology Modeling - Collection Functional Requirements Data

The first stage of the methodology consists of gathering data from running systems in order to conform the knowledge base of the system. The problems are described through a series of features that include performance metrics and workload characteristics, meanwhile the solutions correspond to the distribution of components of the application that produce those QoS values, in other words the μ-topology.

Recalling the cycle of the CBR paradigm presented Section 2.3.2, one of its important aspects is the *Retain* process, which consisted on retaining the characteristics of the new problem and its solution and that is the reason why data is collected after the selected topology is running. This allows to enrich the case base and therefore the quality of the results. Furthermore,

obtaining metrics that describe the behavior of the deployed application allows to evaluate its performance and to improve the quality of the features describing a certain problem.

4.1.2 Non Functional Requirements - Data Collection

Case Representation

In this work, a *case* corresponds to a cloud application and its distribution. Its *characterization* is given by a set of different features, which can be of one of the types presented bellow:

- Functional requirements
- Non-functional requirements

Those characteristics are the ones that allow to describe the past cases and the desired features of the application to be deployed, they are also used to perform similarity analysis and retrieve possible solutions for the new case. A viable distribution topology, depicted through a μ-topology, representing the distribution of components of some application corresponds to the *lesson* or solution and they are retrieved if the features of a given problem match the ones of the application it represents. Figure 4.3 shows a diagram of the structure of the case representation in the proposed approach.

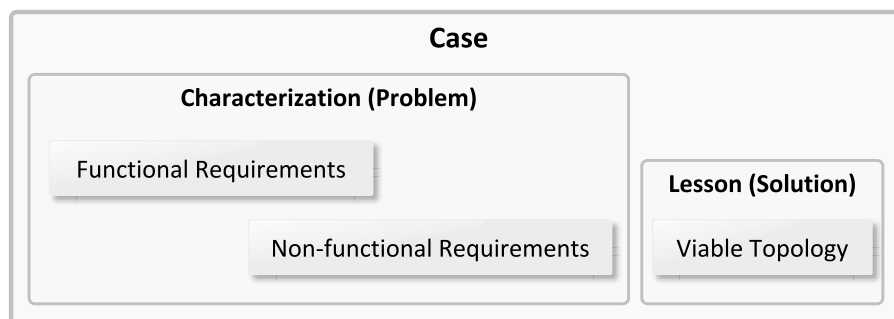


Figure 4.3: Representation of a Case in the proposed system.

As previously presented, different sets of characteristics have to be considered in order to describe an application. Following this, the data model depicted in the Figure 4.4 is proposed.

The *functional characteristics* are typically depicted through the specification of the cloud application's topology. Its different nodes and their type and the relations among them are the elements describing the main functional features that a concrete application has. In the case of the *Non-functional characteristics* different aspects should be considered, these characteristics describe the workload and the performance of the application and they are presented in the remaining of this section.

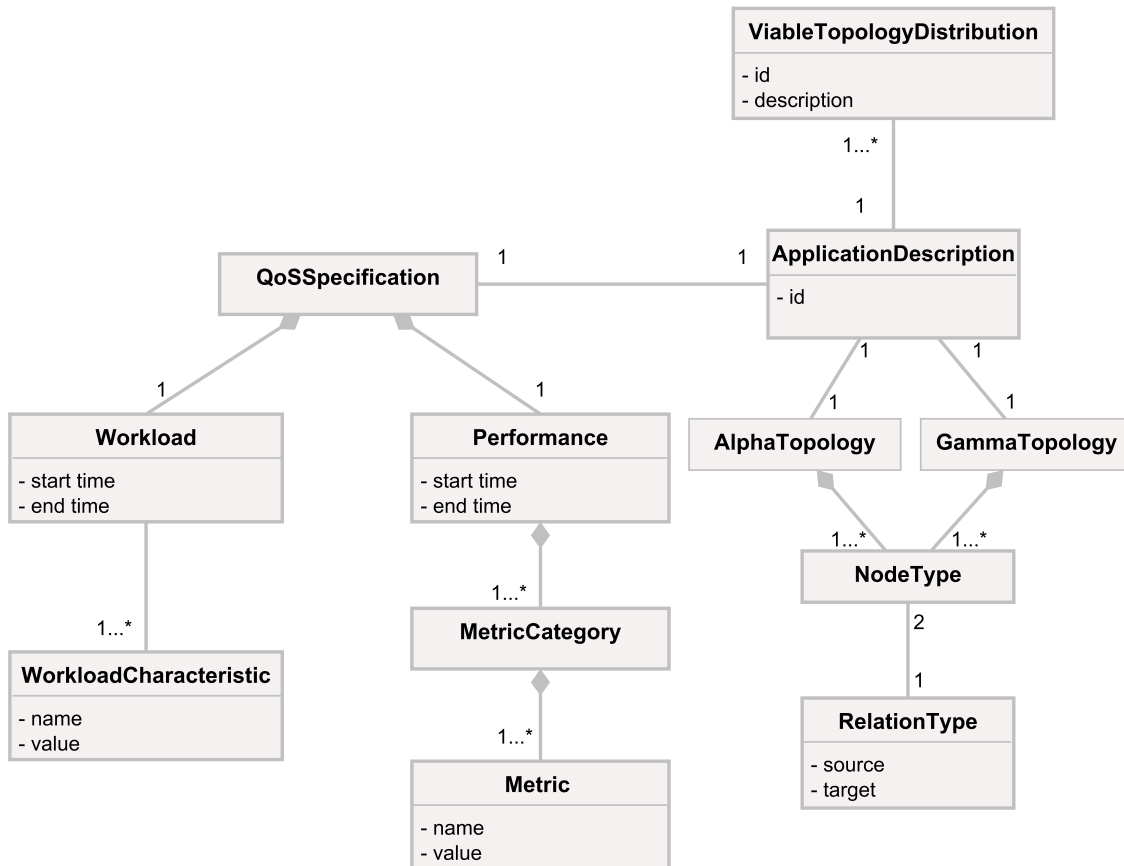


Figure 4.4: Data model of the description of an application

QoS Performance Categories and Metrics

There is a number of different metrics describing the performance aspects of an application and they depend on its type. According to the works conducted by [CS09] and [Gan15], the performance specification can be described in terms of multiple performance metrics categories and at the same time those categories contain a number of different performance metrics. Furthermore, a *performance metric* that belongs to a *category* contains different performance metric values, a monitoring resource, several analytical indicators and a threshold limiting the value range of the metric.

The analytical indicators provide information about different index attributes and their values. An index has a name and a measurement unit e.g. *measurement accuracy*, *measurement time interval*, *level of confidence*, etc. which provide different informations regarding the measurement of the metrics [ADC10, FRL13].

For the purposes of this study only statistical analytical indicators are employed to describe the different metrics. The indicators to be considered when describing performance characteristics of a case are:

- Minimum and maximum values
- Mean value
- Standard deviation value

[Gan15] also proposed a taxonomy for performance metrics categories and performance metrics that are important to consider. From this taxonomy, only the ones that are applicable to cloud applications and that we consider relevant to perform the assessment of similarity were extracted. The units of each one of them were also defined and they are shown in the Table 4.1.

Metric Category	Metric	Unit
Time Behaviour	Response Time	Milliseconds
	Throughput	Requests per second
	Processing Time	Milliseconds
	Read Speed	Revolutions per minute
	Write Speed	Revolutions per minute
	Resource Migration Time	Seconds
	Latency	Milliseconds
	Backup Time	Seconds
Capacity	Bandwidth	Megabits per Second
	Processor Speed	Gigahertz
	Storage Size	Gigabyte
	Memory Allocation to VM	Gigabyte
	Number of VM	Integer value
	Number of Processors	Integer value
	I/O Operations	Real value
Resource Utilization	Network Utilization	Percentage
	Memory Utilization	Percentage
	Disk Utilization	Percentage
	CPU Utilization	Percentage
	VM Utilization	Percentage
	Number of VM per Physical Server	Integer value
Scalability and Elasticity	Resource Acquisition Time	Seconds
	Resource Provisioning Time	Seconds
	Deployment Time	Seconds
	Resource Release Time	Seconds
	VM Startup Time	Seconds
Availability	Cloud Service Uptime	Percentage
	Cloud Resources Uptime	Percentage
	Mean Time Between Failures	Hours
	Mean Time to Repair	Hours

Table 4.1: Application QoS performance metrics per category

Workload Characteristics

[CS93] defines the workload of a system as the set of all input service requests to the system. There are different techniques that can be used to analyze the workload characteristics in terms of workloads parameters. According to [Gan15] there are two main types of such techniques:

- *Static techniques*: They characterize features of workload that do not change over time. Some examples are:
  - Statistical description
  - Single-parameter histogram
  - Multi-parameter histogram
  - Principal component analysis
  - Clustering technique
- *Dynamic techniques*: They characterize dynamic behavior of workload that changes over time. Among these techniques are:
  - Markov model:
  - User-behaviour graph:

Furthermore, there are applications whose resource usage over a period of time follows some patterns also known as usage pattern. According to [FLR+14], the resource usage pattern can be classified into five categories:

1. *Continuously Changing Workload:* This pattern is used to group workloads that show a continuous growth or decline.
2. *Once-in-a-lifetime Workload:* This type of pattern suggests that IT resources are equally used over the time, but strong peaks can occur and they happen once in a long time frame.
3. *Periodic Workload:* Peaks of utilization of IT resources happen in a periodically fashion.
4. *Static Workload:* In this pattern, the utilization of IT resources happens in a approximately flat manner over the time, this means that the use of resources is relatively constant.
5. *Unpredictable Workload:* As the name suggests, the use of IT resources cannot be predicted or approximated to some behavior.

For the purposes of this work, only a set of attributes describing the workload behavior of a system and the pattern characterization are considered, such attributes are presented in the Table 4.2.

Attribute	Values
Pattern	Continuously changing, once in a lifetime, periodic, static, unpredictable
Arrival Rate Distribution	Normal, logarithmic, gamma, uniform
Behavioral Model	Normal, logarithmic, gamma, uniform
Average Number of Users	Integer value
Average number of transactions	Integer value
Time Interval	Interval value

Table 4.2: Attribute-based characterization of the Workload of an application

4.1.3 Similarity Measures Calculation

This stage of the methodology is without doubts the most important. As it has been presented in previous sections of this work, a not appropriate definition of similarity measures will lead to a retrieval of cases that are no useful for a determined situation. Therefore, this section presents the procedures that are followed to calculate the similarity between two cases, considering that their structure not only consists of a big set of attributes as presented in Section 4.1.2.

There are a series of elements that need to be considered when depicting the requirements of a determined system. One of them correspond to the description of functional requirements, which are given through the $\alpha-$topology of the application and since it can be represented as a labeled graph, the use of graph similarity makes sense. This technique is used to compare the given $\alpha-$topology with the $\alpha-$topologies of the cases contained in the knowledge base and retrieve a subset of viable topologies that are in some grade similar to the current case.

The described above is not enough to claim that the obtained subset contains a proper solution, but the use of non-functional requirements allows to further refine the results obtained at this point. As presented Section 4.1.2, the non-functional characteristics are given in this proposal on terms of a number of performance metrics and workload characteristics, which can be expressed in most cases as attribute-value pairs and therefore a number of different similarity measures can be applied in this situation.

The selection of the similarity measures depends basically on the nature of the values of the different attributes, for example similarity measures that use numeric values cannot be applied to other types. Calculations of similarity are performed between each attribute of the current case and the ones describing a past case, these calculations correspond to the measurement of local similarities and per each attribute a value of similarity is obtained as result. It is necessary now to perform an aggregation of all the obtained values by performing a global similarity calculation and after this process, single values of similarity are obtained per each case and the ones with higher values correspond to the most similar.

This process allows to extract once more a subset of the elements obtained when performing the graph similarity. Figure 4.5 presents a summarized view of the different steps executed when calculating the similarity between two cases.

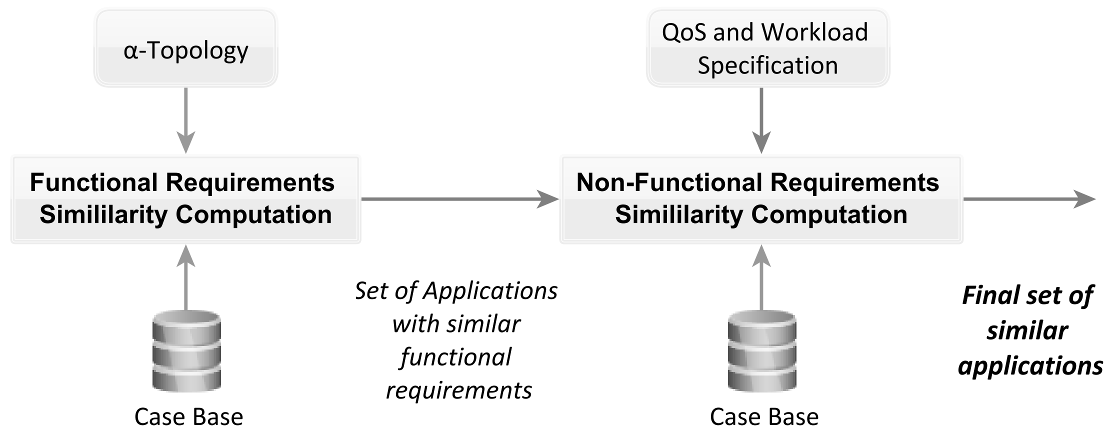


Figure 4.5: Similarity calculation process.

4.1.4 Solution Selection and Adaptation

As a result of the application of the similarity analysis several topologies are obtained. In this stage, the developer makes a selection a topology distribution, which basically is the one that meets the desired requirements. The selected topology may not be identically used and it might require adaptations in order to effectively fulfill all the needs and requirements.

The actions previously described correspond to the *revise* process of the CBR cycle. The *retain* process is also performed in this stage since the changes executed by the application architects are going to produce a new solution, and retaining this new solution allows the enrichment of the case base and a possible improvement of the quality of the results.

4.1.5 Deployment, Monitoring and Knowledge Retrieval and Aggregation

Given a determinate system, it will be ideally deployed according to the topology previously obtained. Such a system is going to be running and its performance can be observed. This information is very important because it allows to the developer to evaluate if the adopted distribution effectively meets all the needs and requirements. If the application is performing in a non-desired way, it will conduct to the developer to perform changes in the topology naturally.

Such changes and actions have a high relevancy and it is necessary to capture them in the case base, because of that this part of the methodology aims to capture those changes, the metrics describing the QoS of the modified application and it that way have a record of the

evolution of the performance of the system and use that information to possibly provide more accurate results.

4.2 Formalizing Similarity

This section presents the approaches and models employed to conduct the similarity analysis. Since the chosen attributes have different nature and types, it is necessary to use the appropriate similarity measures, otherwise the results returned by te system might not be accurate enough. The sections bellow present in detail the different approaches employed to determine whether two applications are similar or not.

4.2.1 Similarity of Functional Requirements

As introduced in Section 4.1.1, the functional requirements of an application are represented through the specification of its topology, which implies the definition of the different nodes types and their relationships.

In the case base, the functional characteristics of the different solutions are depicted in terms of their μ-topology, which is composed of α- and γ-topology. Therefore, two aspects must be considered when deciding if two application have comparable functional requirements.

- Structure of the $\alpha-$topology
- The node type definition

The similarity among two topologies can be analyzed through the usage of graph similarity techniques. This approach lets us determinate if the application that is being analyzed and the topology of another application have the same distribution or not. For example, if the developer has depicted a two-tier application as its $\alpha-$topology, through the execution of graph similarity is possible to discard all the cases that correspond to three-tiers applications and only consider the ones that fit a two-tier structure.

Further filtering consists of determining whether the node types conforming the selected topologies are similar or not. This is not an easy task, since there are plenty of services in the market and new ones are launched everyday and there are not means to effectively compare them. Trying to answer the question *how similar are Ubuntu Server and 10.0 and Windows Server 2012?* is an example of the previously exposed. There might be ways to do this for example by comparing their characteristics, the programming languages they support, through the evaluation of other developers, etc., but there is not a single way to approach them to a particular model, given the huge amount of services and the aspects that should be considered.

In order to mitigate this the approach of similarity tables is adopted. At the beginning the table will contain values of 1 and 0, following the *Is equal?* similarity model, which indicates that if two node types are equal the similarity value is 1 and 0 otherwise. Nevertheless, the

	Web Shop	e-Commerce	Rich Internet	Mobile Backend	Customer Relationship	News Feed	Email
Web Shop	1	0	0	0	0	0	0
e-Commerce	0	1	0	0	0	0	0
Rich Internet	0	0	1	0	0	0	0
Mobile Backend	0	0	0	1	0	0	0
Customer Relationship	0	0	0	0	1	0	0
News Feed	0	0	0	0	0	1	0
Email	0	0	0	0	0	0	1

Table 4.3: Sample of a similarity table for *type of application*

developed system has to provide the means for experts to modify the values of similarity according to their knowledge and in this way the similarity measures can be enriched and the model could reach a knowledge-intensive status.

After applying the approach of similarity table, a similarity value is obtained per each node of the topology. These values represent a set of of local similarities, to obtain an unique result representing the global similarity the Minkowski aggregation model, presented in Section 2.4.2, is applied with equal weight value for all the local similarity measures and with the parameter $p = 1$ as recommended by [MC11] in these cases. Table 4.3 presents an example of a similarity table that follows the approach described previously, in this case the compared node types correspond to the type of application. The names of the different columns and rows correspond to examples of application types, each cell represent the similarity value among two types. When comparing two nodes, one of type *Web Shop* and other of type *Mobile Backend*, it is enough just to look for the correspondent column and row to find out the value, in this case is 0; if both of them were of the same type then the result would have been 1.

Algorithm number 1 illustrates the process of the calculation of similarity of node types between a query and a case of the knowledge base. In order to obtain an accurate result is assumed that the query and case received by the function have a similar structure and their comparison can be performed in a sequential order.

Algorithm 1 Node types similarity calculation

```
1: function CALCULATENODETYPEPERFORMANCESIMILARITY(queried_app,case)
2:     queryNodeTypes ← GETNODETYPES(queried_app)
3:     caseNodeTypes ← GETNODETYPES(case)
4:     similarityTable ← GETSIMILARITYTABLE()
5:     localSimilarityList ← CREATELIST
6:     for i ← 0, SIZE(queryNodeTypes) do
7:         queryNode ← GETNODE(queryNodeTypes, i)
8:         caseNode ← GETNODE(caseNodeTypes, i)
9:         localSimilarity ← RETRIEVESIMILARITY(similarityTable, queryNode, caseNode)
10:        ADD(localSimilarityList, localSimilarity)
11:    end for
12:    nodeTypeSimilarity ← COMPUTEMINKOWSKIAGGREGATION(localSimilarityList)
13:    return nodeTypeSimilarity
14: end function
```

4.2.2 Similarity of Performance Metrics

As presented in Section 4.1.2, the different metrics describing performance characteristics of an application are expressed in terms of statistical analytical indicators and in this work the ones considered are:

- Minimum and maximum values
- Mean value
- Standard deviation value

Contrasting with the previous problem of comparing node types, in this case the computation of similarity is easier since the mentioned indicators are numeric and continue values. The local similarity principle is used to calculate the similarities among the statistical descriptors and the global similarity principle is used to aggregate the local similarity values and get an unique value per each metric.

Since a similarity value per each metric is obtained, once again the global similarity principle is applied in order to obtain a measure depicting the degree of similarity between the descriptions provided by the application developer and the cases stored in the knowledge base.

To calculate the local similarities, a *difference-based* function is used, that indicates that the bigger the absolute difference between two values, the smaller the value of the similarity. The base function used to compute the difference-based local similarity is given by the exponential model presented in 4.1.

$$SIM(X,Y) = e^{|\delta(X,Y)|.\alpha} \tag{4.1}$$

Where $\alpha = -1$ and $\delta(X,Y)$ is a difference function and can take the following values:

$$\delta(X,Y) = \begin{cases} ln(X) - ln(Y) & \text{for } X, Y \epsilon \mathbb{R}_+ \\ -ln(-X) + ln(-Y) & \text{for } X, Y \epsilon \mathbb{R}_- \\ X - Y & \text{if X=0 or Y=0} \\ undefined & \text{if any of the values is not specified} \end{cases}$$

If the difference function $\delta(X,Y)$ takes the special value of *undefined* because at least one of the measures is not specified, then that parameter is not considered when computing the global similarity. Furthermore, the exponential base function was chosen because it offers better results for both continuous and discrete values [Sta03] and the statistical descriptors here employed as attributes could be of any of those types.

The value of the global similarity is calculated as in the case of the functional requirements, using the Minkowski aggregation model. The algorithm 2 presents how the similarity calculation process is performed given a new query and a retrieved case from the knowledge base.

Algorithm 2 Performance metrics similarity calculation.

```
1: function CalculatePerformanceSimilarity(queried_app,case)
2:     queryMetrics ← getPerformanceMetrics(queried_app)
3:     globalSimilarityList ← createList
4:     for each metric ϵ queryMetrics do
5:         queryMetricDescriptors ← getDescriptors(metric)
6:         caseMetric ← getMetric(case, metric)
7:         localSimilarityList ← createList
8:         for each descriptor ϵ queryMetricDescriptors do
9:             caseMetricDescriptor ← getDescriptor(caseMetric, descriptor)
10:            difference ← computeDifference(descriptor, caseMetricDescriptor)
11:            localSimilarity ← e^(abs(difference) * −1)
12:            add(localSimilarityList, localSimilarity)
13:        end for
14:        globalSimilarityMetric ← ComputeMinkowskiAggregation(localSimilarityList)
15:        add(globalSimilarityList, globalSimilarityMetric)
16:    end for
17:    performanceSimilarity ← ComputeMinkowskiAggregation(globalSimilarityList)
18:    return performanceSimilarity
19: end function
```

4.2.3 Similarity of Workload Characteristics

Table 4.2 presented the list of characteristics that the present approach employs to describe the workload of an application. The difference of these attributes and the ones describing the performance requirements lies in the fact that they can be of numeric or categorical nature. The procedure to calculate the similarity between two sets of workloads characteristics, consists of the use of similarity tables for attributes of categorical type e.g. pattern, arrival rate distribution and difference-based function for the numerical attributes e.g. average number of users, average number of transactions per second. The similarity tables use also the *Is Equal?* model, and the system will provide means to developers to update the values and add further categories, similarly to the case of similarity tables for node types.

In other words the approach is a combination of the previous cases, employing a similarity model based on the type of the attribute to calculate local similarities and aggregating them through the use of the Minkowski aggregation model. Algorithm 3 presents the process followed to make the computation of the similarity in this case.

Algorithm 3 Workload characteristics similarity calculation.

```
1:  function CalculateWorkloadSimilarity(queried_app,case)
2:      queryWorkloadAtts ← getWorkloadAttributes(queried_app)
3:      localSimilarityList ← createList
4:      for each queryAttribute ϵ queryWorkloadAtts do
5:          queryMetricDescriptors ← getDescriptors(metric)
6:          caseAttribute ← getAttribute(case, queryAttribute)
7:          attributeType ← getAttributeType(queryAttribute)
8:          if attributeType = numeric then
9:              difference ← computeDifference(queryAttribute, caseAttribute)
10:             localSimilarity ← e^(abs(difference) * −1)
11:         else
12:             similarityTable ← getSimilarityTable(queryAttribute)
13:             localSimilarity ← retrieveSimilarity(similarityTable, queryAttribute, caseAttribute)
14:         end if
15:         add(localSimilarityList, localSimilarity)
16:     end for
17:     workloadSimilarity ← ComputeMinkowskiAggregation(localSimilarityList)
18:     return workloadSimilarity
19: end function
```

4.2.4 Application Similarity Calculation

In order to get a single value of similarity and finally determinate if two applications are similar or not, the global similarity principle must be applied once more to the measures previously obtained. The Minkowski aggregation model is applied to the similarity values of functional characteristics, performance and workload, in this way an unique measure

between a query and a case is obtained. Algorithm number 4 illustrates how the process of computation is executed.

Algorithm 4 Application similarity calculation.

```
1:  function CalculateApplicationSimilarity(queried_app,case)
2:      functionalSimilarity ← CalculateNodeTypeSimilarity(queried_app, case)
3:      performanceSimilarity ← CalculatePerformanceSimilarity(queried_app, case)
4:      workloadSimilarity ← CalculateWorkloadSimilarity(queried_app, case)
5:      localSimilarityList ← createList
6:      add(localSimilarityList, functionalSimilarity)
7:      add(localSimilarityList, performanceSimilarity)
8:      add(localSimilarityList, workloadSimilarity)
9:      applicationSimilarity ← ComputeMinkowskiAggregation(localSimilarityList)
10:     return applicationSimilarity
11: end function
```

4.3 System Requirements

The CBR system for retrieval of cloud applications viable topologies, should provide ways to offer the features presented bellow:

4.3.1 Functional Requirements

- *Knowledge Base:* The system requires a properly structured knowledge base containing meaningful description of past cases and their respective solution.
- *Specification of functional characteristics:* The system should provide means to allow the developer to specify the functional requirements of the application he is intending to deploy in the cloud. This specification should be done by allowing the developer to model the α-topology through some topology modeling tool.
- *Specification of non-functional characteristics :* The system should provide means to allow the developer to specify non-functional requirements of the application he is intending to deploy in the cloud. This specification should be done by allowing the developer to input to the system characteristics of the required QoS, through the description of performance metrics and workloads characteristics.
- *Retrieval of similar applications and their topologies:* Given the description of functional, non-functional characteristics and constraints, the system should retrieve a list of applications that have a high degree of similarity with the problem being described and the respective solutions, which correspond to the different topology distributions.
- *Administration and Maintenance of Similarity Measures:* The system should provide means to maintain similarity measure models in order to enrich them and make them knowledge-intensive.
- *Continuous Update of Knowledge:* Changes performed to the obtained solution and the different characteristics specified by the developer should be retained and stored in the knowledge base.
- *QoS Evolution Monitoring:* Once the adopted solution has been deployed in the cloud, the system should capture any changes in the distribution and all the QoS attributes that the application had when the changes were made.

4.3.2 Non-Functional Requirements

- *Usability:* The system must allow to the users specify functional and non-functional characteristics of the system they want to deploy in an easy and interactive manner.
- *Consistency:* The operations performed by the system must always behave in the same way.
- *Compatibility:* The system should be compatible with a topology modeling tool.

- *Performance:* The system should perform efficiently and make computations in a reasonable time.
- *Documentation:* The system should provide proper descriptions of its different functions in order to facilitate the correct use of its functionalities.

4.4 Use Cases

There are two kind of actors in the system: The *application developer*, which is the main user interacting with the system also *domain experts*.

The roles of the actors interacting with the system are described in the sections bellow:

4.4.1 Application Developer

- *Describe system requirements:* The developer describes the application he is intending to deploy by depicting functional and non-functional characteristics, this means modeling the α-topology and describing the QoS requirements respectively.
- *Select a viable topology:* Since the CBR system returns a set of different topologies, the developer must select one which is the proposed solution that better fits the requirements of the application.
- *Adapt proposed viable topology:* Once a solution is picked, it might be necessary that the developer performs changes to the distribution in order to fulfill completely the requirements.
- *Persist Knowledge:* The developer can save all the description of characteristics and modifications to the topology distributions he has performed and in that way enrich and improve the quality of the knowledge base.

4.4.2 Domain Expert

In order to enrich knowledge of the CBR application and its similarity measures, the system allows to *domain experts* to perform the next actions:

- *Enrich similarity tables:* An expert can retrieve and update the different similarity tables that the system uses to get the values for local similarity of categorical attributes.
- *Update Knowledge:* An expert should be able to retrieve and update knowledge from the case base in order to keep the information as accurate as possible.

4.4.3 Use Cases Diagram

Figure 4.6: Use Cases Diagram.

The proposed system has two boundaries that group the different use cases. The modeling system boundary groups cases that can be performed through the use of a selected topology

modeling tool, the use cases that are depicted with dashed lines indicate that the use case is being adapted or extended in order to use the already implemented functions of the modeling tool in conjunction with the CBR system. Use cases drawn with a continuous line indicate that they are built from scratch.

The CBR-Similarity boundary groups the use cases that only concern to this system. The use cases diagram of the CBR system for retrieval of cloud applications viable topologies is shown in the Figure 4.6

4.4.4 Use Cases Description

Use cases that are implemented from scratch are described in details in the use case tables shown below while the ones that constitute an extension of the modeling tool are depicted in terms of a description indicating the type of adaptation performed.

Name	**Model Alpha Topology**
Goal	The user wants to provide a description of the functional requirements of an application through the model of the α-topology of an application.
Actor	Application developer
Description	User employs tools provided by a topology modeler software in order to depict the α-topology of an application, this software is able to communicate with the CBR system and send information in an understandable format for it.

Table 4.4: Description of Use Case *Model Alpha Topology*.

Name	**Specify Performance Requirements**
Goal	The user wants to provide a description of the performance requirements that the application needs to meet.
Actor	Application developer
Pre-Condition	1. The performance metrics characteristics are specified under common properties and are specific to an application.
Post-Condition	1. The performance metrics specification has been successfully added. 2. The performance metrics specification can be viewed as XML representation.

Post-Condition in Special Case	1. The performance metrics specification cannot be added.
Normal Case	1. User selects *Specify Performance Metrics*. 2. An interface that allows the input of a series of metrics and their values is rendered. 3. User fills in with the desired information. 4. User adds the specification to the system.
Special Cases	2a. Interface cannot be rendered. a) System shows an informative error message. 4a. The performance metrics specification cannot be added. a) System shows an informative error message.

Table 4.5: Description of Use Case *Specify Performance Requirements*.

Name	**Specify Workload Characteristics**
Goal	The user wants to provide a description of the workload that the application must cope with.
Actor	Application developer
Pre-Condition	1. The workload characteristics are specified under common properties and are specific to an application.
Post-Condition	1. The workload characteristics specification has been successfully added. 2. The workload characteristics specification can be viewed as XML representation.
Post-Condition in Special Case	1. The workload characteristics specification cannot be added.

Normal Case	1. User selects *Specify Workload Characteristics*. 2. An interface that allows the input of a series of attributes and their values is rendered. 3. User fills in with the desired information. 4. User adds the specification to the system.
Special Cases	2a. Interface cannot be rendered. a) System shows an informative error message. 4a. The workload characteristics specification cannot be added. a) System shows an informative error message.

Table 4.6: Description of Use Case *Specify Workload Characteristics*.

Name	**Specify Hard Constraints**
Goal	The user wants to provide a description of hard constraints that the application needs to meet.
Actor	Application developer
Pre-Condition	1. The application hard constraints are specified under common properties and are specific to an application.
Post-Condition	1. The user can use the provided information to estimate the cost of the application he is intending to distribute and is able to change it and perform new queries with it. 2. The application constraints specification can be viewed as a series of attributes and values.
Post-Condition in Special Case	-
Normal Case	1. User selects 'Distribution Cost'. 2. An interface that allows the input of a series of attributes and their values is rendered. 3. User fills in with the desired information. 4. The user is able to use the specification in order to estimate the cost of the distribution.

Special Cases	2a. Interface cannot be rendered. a) System shows an informative error message.

Table 4.7: Description of Use Case *Specify Hard Constraints*.

Name	**Calculate Distribution Cost**
Goal	The user wants to obtain an estimation of the cost of the application he is intending to deploy, given its functional, non-functional characteristics and hard constraints.
Actor	Application developer
Pre-Condition	1. The user has specified functional, non-functional characteristics and hard constraints of the application.
Post-Condition	1. The user is capable the cost of the current distribution according to all the characteristics he has provided.
Post-Condition in Special Case	1. Cost estimations cannot be retrieved.
Normal Case	1. User selects 'Distribution Cost'. 2. The modeling tool contacts a cost calculation service, sends to it the descriptions provided by the user and renders the received answer.
Special Cases	2a. Cost calculation service cannot be contacted a) System shows an informative error message. 3a. Description of the cost cannot be rendered. a) System shows an informative error message.

Table 4.8: Description of Use Case *Calculate Distribution Cost*.

Name	**View Model**
Goal	The user wants to visualize the schema of one of the proposed solutions.
Actor	Application developer
Description	User employs functionalities provided by the modeling tool to visualize the different topology distributions suggested by the CBR System.

Table 4.9: Description of Use Case *View Model*.

Name	**Refine Application**
Goal	The user modifies one of the available suggested solutions by the system in order to make it fit completely to the conditions of his problem.
Actor	Application developer
Description	User employs tools provided by the topology modeler software in order to modify a distribution returned by the CBR system and make it fit to all the requirements.

Table 4.10: Description of Use Case *Refine Application*.

Name	**Retrieve Deployment Package**
Goal	The user wants to a deployment package of the application.
Actor	Application developer
Description	User employs functionalities provided by the modeling tool to download a deployment package of the topology that meets the requirements of the application he is intending to deploy.

Table 4.11: Description of Use Case *Retrieve Deployment Package*.

Name	**Discover Similar Applications**
Goal	The user wants to retrieve a list of applications and their solutions that could fulfill the specified set of requirements. The user also wants to see indicators that inform him about how similar his application and the ones in the knowledge base are.
Actor	Application developer

Pre-Condition	1. The users has specified a set of functional and/or non-functional characteristics of an application and they are successfully received by the CBR engine.
Post-Condition	1. A list of similar applications, similarity measures and possible topologies solutions are displayed to the user.
Post-Condition in Special Case	1. No results are shown.
Normal Case	1. User selects 'Discover Similar Applications'. 2. The modeling tool contacts the similarity engine and sends all the functional and non-functional characteristics the user has specified. 3. The CBR engine receives the information, processes it and obtains a list of similar problems from the knowledge base. 4. Once a list of similar application is obtained, the CBR engine retrieves their respective viable topology distributions and sends them to the modeling tool. 5. The modeling tool renders the answer sent by the CBR engine and the user obtains a list of possible solutions for the problem he has described and he can inspect all of them and check the value of the similarity measure.
Special Cases	2a. The CBR Engine cannot be reached. a) System shows an informative error message. 3a. The CBR Engine cannot retrieve the solutions of the applications. a) System shows an informative error message. 4a. Answer of the CBR cannot be rendered. a) System shows an informative error message.

Table 4.12: Description of Use Case *Discover Similar Applications*.

Name	**Compute Similarity**
Goal	In order to discover similar applications, the computation of different similarity measures of the characteristics of a given problem and the ones stored in a knowledge base should be performed
Actor	Application developer

Pre-Condition	1. The users has specified a set of functional and/or non-functional characteristics of an application and they are successfully received by the CBR engine.
Post-Condition	1. Similarity measures of non-functional characteristics, per each similar application are returned.
Post-Condition in Special Case	1. Similarity measure computation cannot be performed.
Normal Case	1. The CBR engine contacts a system where different viable topologies are stored and sends to it the functional characteristics of an application provided by the user. 2. A list of applications with similar functional characteristics is obtained. 3. Non-functional characteristics of the applications previously obtained are retrieved and computations are performed in order to obtain similarity measures 4. A list of applications with the respective similarity values is returned.
Special Cases	1a. Is not possible to reach the system containing functional requirements description of applications. a) System shows an informative error message. 3a. The Similarity measures cannot be obtained. a) System shows an informative error message.

Table 4.13: Description of Use Case *Compute Similarity*.

Name	**Retrieve Viable Distribution**
Goal	Given an application the retrieval of its viable topology distributions needs to be performed
Actor	Application developer
Pre-Condition	
Post-Condition	1. Information describing a viable topology distribution is returned.

Post-Condition in Special Case	1. Retrieval cannot be performed.
Normal Case	1. The CBR engine queries the knowledge base in order to get descriptions of the solutions of a determinate application.
Special Cases	1a. Information cannot be retrieved. a) System shows an informative error message.

Table 4.14: Description of Use Case *Retrieve Viable Distribution*.

Name	**Store Adapted Solution**
Goal	The user wants to store in the CBR case base the solution he has adapted and its characteristics.
Actor	Application developer
Pre-Condition	1. The user has performed changes to the topology he has selected and wants to save it. along with its characteristics to the system.
Post-Condition	1. The user receives a confirmation message.
Post-Condition in Special Case	1. The topology and its characteristics cannot be saved.
Normal Case	1. User selects the option 'Save Solution to Case Base'.
Special Cases	1a. It is not possible to store the new case and its solution. returned. a) System shows an informative error message. 2a. It is not possible to continue with the next step. a) System shows an informative error message.

Table 4.15: Description of Use Case *Store Adapted Solution*.

Name	**Persist Viable Distribution**
Goal	The user wants to store in the system the *mu*−topology he has modeled in order to enrich the case base
Actor	Application developer
Pre-Condition	1. The user has redefined a proposed viable topology and has stored it in he repository employed by the modeling tool
Post-Condition	1. The user obtains a confirmation message.
Post-Condition in Special Case	1. The user is not able to persist the topology distribution.
Normal Case	1. User selects an option that allows him to persist the distribution. 2. An interface is rendered where he fills required informations. 3. User proceeds to save the changes he has performed. 4. The modeling tool sends the information to the CBR system and it is stored
Special Cases	2a. No interface is rendered. a) System shows an informative message. 4a. The modeling tool is not able to reach the CBR engine and it is not possible to store the changes. a) System shows an informative message.

Table 4.16: Description of Use Case *Persist Viable Distribution*.

Name	**Persist Knowledge**
Goal	The user wants to store in the system the characteristics of the application he has described.
Actor	Application developer
Pre-Condition	1. The user has previously persisted a viable distribution and has provided description of non-functional characteristics.

Post-Condition	1. The user obtains a confirmation message.
Post-Condition in Special Case	1. The user is not able to persist the knowledge of the application.
Normal Case	1. User selects an option that allows him to persist the knowledge of the application he has described. 2. The modeling tool sends the data to the CBR system. 3. The information is stored.
Special Cases	2a. The modeling tool is not able to reach the CBR engine and it is not possible to store the changes. a) System shows an informative message. 3a. It is not possible to store the provided information in the CBR System . a) System shows an informative message.

Table 4.17: Description of Use Case *Persist Knowledge*.

Name	**Retrieve Knowledge**
Goal	The user wants to retrieve the characteristics of a determined application.
Actor	Domain expert, Application developer
Pre-Condition	1. The user has retrieved IDs of applications that are similar to a current problem from the system.
Post-Condition	1. The user obtains the knowledge and characteristics of an applications that were used to perform the similarity analysis.
Post-Condition in Special Case	1. The user is not able to see any information.
Normal Case	1. User requests the knowledge of an specific application.

Special Cases	1a. No results are returned. a) System shows an informative message.

Table 4.18: Description of Use Case *Retrieve Knowledge*.

Name	**Update Knowledge**
Goal	The user wants to update the information of an application stored in the knowledge base.
Actor	Domain expert
Pre-Condition	1. The user has retrieved knowledge of an specific application.
Post-Condition	1. The user obtains a confirmation message.
Post-Condition in Special Case	1. The user is not able to update the data.
Normal Case	1. User requests to update the knowledge and characteristics of a determined application.
Special Cases	1a. It is not possible to delete the information. a) System shows an informative error message.

Table 4.19: Description of Use Case *Update Knowledge*.

Name	**Retrieve Similarity Tables**
Goal	The user wants to visualize the similarity tables employed by the system.
Actor	Domain expert
Pre-Condition	1. The system is up and running.
Post-Condition	1. The user obtains a list of the similarity tables used by the system.

Post-Condition in Special Case	1. The user is not able to see any similarity table.
Normal Case	1. The user retrieves the similarity tables employed by the system.
Special Cases	1a. No results are returned. a) System shows an informative message.

Table 4.20: Description of Use Case *Retrieve Similarity Tables*.

Name	**Update Similarity Table**
Goal	The user wants to add more rules to one similarity measures table.
Actor	Domain expert
Pre-Condition	1. A similarity table has been retrieved.
Post-Condition	1. Changes to the similarity table are saved.
Post-Condition in Special Case	1. Is not possible to save changes into the system.
Normal Case	1. The user makes a request to modify a similarity table. 2. An interface is rendered and the user fills in with the desired information. 3. The user saves the performed changes.
Special Cases	2a. No interface is rendered. a) System shows an informative error message. 4a. It is not possible to save changes into the system. a) System shows an informative error message.

Table 4.21: Description of Use Case *Update Similarity Table*.

4.5 System Overview

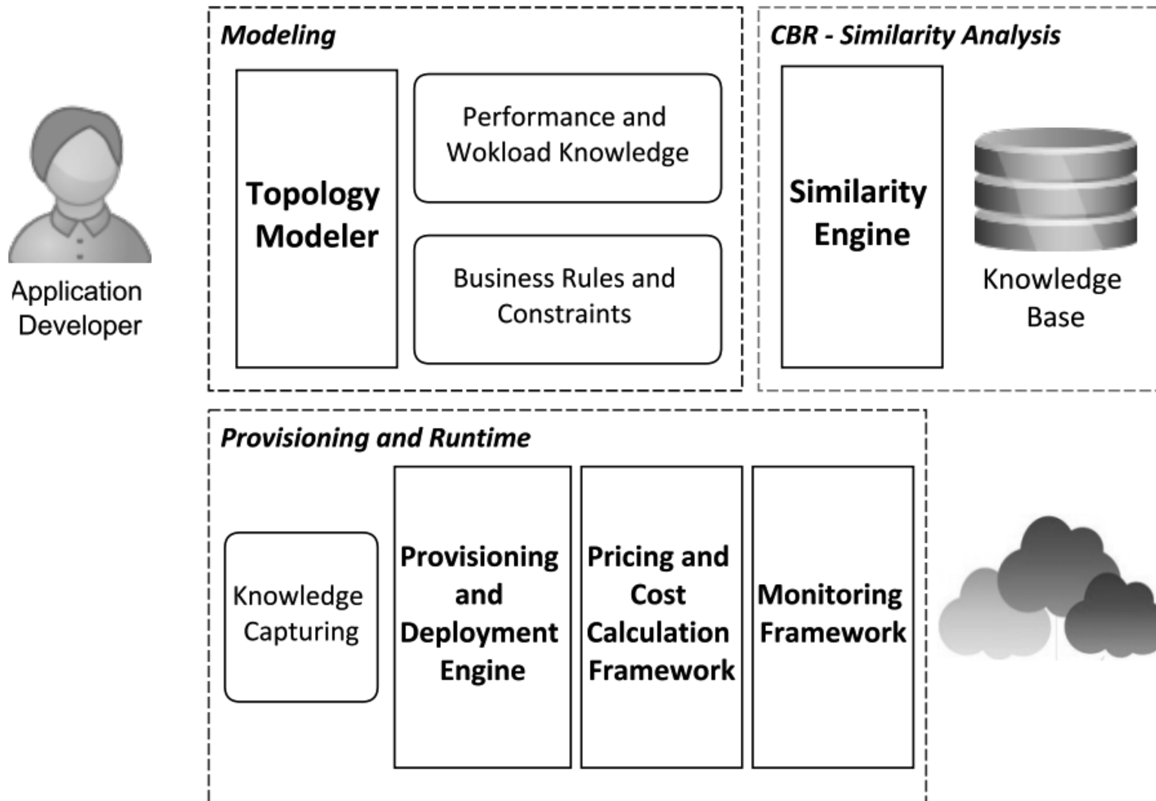


Figure 4.7: System Architecture Overview

Our methodology has several phases which require a system constituted of several environments and tooling support, as depicted in Figure 4.1 presents. An user access the system through a topology modeler tool in order to model the α-topology of the application he is intending to deploy and also making use of knowledge related to the performance and workload with the objective of providing an accurate description of the application to be deployed.

Once all the required characteristics have been collected, they are processed by a similarity engine, which is the component performing the different similarity calculations and aggregations using past cases and solutions stored in a knowledge base. This component returns a list of the possible solutions for the current problem, ranked accordingly to the results of the similarity computation.

The incurred costs of a deployed application can be estimated through the use of a pricing framework. This framework receives characteristics of the application profile as well as hard constraints to perform a cost estimation of the application running in different cloud providers.

As it has been previously mentioned, in CBR systems something really important is to

keep improving and enriching the knowledge base, because of that and once a proposed distribution topology has been used deployed with the help of a provisioning engine, a monitoring framework is in charge of collecting and recording metrics of the performance and workload of that system. Since a indefinite number of values may be captured, a knowledge capturing component is the responsible element of filtering the relevant information and recording it in the knowledge base, in this way the case base will not be populated with non-relevant data that might deteriorate the quality of results instead of improving them.

Figure 4.7 presents a schema of the different components that conform the proposed system, the elements are grouped according to the phase of the life cycle of the methodology just for illustration purposes.

5 Design

Based on the concepts and specifications proposed in the previous chapter, this one presents in detail the architecture of a system capable of capturing and using knowledge of evolving cloud application topologies through the employment of CBR techniques. First, an overview of the architecture of the framework and its different components and interfaces is provided. Subsequently, design considerations of the non-functional aspects of the data model as well as of the modeling layout are presented. The chapter finishes defining the functions that a RESTful API that allows the control of the framework via HTTP requests supports.

5.1 Architectural Overview

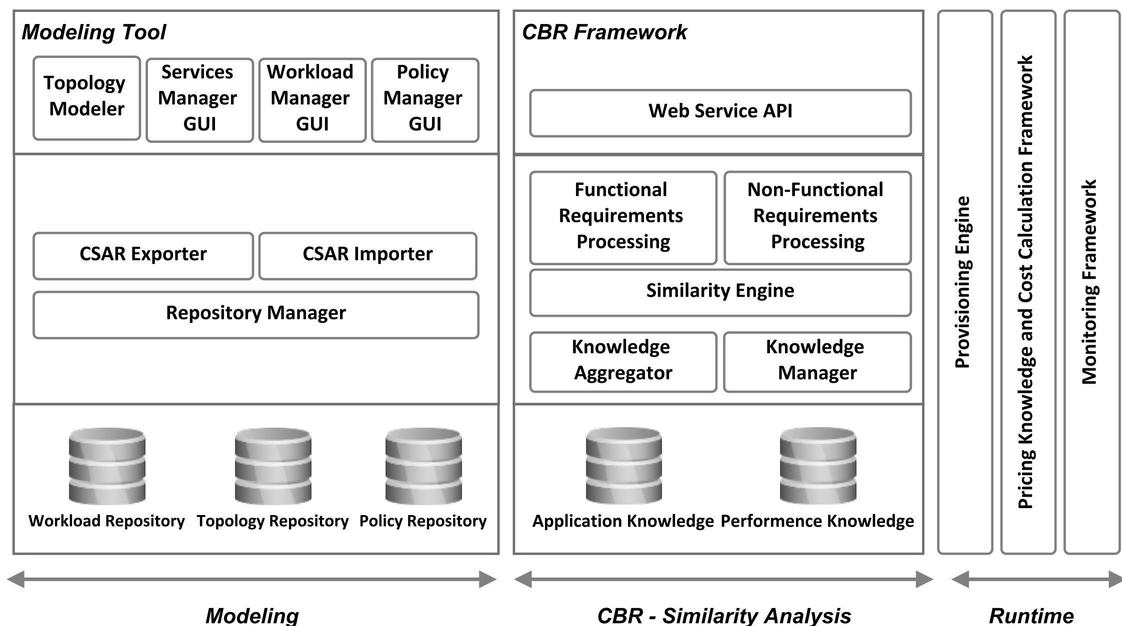


Figure 5.1: Architectural overview of the system.

The representational architectural model employed to express the system is a three-layer architectural representation, being the layers the Presentation Layer, Business Layer and the Data Layer. Each one of them are constituted by a number of components that are necessary to accomplish the objectives of this work. Figure 5.1 shows such representation, which support the different phases of the life cycle presented in Section 4.1 are used in this diagram to illustrate the different systems that are accomplishing the tasks and functions in each stage.

The functions of the modeling phase are executed through a modeling tool system and

it should provide functionalities to graphically depict the architecture of an application, have elements that allow to export and import deployment packages and components for managing the different elements that should be stored when saving the application model.

The CBR framework is also depicted in terms of three layers, the application layer contains a component that allows the communication with other entities the system, the web service API and is used to send the different characteristics that describe an application to the components of the engine, among other functionalities. The business layer contains elements that constitute the logical implementation of the application. In this proposal the similarity engine logic is composed of two similarity computation engines. The Functional Requirements Processor is the component responsible of computing graph similarity between the given $\alpha-$topology and the topologies stored in the knowledge base. The Non-functional Requirements Component has a similar function as the other element but in this case taking into account requirements of QoS. The similarity engine also interacts with the components of knowledge aggregator and knowledge manager which are the ones reaching with the data layer. Furthermore, the similarity engine processes the result provided by the functional and non-functional requirements processing components, retrieves and ranks the obtained solutions and shows in the first positions the topologies that are more appropriate to solve a current situation.

The Data Layer refers to the persistence means that the system uses. One of them corresponds to the performance knowledge, this is a database containing the description of different μ-topologies of applications that are running and also it stores data of different metrics and descriptions of the other non-functional characteristics. The application knowledge persistence unit saves the different topologies that are used by the CBR engine to determinate similarity of functional characteristics.

In the runtime phase three other system are employed. The first one is the *Provisioning Engine* which helps the developers in the task of deploying an application to the cloud and providing the resources that it needs. *The pricing and Cost Calculator Framework* returns estimation of incurred costs of applications when running in offerings of different cloud providers. *The Monitoring Framework* is responsible of updating the knowledge of applications that are deployed in different clouds in order to enrich the knowledge base of the CBR System. It must be pointed that the implementations of these three systems is not part of the focus of this work.

5.2 Non-functional Aspects Data Model

From the data model representation of an application presented in the section 4.1.2, the components that correspond to the non-functional characteristics and the viable topology distribution, were implemented following a relational database design an it is shown in Figure 5.2.

The entity-relationship model, depicts that the description of an application consists of one *QoS Specification* and one *Viable Topology*. One *QoS Specification* could correspond to more than

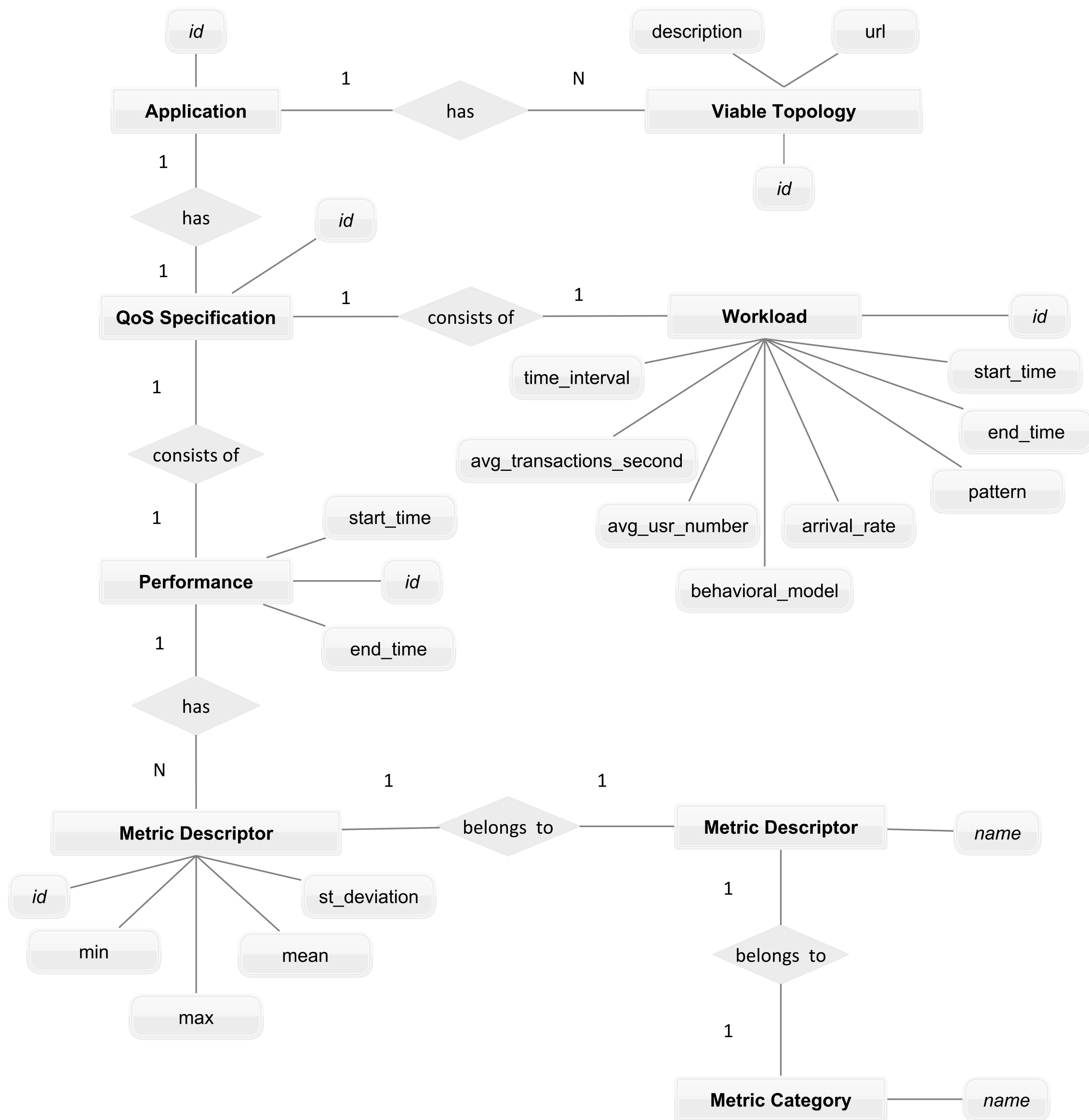


Figure 5.2: Cloud application non-functional aspects data model

one application and a *Viable Topology* distribution too. One *QoS Specification* entry consists of a *Workload* and a *Performance* description and they could also be part of other *QoS Specifications*. A *Performance* specification is composed of a set of performance *Metric Descriptors* that belong to a determinate *Metric* and a *Metric Category.*

Regarding the functional characteristics, represented by the α- and γ-topologies, since they are not handled directly by the CBR system but for an external service, they are represented in terms of a service template which basically consists of a XML representation of the node types and relationships of the distribution topology of the application.

5.3 Modeling Layout Design

The interfaces employed to get the information from the developer should be integrated in a smooth way to the modeling tool user interface in order to make them look as one more part of the environment. To achieve that, a new option to the main menu of the topology modeler and it has the name *Similarity Analysis*, when pressed it displays the following options :

- Performance Requirements
- Workload Characteristics
- Discover Similar Applications

5.3.1 Performance Requirements Specification

When the user selects the option *Performance Requirements* a panel is rendered and the current information about the performance is displayed, if no characteristics have been specified then the XML is displayed empty. Buttons that allow to add metrics according to a determined category are also displayed, when the user selects one of then a new panel with a series of fields is rendered; here the developer can specify per each metric the different descriptors (minimum, maximum, mean and standard deviation), once the desired fields are filled in he can add them by pressing the button *Add* or discard the changes by selecting *Cancel*.

If *Add* was pressed, then the panel containing the specification in XML format is reloaded to reflect the new changes. The developer can make this process the number of times he wishes per each metric category. The exposed above is illustrated in Figure 5.3.

5.3.2 Workload Characteristics Specification

The specification of workload characteristics works in the same way as in the case of the performance specification. When the user selects the option *Workload Characteristics*, a panel containing the description of the workload of the application is rendered and it is empty if it is the first time that is being rendered or no characteristics have been specified. If the developer selects the button *Specify Workload Characteristics*, a new panel is rendered with

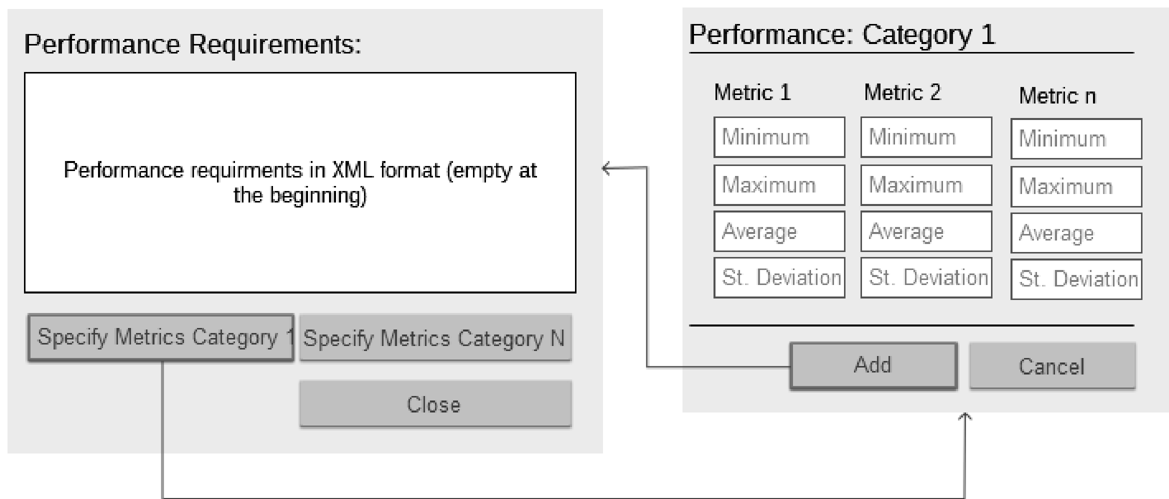


Figure 5.3: Performance specification layout

some fields where the user can describe the workload requirements of the application he is intending to deploy.

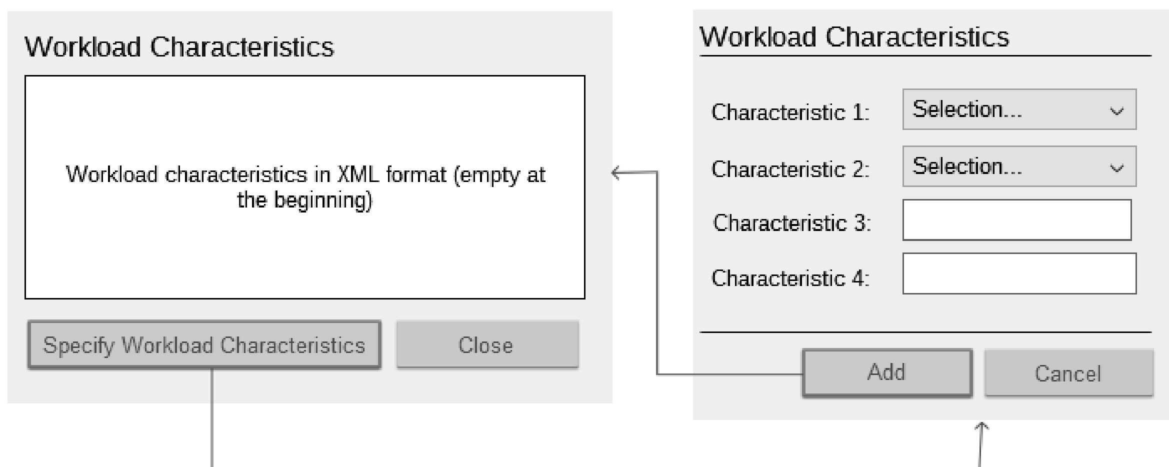


Figure 5.4: Workload specification layout

If certain field is allowed to have only a small set of values it is rendered as a combo box, otherwise the input field is displayed as a text box. Once the user has filled the desired information, he must press the button *Add* to effectively assign the specification to the application. If this process is completed successfully, then the panel containing the form closes and the characteristics he has specified are rendered in the first panel. The previously explained is illustrated in Figure 5.4.

5.3.3 Discovery of Similar Applications

Once the user has modeled an alpha topology of some application and has specified characteristics of workload and performance, he is able to retrieve similar applications and their solutions. In order to invoke the services of the CBR framework, the option *Discover Similar Applications* should be selected. A panel containing a table and two buttons is rendered, the table is empty at the first time but it is filled when the user uses the button *Discover Similar Apps*.

This button starts the actions required to send all the provided specifications to a service of the CBR framework, as an answer the list of similar applications, as well as the obtained similarity values, is rendered in the panel. If the user clicks on any cell under the column called ID, he will be able to visualize the viable topology of that particular application. When selecting the option view under the column Knowledge, a panel rendering the workload and performance characteristics of the respective application will be displayed. If the user selects the option view under the column Offering, a service that looks for candidate providers and offerings according to the specified characteristics of the application is called, the different options are rendered in a new panel and when the user selects one of them, he is able to see details regarding costs and characteristics of that particular offering. The descriptions provided above are illustrated in Figure 5.5.

There is one more column called *μ-topology*, if the developer selects the option Refine under this column, he is redirected to a new page where he is able to clearly visualize the α-topology he has depicted merged with the γ-topology of the viable distribution of the application he is inspecting. In this page the developer is also able to change nodes and elements of the distribution in order to make the proposed solution totally fit all the requirements.

The user is also able to save this new μ-topology to the repository of the modeling tool and the knowledge of the application to the CBR Engine. To accomplish this an option called *Persist Topology and Knowledge* is provided and when the user selects it, a panel is rendered where the user can input the name of the new distribution topology and store it and the given knowledge, in both cases and informative message of the result of the process is returned.

Furthermore, the developer may also request estimation costs for this new topology, this can be done by selecting the option *Distribution Cost*, immediately a panel with some fields is rendered and he is able to fill information regarding hard constraints and the application profile, used to request the cost information to an external service, and once the button *Estimate Cost* is pressed the respective information is rendered in the same panel. The user can perform changes to the different criteria and request estimations the times he considers necessary. Figure 5.6 illustrates the descriptions previously provided.

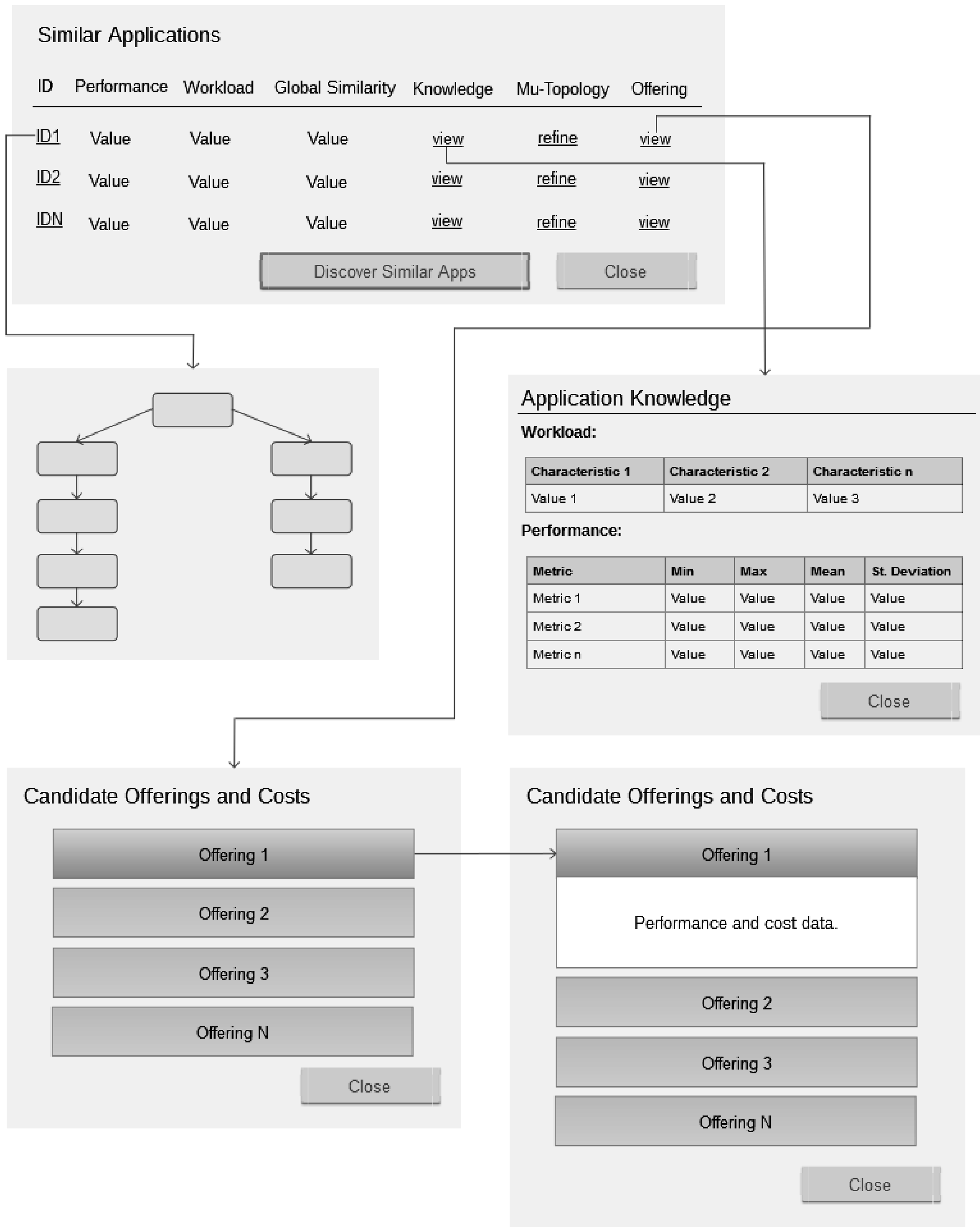


Figure 5.5: Discover similar applications layout

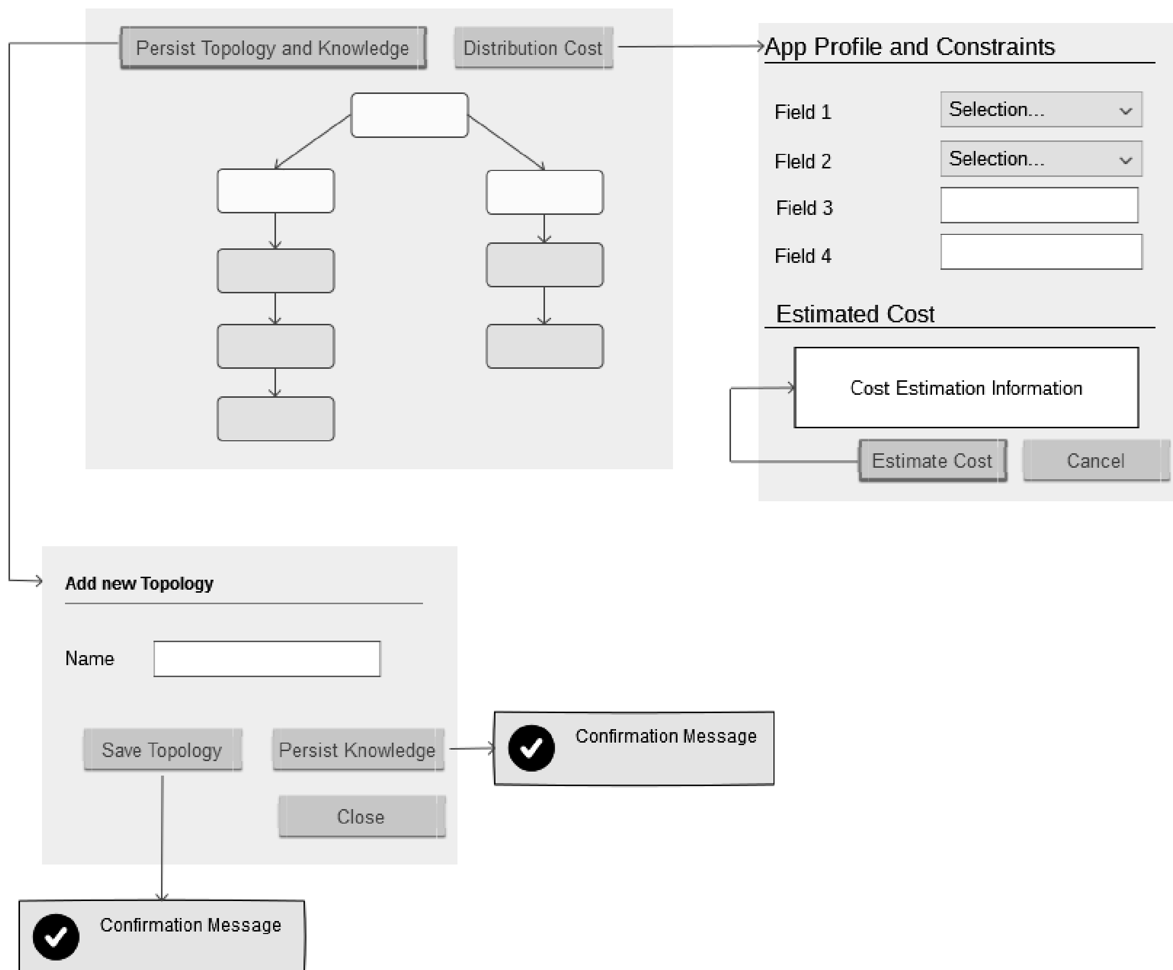


Figure 5.6: Refine application and cost calculation layout. In the refinement interface, the white boxes represent the nodes of the depicted α-topology and the gray boxes the γ-topology, together they conform the proposed μ-topology.

5.4 RESTful API

The REST API makes possible the control of the whole system via HTTP requests. A summary of the supported operations in the REST API is shown in Table 5.1.

Name	Description
Discover Similar Applications	This methods computes and returns local and global similarity measures between an application described through a set of characteristics and applications stored in a knowledge base. Based on the results of the similarity measures, a set of possible solutions for the current problem or μ-topologies is also returned.
Persist Knowledge	This method records in the knowledge base the characteristics of an application as well as the URI of the modeled μ-topology.
Retrieve Knowledge	This method returns the characteristics of a specific application given an id.
Update Knowledge	This method updates the attribute values of an application in the knowledge base, given an id.
Retrieve Similarity Tables	This method returns a list of similarity tables used by the system.
Update Similarity Table	This method updates the similarity values stored in a specific similarity table.

Table 5.1: REST API summary

Further descriptions of the different methods of the RESTful API are presented in the remaining of this section.

Method	**Discover Similar Applications**
Description	This methods computes the similarity of an application with others stored in a knowledge base, given a set of characteristics.
HTTP Request	POST /application-discoverability
URI params	-
Query params	• Workload and performance characteristics serialized in XML format.
Post params	-
Response	Returns a JSON object with the values of local workload and performance similarity per each similar application discovered and the URIs of their $/mu-$topologies.
Error responses	• 500 Internal Server Error • 400 Bad Request • 404 Not Found

Table 5.2: Description of REST method *Discover Similar Applications*.

Method	**Persist Knowledge**
Description	This method records the characteristics of an application as well as the URI of the modeled solution in the knowledge base.
HTTP Request	POST /application-knowledge
URI params	-
Query params	• Workload, performance characteristics and URI of the modeled μ-topology serialized in XML format.
Post params	-
Response	201 Created
Error responses	• 500 Internal Server Error • 400 Bad Request • 404 Not Found

Table 5.3: Description of REST method *Persist Knowledge*.

Method	**Retrieve Knowledge**
Description	This methods retrieves non-functional characteristics of an application given an identifier.
HTTP Request	GET /application-knowledge/{*id*}
URI params	• ID: Identification of an application object.
Query params	-
Post params	-
Response	Returns a set of the characteristics of the application in JSON format.
Error responses	• 500 Internal Server Error • 400 Bad Request • 404 Not Found

Table 5.4: Description of REST method *Retrieve Knowledge*.

Method	**Update Knowledge**
Description	This methods updates the non-functional characteristics of an application given an identifier.
HTTP Request	PUT /application-knowledge
URI params	-
Query params	• Application ID, workload, performance characteristics and URI of the modeled μ-topology serialized in XML format.
Post params	-
Response	200 OK.
Error responses	• 500 Internal Server Error • 400 Bad Request • 404 Not Found

Table 5.5: Description of REST method *Update Knowledge*.

Method	**Retrieve Similarity Tables**
Description	This methods retrieves the similarity tables used by the system.
HTTP Request	GET /similarity-tables
URI params	-
Query params	-
Post params	-
Response	Returns a list of all the similarity tables employed by the CBR system in JSON format.
Error responses	• 500 Internal Server Error • 400 Bad Request • 404 Not Found

Table 5.6: Description of REST method *Retrieve Similarity Tables*.

Method	**Update Similarity Table**
Description	This methods updates the entries of a similarity table.
HTTP Request	PUT /similarity-table
URI params	-
Query params	• Similarity table ID and table entries new values serialized in XML format.
Post params	-
Response	200 OK
Error responses	• 500 Internal Server Error • 400 Bad Request • 404 Not Found

Table 5.7: Description of REST method *Update Similarity Table*.

6 Implementation

This chapter presents details of the prototypical implementation of this thesis, taking into account the design considerations presented in the previous chapter. As explained in Chapter 4, the proposed methodology follows different stages which were also considered when realizing the designs and once again are employed to illustrate the different components of the implementation of the this prototype.

6.1 Modeling

As previously presented in Figure 5.1 the modeling stage makes use of a modeling tool system. As also pointed in Chapter 5, an existing system is adapted in order to provide the user interface that gets data from the user and contact the similarity analysis framework. The topology modeling tool should allow the developer to depict and visualize a graph-based application topology and also to add characteristics to each of its component such as requirements, capabilities, policies, among others. The selected tool for the prototypical implementation is Perfinery, which is an OpenTOSCA Winery extension that allows the creation of TOSCA-based application models through a web interface.

Winery is a web-based graphical modeling tool that allows the depiction of application topologies and management plans, its main components are: [Gan15]:

- Type, template and artifact management
- Topology modeler
- BPMN4TOSCA plan modeler
- Repository with storage systems

The type, template and artifact management are the components managing the node, relationship types and templates, etc. A graph-based visual topology is created through the use of the topology modeler component which employs node and relationship templates contained in a so-called service template. It also allows to attach relationship constraints, deployment artifacts and policies to the nodes and relationships of a depicted distribution template [KBBL13]. Perfiney extends Winery in order to allow the users to specify workload and performance characteristics of the different components of a distribution topology through the use of policy templates [Gan15], among other features.

For the purposes of this work the topology modeler is the only component that is being extended, furthermore the approach followed for Perfinery of using policies to specify workload and performance characteristics is not used because the requirements employed by the CBR system are the ones belonging to the whole application, not to individual components.

Additionally, the characteristics added by the developer are used for information purposes in this approach and should not be part of the service template of the application as Perfinery does.

The topology modeler component is implemented using Java and Java Server Pages (JSP) technologies, its web interface makes use of HTML5 and Javascript. The same technologies are employed to extend the web interface and invoke the services of the CBR Framework, the new functionalities are implemented in a way that they look like another part of the system. This extension consists of adding the menu *Similarity Analysis* to the main bar, the panels that are rendered when selecting options of the new menu option follow the same design and style principles of the existing functionalities, the refine topology interface described in the Figure 5.6 also follows these considerations. Figure 6.1 shows an overview of the Winery Topology Modeler interface with the menu option Similarity Analysis added to it.

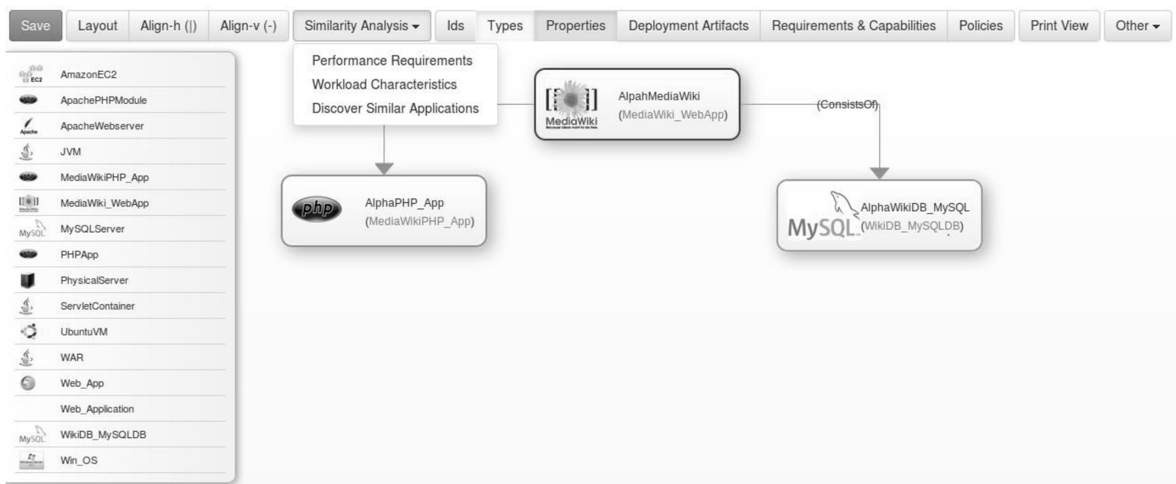


Figure 6.1: Winery Topology Modeler interface extended with the menu Similarity Analysis added

6.2 CBR - Similarity Analysis

The system executing the processes held on the similarity analysis phase is the CBR framework which is the one receiving data from the modeling tool, processing it and returning a list of similar applications and solutions. Updating and retaining knowledge in the different storing entities are also functions that the CBR framework must deal with. Details about implementation of the main components of this system are presented bellow.

6.2.1 Web Service API

The web service API that allows accessing the main functions of the CBR engine is implemented using the Java-based framework Spring. This is a high-level web framework that allows the development of RESTful applications, among other functionalities. Spring is a

URI	Parameters	Response Content
POST /application-discoverability	List of IDs of similar applications and specification of non-functional characteristics in XML format.	*List of objects, with fields:* appId workloadSimilarity performanceSimilarity globalSimilarity url
POST /application-knowledge	Specification of non-functional characteristics of the application and URI of solution in XML format	Status message
GET /application-knowledge/{id}	ID of the application	*workload:* id pattern arrival_rate behavioral_model avg_usr_number avg_transactions_second time_interval List of metric descriptors *lmetricd:* id min max mean st_deviation
PUT /application-knowledge	Application ID and specification of non-functional characteristics and viable distribution URI in XML format.	Status message
GET /similarity-tables	-	List of *similarityTable* objects, containing a list of *tableEntry* objects: id column_name row_name similarity_measure
PUT /similarity-table	Similarity table ID and table entries new values serialized in XML format.	Status message

Table 6.1: Details of URIs supported by the CBR Engine

lightweight container that offers a number of advantages when it comes to complexity and portability since it only needs a servlet container to execute. The main components of Spring include the *BeanFactory* which offers the basic functionality of managing Java beans of any nature and the *ApplicationContext*, which adds more advanced and specialized functionality to the BeanFactory [AA05].

Database entities are represented as Plain Old Java Objects (POJO) and Data Access Objects (DAO) to interact with the persistence media. Spring also makes uses of a class called controller to map the requesting URLs to a service and return an answer accordingly.

Table 6.1 shows a list of the URIs of the services provided by the CBR Framework, the parameters that the service expects and details of the returned answer.

As shown in some entries of the Table 6.1, several of the supported functions of the CBR Framework API need to receive information of the non-functional characteristics and solutions of the application in XML format, therefore a XML schema needs to be defined. Listing 6.1 shows the main structure of the XML schema employed when discovering similar applications, the main element is *SimilarityData* and is formed by a sequence of four other elements: *SimilarApps*, *Workload* , *Performance* and *Solution*. The element *SimilarApps* is a complex type formed by a series of simple types called *app_id* whose values represent the IDs of similar applications but only considering the functional requirements.

```
1  <xs:schema attributeFormDefault="unqualified" elementFormDefault="
      qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
2    <xs:element name="SimilarityData">
3      <xs:complexType>
4        <xs:sequence>
5        <xs:element name="SimilarApps">
6          <xs:complexType>
7               <xs:sequence>
8          <xs:element type="xs:string" name="app_id"/>
9         </xs:sequence>
10            </xs:complexType>
11     </xs:element>
12         <xs:element name="Workload">
13            <xs:complexType>
14             ...
15           </xs:complexType>
16         </xs:element>
17         <xs:element name="Performance">
18            <xs:complexType>
19             ...
20           </xs:complexType>
21         </xs:element>
22         <xs:element name="Solution">
23           <xs:complexType>
24             ...
25           </xs:complexType>
26         </xs:element>
```

```
27          </xs:sequence>
28        </xs:complexType>
29      </xs:element>
30    </xs:schema>
```

Listing 6.1: Specification of non-functional characteristics and solution of an application schema

The schema shown above is similar to the ones employed when persisting and updating knowledge but instead of using the type *SimilarApps*, a single element called *App* whose content is the ID of an application must be used.

The complex type *Workload* contains a sequence of simple types that correspond to the attributes defined previously in Table 4.2, the schema structure is shown in Listing 6.2

```
1  <xs:element name="Workload">
2    <xs:complexType>
3      <xs:sequence>
4        <xs:element type="xs:string" name="pattern"/>
5        <xs:element type="xs:string" name="arrival"/>
6        <xs:element type="xs:string" name="behavioral"/>
7        <xs:element type="xs:string" name="avg_users"/>
8        <xs:element type="xs:string" name="avg_transactions"/>
9      </xs:sequence>
10   </xs:complexType>
11 </xs:element>
```

Listing 6.2: Workload schema

In the case of the type *Performance*, it is formed by a sequence of complex types whose element represent a metric category, for example *Time Behavior*. The metric category type is conformed of another complex type that corresponds to an specific metric, followed by single types that depict the minimum, maximum, mean and standard deviation values. Due to space reasons, only a part of the schema is shown in Listing 6.3, nevertheless it illustrates the main parts of the mentioned schema. The categories and metrics that give name to the different types are the ones that are specified in Table 4.2.

```
1  <xs:element name="Performance">
2    <xs:complexType>
3      <xs:sequence>
4        <xs:element name="time_behaviour">
5          <xs:complexType>
6            <xs:sequence>
7              <xs:element name="response_time">
8                <xs:complexType>
9                  <xs:sequence>
10                   <xs:element type="xs:string" name="min"/>
11                   <xs:element type="xs:string" name="max"/>
```

```
12                  <xs:element type="xs:string" name="avg"/>
13                  <xs:element type="xs:string" name="st"/>
14                </xs:sequence>
15              </xs:complexType>
16              </xs:element>
17              <xs:element name="throughput">
18                <xs:complexType>
19                  <xs:sequence>
20                    <xs:element type="xs:string" name="min"/>
21                    <xs:element type="xs:string" name="max"/>
22                    <xs:element type="xs:string" name="avg"/>
23                    <xs:element type="xs:string" name="st"/>
24                  </xs:sequence>
25                </xs:complexType>
26              </xs:element>
27              ...
28         </xs:sequence>
29        </xs:complexType>
30        </xs:element>
31        <xs:element name="capacity">
32            <xs:complexType>
33              <xs:sequence>
34                <xs:element name="bandwith">
35                  <xs:complexType>
36                    <xs:sequence>
37                      <xs:element type="xs:string" name="min"/>
38                      <xs:element type="xs:string" name="max"/>
39                      <xs:element type="xs:string" name="avg"/>
40                      <xs:element type="xs:string" name="st"/>
41                    </xs:sequence>
42                  </xs:complexType>
43                  </xs:element>
44                    ...
45          </xs:sequence>
46            </xs:complexType>
47        </xs:element>
48      ...
49      </xs:sequence>
50    </xs:complexType>
51  </xs:element>
```

Listing 6.3: Performance schema

The complex type *Solution* is formed by a sequence of simple types, the element called *name* identifies the solution, *view_url* references the URI that has to be used to display the solution in the modeling tool, meanwhile the data contained in the elements *dist_url_host*, *dist_url_nsmu*and *dist_url_muid* is used to build the URI of the site where the refinement of the topology takes place. Listing 6.4 shows the schema for this type.

```
<xs:element name="Solution">
  <xs:complexType>
    <xs:sequence>
      <xs:element type="xs:string" name="name"/>
      <xs:element type="xs:string" name="view_url"/>
      <xs:element type="xs:string" name="dist_url_host"/>
      <xs:element type="xs:string" name="dist_url_nsmu"/>
      <xs:element type="xs:string" name="dist_url_muid"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Listing 6.4: Solution schema

When using the operation of update similarity table an object serialized in XML format also should be passed as parameter, Listing 6.5 shows the employed XML schema. The main element is called *SimilarityTable* and it must have an attribute *id*, that makes reference to the identification of the table in the knowledge base, this element is followed by a sequence of a complex type named *TableEntry* which represents the different cells in the table and is formed by simple elements that reference a column, row and the similarity value that should be stored the given position.

```
<xs:schema attributeFormDefault="unqualified" elementFormDefault="
    qualified" xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="SimilarityTable">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="TableEntry">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:string" name="column"/>
              <xs:element type="xs:string" name="row"/>
              <xs:element type="xs:string" name="value"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute type="xs:byte" name="id"/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Listing 6.5: Similarity table schema

6.2.2 Similarity Engine

The similarity engine is the component that computes the local and global similarity measures between a given problem and the cases stored in the knowledge base. The engine is compound of two elements, as previously shown in Figure 5.1:

- Functional requirements processing
- Non-functional requirements processing

For the implementation of the functional requirements processing component an external service is used. This service receives a depicted α-topology expressed as a service template and serialized in XML format. The requirement is done from the topology modeling tool, the external service performs the functional similarity computation trough the employment of a graph similarity method that considers the structure of the graphs, node and relationship types and returns a list of similar application IDs. The service is called *Pertos* and it is part of the work conducted by [Din16] [1], the functions that are being used are the ones that allow to perform the actions previously depicted, following the instructions provided in the documentation of Pertos.

It should be pointed that the database employed by the service performing the graph similarity is the one named as *Application Knowledge* in Figure 5.1 and there is consistency between the IDs of the applications stored in that database and the ones that are saved in the persistence unit called *Performance Knowledge*, where information about the non-functional characteristics of the applications is stored.

The processing of the non-functional requirements is implemented in Java and it basically consists on computing local similarity for every single attribute of workload and performance, following the algorithms defined in Sections 4.2.2 and 4.2.3.

Listing 6.6 shows the Java implementation of the computation of the workload similarity, it makes use of a function called *computeSingleLocalSimTable* which returns the similarity value of two attributes given their values and the respective similarity table.

```
1  public static float computeLocalWorkloadSim (Workload workloadCase,
      Workload workloadQuery,
2    TreeMap<Integer,List<TableEntry>> lTables) throws Exception{
3
4    float sim =-1;
5    ArrayList<Float> lSimList = new ArrayList<Float>();
6
7    lSimList.add(computeSingleLocalSimTable(workloadCase.getPattern(),
8              workloadQuery.getPattern(),lTables.get(1)));
9    lSimList.add(computeSingleLocalSimTable(workloadCase.getArrival_rate
        (),
10           workloadQuery.getArrival_rate(),lTables.get(2)));
```

[1] To be published

```
11     lSimList.add(computeSingleLocalSimTable(workloadCase.
           getBehavioral_model(),
12             workloadQuery.getBehavioral_model(),lTables.get(3)));
13     lSimList.add(computeSingleLocalSim(workloadCase.getAvg_usr_number(),
14           workloadQuery.getAvg_usr_number()));
15     lSimList.add(computeSingleLocalSim(workloadCase.
           getAvg_transactions_second(),
16           workloadQuery.getAvg_transactions_second()));
17     sim = GlobalSimilarity.computeSingleGlobalSimilarity(lSimList);
18     return sim;
19
20   }
```

Listing 6.6: Workload similarity computation

To compute the similarity between two metrics it is necessary to calculate the values of local similarity per each metric descriptor, add them to a list and them aggregate them through the application of the global similarity measure. Listing 6.7 shows details of the implementation of the function that makes the computation of local similarity.

```
1  public static float computeSingleLocalSim(Float caseCB, Float query){
2    float difference = 0;
3    float similarity =-1;
4    if (caseCB != null && query != null && caseCB != UNDEFINED && query
         != UNDEFINED) {
5      if (caseCB > 0  && query > 0 ){
6        difference = (float) (Math.log(caseCB) - Math.log(query));
7      }
8      else if(caseCB < 0  && query < 0){
9        difference = (float) (-1*(Math.log((caseCB*-1))) + Math.log((
             query*-1)));
10     } else if (caseCB == 0 || query == 0) {
11       difference = caseCB - query;
12     }else{
13       difference = -1;
14     }
15
16     if(difference != -1){
17       similarity = (float) Math.exp(-1* (Math.abs(difference)));
18     }
19   }
20
21   return similarity;
22 }
```

Listing 6.7: Single local similarity computation

All the local similarities values of attributes are aggregated in order to obtain a single value of similarity of workload and performance; these values are once more aggregated in order to

obtain an unique value of global similarity, in both cases the employed algorithm is the one presented in Listing 6.8.

```
1  public static float computeSingleGlobalSimilarity(List<Float>
      localSimList) {
2    float acum = 0;
3    int cont = 0;
4    for (int i = 0; i < localSimList.size(); i++) {
5      if (localSimList.get(i) != -1)
6      cont++;
7    }
8    if (cont != 0) {
9      float weight = 1.0f / cont;
10   for (int j = 0; j < localSimList.size(); j++) {
11     if (localSimList.get(j) != -1) {
12       cont++;
13     acum = acum + (weight * localSimList.get(j));
14     }
15   }
16   return ParserXML.round(acum, 2);
17   }
18   return -1;
19 }
```

Listing 6.8: Global similarity computation

6.2.3 Knowledge Aggregator and Manager

These two components are implemented as the data services and data access classes that need to be created as part of the Spring programming environment. As their names point, their functions are in the frame of data administration, access and preparation for the similarity analysis.

6.3 Runtime

6.3.1 Provisioning Engine and Monitoring Framework

The provisioning process takes place when deploying a modeled application to the cloud and assigning the requested resources to it. The provisioning engine is not part of the implementation of this work, nevertheless in a real scenario the use of one is very important. One example of this is *Vinothek*, which is part of the OpenTOSCA environment and allows users to provision cloud instances through the use of a web interface [BBKL14b].

As it has been mentioned in previous chapters, the use of a framework that monitors and periodically registers metric measures of applications that are running in the cloud is necessary

Attribute	CBR Framework	Nefolog
CPU Cores	Number of Processors	cpuCores
CPU SPeed	Processor Speed	cpuSpeed
I/O Performance	I/O Operations	io
RAM	Memory Allocation to VM	memory
Local Disk	Storage Size	storage
Bandwith	Bandwith	bandwith
Transactions	Average number of transactions	transactions

Table 6.2: Mapping of existing attributes in the case base to Nefolog available attributes

to update and maintain the information that is stored in the knowledge base. The implementation of such a framework is not part of this work as well, however the employment of one is crucial in order to enrich the stored cases and therefore the quality of the results of the CBR Framework.

6.3.2 Pricing Knowledge and Cost Calculation Framework

When distributing applications in the cloud and considering that nowadays there is a number of providers, having an estimation of the monthly cost of a running application that meets all existing requirements is without doubts really useful. The present work does not implement a pricing knowledge and cost calculation framework but it uses the services of one existing called *Nefolog*.

Nefolog is a decision support system that provides cloud candidate offerings search and cost calculation functionalities, exposed through a RESTful API. The candidate search functionality is performed considering the requirements and specifications of the application and comparing them to the data stored in its knowledge base, meanwhile the cost calculation is done by applying cost models to the different candidate offerings [XAo13].

The services of Nefolog are being invoked in the current implementation when discovering a list of similar applications for a given description, this was previously presented in the Figure 5.5. Nefolog defines a number of different parameters that should be provided in order to determinate the offerings that suit the current conditions. Not all of these parameters are considered in this work, since not all of them are present in the case base, therefore a mapping of the existing attributes of the applications saved in the case base with the equivalent ones in Nefolog is performed, this mapping is presented in Table 6.2.

The Nefolog attributes shown in Table 6.2 used in conjunction with the attribute *service type* and filled with the information of a particular application stored in the knowledge base, are employed to build the necessary query strings to obtain the candidate offerings that could fulfill the existing requirements. An example of the use of one query string is presented in the code snipped 6.9.

```
1 ../nefolog/candidateSearch?servicetype=application&cpuCores=9&cpuSpeed
    =1500&io=moderate&memory=15&storage=600&bandwidth=400&transactions
    =5000&media=json
```

Listing 6.9: Example of a Nefolog Candidate Search query string invoked from the CBR framework

Nefolog requires a configuration of particular offering and a set of other attributes in order to estimate the costs of a distribution in the cloud. Since the candidate search functionality provides configuration identifications of the offerings, they can be used to perform the cost calculation query. In the case of the attributes, the ones that Nefolog defines are mostly different from the ones that are used for the candidate search, therefore the only that can be employed when estimating costs of an existing solution in the case base are the storage size, the number of transactions, I/O operations and virtual machines. The code snipped presented in 6.10 shows a sample of the use of query string from the CBR engine in this situation.

```
1 ../nefolog/costCalculator?configid=32&GB=600&Transactions=5000&Server
    =13
```

Listing 6.10: Example of a Nefolog Cost Calculator query string invoked from the CBR framework

The disadvantage of only considering few attributes when performing the cost calculator query is that the obtained result might not be as accurate as it could be. In order to face this situation, the implemented prototype allows the user to add more attributes to the Nefolog query through the definition of an application profile and hard constraints when he is in the stage of refining an application and in this way the calculations returned by the decision support system could reach higher levels of accuracy. The user can define the location zone of the infrastructure as a constraint and also select the attributes he considers necessary to perform the estimations. The different fields as well as the available location zones were obtained from the documentation of the Nefolog Service [XAo13].

7 Validation and Evaluation

This chapter presents the methodology employed to verify the validity of the results provided by the developed prototype as per the specifications and designs considerations presented in Chapters 4 and 5.

7.1 Methodology

The methodology employed to evaluate the prototypical implementation of this work consists of three main parts, which are presented below and that have been conceived based on the proposals of [WL01]:

- Test cases definition
- Case retrieval correctness evaluation
- System behavior after adaptation evaluation

The *test cases definition* is composed of two tasks: The first one consists of defining the full set of functional and non-functional characteristics as well as the application profile that the test case should have. In the second one, the characteristics that the cases stored in the knowledge base must meet in order to effectively measure the correctness of the system have to be established. In this set of cases there should be elements whose characteristics are approximately equal to an application used to test the system, cases with a minor degree of equality and cases that are completely different. At this point is obvious that the expected result from the system is the set of cases which have certain degree of similarity with the test application.

The tasks that belong to the *case retrieval correctness evaluation* are the ones that are helping to determinate if the functionalities offered by the prototype are working properly and if the CBR Framework returns the expected results. The first two tasks consist of modeling the α-topology of the previously defined test application and entering all its non-functional characteristics through the employment of the modeling tool Perfinery.

Third task entails the invocation of the discover similar application service of the CBR Framework whereas the fourth is the selection and refinement of a solution. Once changes to the proposed topology have been performed, the next step is to persist the viable distribution and its knowledge to the system, which constitutes the fifth task. If all the previous steps were successfully completed, then it is possible to proceed with the last task which consists of analyzing the obtained results, this means to examine which cases where returned when discovering the similar applications and compare them with the expected results. Furthermore, verifications of the calculations of the different similarity measures are carried out.

Last process of the methodology consists of verifying and evaluating the behavior of the system once a new case, from the previous tasks, is added to the case base. This process has been called *system behavior after adaptation evaluation* and it basically involves the repetition of tasks of the previous part with exception of the adaptation and storage of the new solution. The premise at this stage is that if the developer models the exact application that he modeled before and inputs in the system the same non-functional characteristics, the application and the solution that he previously depicted is returned with a 100% degree of similarity, besides the other similar applications that already exist in the case base.

After finishing all the set of defined tasks, it is possible to determinate whether the system is behaving as expected or not. Figure 7.1 shows a summary of the three process of the evaluation methodology as well as the tasks that need to be performed in each stage.

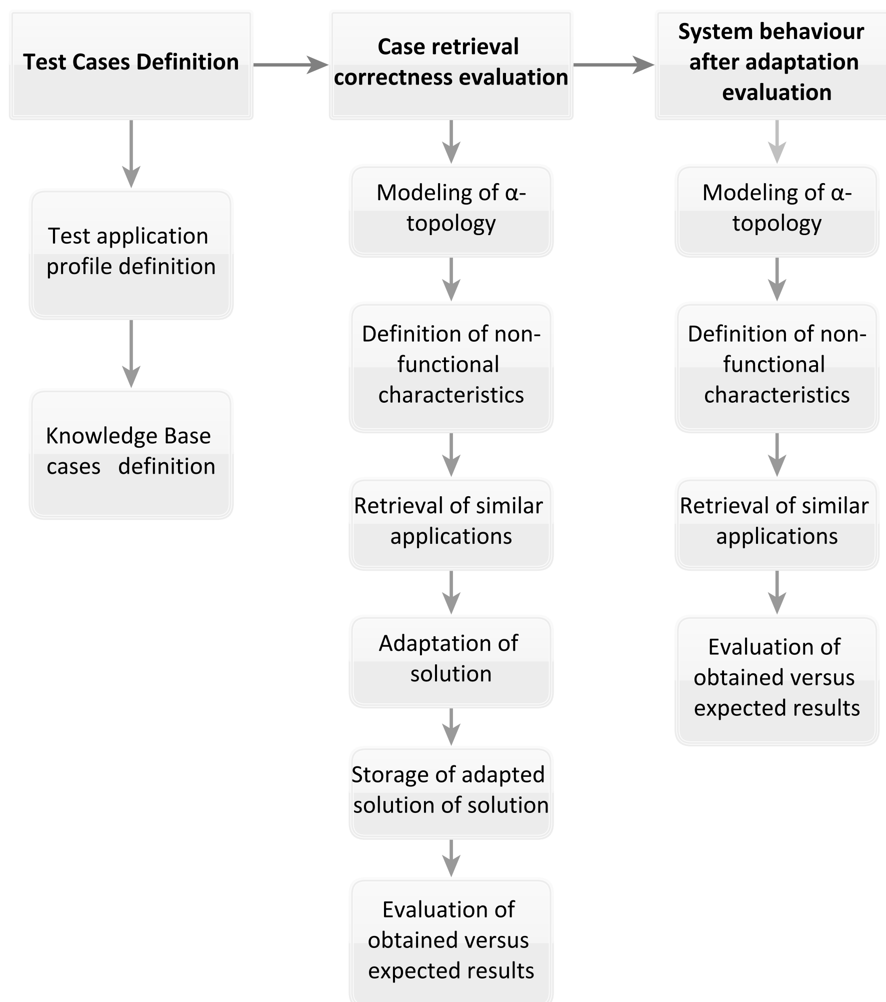


Figure 7.1: Evaluation methodology and task

Additionally, the functions of *Retrieve and Update Similarity Tables*, which are performed by a domain expert, are evaluated by employing the REST API client Postman[1], this two operations are performed and the details of their responses are observed to determinate if they are the ones expected.

7.2 Evaluation by Means of Case Study: MediaWiki Application

7.2.1 Test Cases Definition

The methodology described above is executed by taking the MediaWiki Application (Wikipedia) as case study. MediaWiki is a highly known web application implemented in PHP, makes use of a MySQL database and that is distributed as a two-tier application and they constitute the functional requirements for the test case. A representation of the expressed above is shown in Figure 7.2.

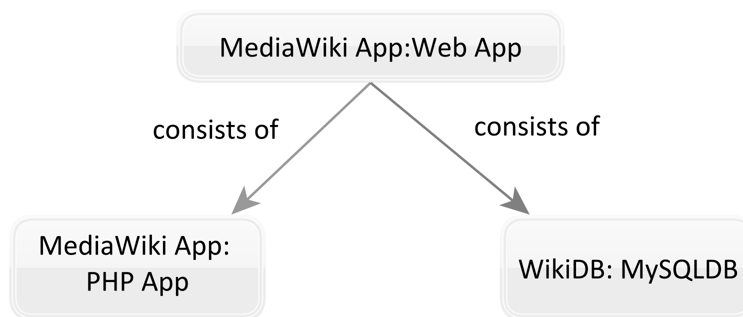


Figure 7.2: MediaWiki Application depicted as a topology

The workload characteristics of the test case are presented below whereas the performance requirements in Table 7.1, part of the data has been obtained from [UPVS09] and [wik15] and used where applicable:

- *Pattern:* Continuously changing
- *Arrival rate distribution:* Normal
- *Behavioral model:* Normal
- *Average number of users:* 2000
- *Average number of transactions:* 10000
- *Time period:* 30 days

As previously established in Section 4.1, the knowledge base is populated with a number of cases that have the following characteristics and based on the ones defined for the test case:

[1]Postman website: https://www.getpostman.com/

Metric	Minimum	Maximum	Mean	St. Deviation
Response Time (ms)	100	800	450	494.97
Throughput (request/sec)	2	3	2.5	0.71
Processing Time (ms)	100	800	450	494.97
Read Speed (revolutions/min)	7200	7200	7200	0.00
Write Speed (revolutions/min)	7200	7200	7200	0.00
Resource Migration Time (sec)	60	120	90	42.43
Latency (ms)	100	480	290	268.70
Backup Time (sec)	120	300	210	127.28
Bandwidth (Mbps)	100	600	350	353.55
Processor Speed (GHz)	8.8	12.8	10.8	2.83
Storage Size (GB)	500	500	500	0.00
Memory Allocation to VM (GB)	5	5	5	0.00
Number of VM	1	5	3	2.83
Number of Processors	4	4	4	0.00
I/O Operations	5000	8000	6500	2121.32
Network Utilization (%)	60	100	80	28.28
Memory Utilization (%)	60	100	80	28.28
Disk Utilization (%)	40	100	70	42.43
CPU Utilization (%)	60	100	80	28.28
VM Utilization (%)	60	100	80	28.28
Number of VM per Physical Server	1	5	3	2.83
Resource Acquisition Time (sec)	60	120	90	42.43
Resource Provisioning Time (sec)	60	120	90	42.43
Deployment Time (sec)	30	120	75	63.64
Resource Release Time (sec)	30	60	45	21.21
VM Startup Time (sec)	30	60	45	21.21
Cloud Service Uptime (%)	99	100	99.5	0.71
Cloud Resources Uptime (%)	99	100	99.5	0.71
Mean Time Between Failures (hour)	48	48	48	0.00
Mean Time to Repair (hour)	1	1	1	0.00

Table 7.1: Performance metrics for the MediaWiki test case

- One application that has the same α-topology and characteristics slightly different to the test case, the assigned ID is 1.
- One application with the same α-topology and characteristics with more differences than the previous case, the assigned ID is 2.
- One application with the same α-topology and characteristics with an even higher degree of difference than the other two cases, the assigned ID is 3.
- One application with very similar non-functional characteristics but a different α-topology, the assigned ID is 4.
- A number of applications with notorious differences of functional and non-functional characteristics. The IDs of these cases start from 5.

7.2.2 Case Retrieval Correctness Evaluation

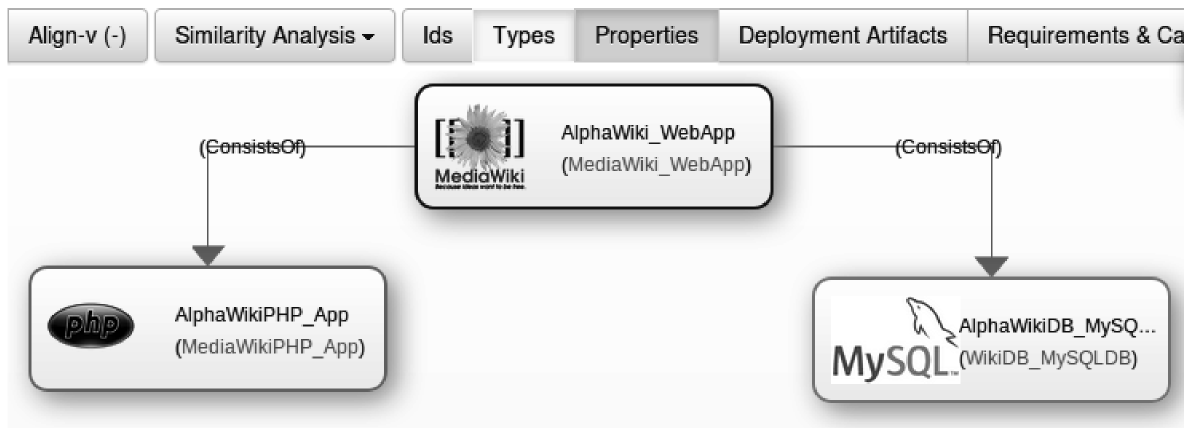


Figure 7.3: α-topology of the test case modeled in Perfinery

Tasks for this stage are performed on a machine with Ubuntu version 14.04, hosting both the modeling tool and the CBR Framework service, the knowledge base deployed on MySQL server version 5.5.44 and employing the browser Mozilla Firefox version 35.0.1.

First step of this stage of the evaluation process consists of modeling an α-topology that represents the functional requirements of the test case. To achieve this the modeling tool Perfinery is accessed through the defined web browser, on the main page a new service template is created by clicking on the button *Add new* with the name *New_Media_Wiki*. Afterwards, the topology editor is accessed and the creation of the nodes that correspond to the types defined in Section 7.1 and with names that start with the prefix *Alpha* is performed and then saved. The result of this task is presented in Figure 7.3.

Next step is the definition of non-functional requirements. At first the workload characteristics are input by selecting the option *Workload Characteristics* from the menu *Similarity Analysis* and by filling in the form that is displayed when clicking on the button *Specify Workload Characteristics*. Figure 7.4 shows the correspondent form with the test data. After adding

them when selecting the option *Add*, the workload information is displayed as XML format, confirming that the process was successful.

Figure 7.4: Specification of workload characteristics in Perfinery

Subsequently the performance characteristics are entered by selecting *Performance Requirements* from the *Similarity Analysis* menu. In the displayed panel each of the buttons that shows the forms to specify the metrics is selected and they are added by clicking on the option *Add*. Figure 7.5 shows the form with the different values of the metrics corresponding to the category of *Resource utilization*, whereas the picture 7.6 shows part of the XML with the recently added values which indicates that the process was successful, in a similar way than in the workload characteristics specification process.

Now one of the most important tasks takes place, the retrieval of similar application and their solutions. Once the functional and non-functional characteristics have been defined, the option *Discover Similar applications* from the menu *Similarity Analysis* is selected and then the services of the Similarity Engine are invoked. After waiting for a few moments an answer containing a list of applications, their similarity values and several other options is returned. Manual calculations are performed and the values returned by the CBR system are the expected ones. At this point is also possible to examine the retrieved viable distribution by clicking on the link under the column ID. Figure 7.8 shows the topologies for the applications with ID 1 and 2, it is possible to appreciate the differences that their distributions have. Figure 7.7 shows the values obtained of the similarity analysis.

When selecting any of the options of *view* under the column *Distribution Cost*, the Nefolog service is invoked and candidate offerings for the particular solution of the specified application is shown as well as an estimation of the monthly costs. Furthermore, the knowledge of every retrieved application can be inspected by clicking on the corresponding link under the column *Knowledge* and invoking the services of the CBR Framework as it can be observed in

Performance: Resource Utilization Requirements

Network Utilization (%)	Memory Utilization (%)	Disk Utilization (%)
60	60	40
100	100	100
80	80	70
28.28	28.28	42.43

CPU Utilization (%)	VM Utilization (%)	VM per Physical Server
60	60	1
100	100	5
80	80	3
28.28	28.28	2.83

Add Cancel

Figure 7.5: Specification of Performance requirements of the category Resource Utilization in Perfinery

Figure 7.9, where a part of the workload and performance characteristics of the application 3 is displayed.

App Knowledge

Workload Time Interval monthly

Pattern	Arrival Rate	Behavioral Model Max	Avg. Transac/Sec	Avg. User number
once in a lifetime	logarithmic	normal	2000	200

Performance:

- Time Behaviour -

Metric	Min	Max	Mean	St. Deviation
Response Time	500	3000	1750	1767.77
Throughput	0	2	1	1.41
Processing Time	300	4000	2150	2616.3
Avg. Read Speed	2000	2000	2000	0
Avg. Write Speed	2000	2000	2000	0
Avg. Migration Time	200	1000	600	565.69
Latency	95	460	277.5	258.09
Backup Time	500	1000	750	353.5

Figure 7.9: Part of the retrieved knowledge from an application

Figure 7.6: Performance Requirements currently added

Similar applications:

ID	Workload	Performance	Global Similarity	Knowledge	Mu-Topology	Distribution Cost
1	0.92727274	0.85930806	0.8932904	View	Refine	View
2	0.51666665	0.6663365	0.5915016	View	Refine	View
3	0.325	0.40495777	0.36497888	View	Refine	View

Discover Similar Apps Close

Figure 7.7: Results of the similarity analysis

The adaptation of the solution is now performed, this is achieved by clicking on the link *refine* under the column *Mu-topology*. A new tab is opened and there the previously modeled α-topology is rendered together with the γ-topology of the selected viable distribution, it is possible to clearly identify them since the α-topology is colored in yellow and the γ-topology in green, as shown in Figure 7.10.

A node of the topology has been modified for purposes of the evaluation, it corresponds to the server hosting the application and the database, the node of type *Amazon EC2* is replaced by one of type *Physical server*. The performance and workload characteristics entered in the previous task can also be displayed here when selecting the corresponding options under *Similarity Analysis*.

Additionally, it is possible to invoke the Nefolog candidate offerings search and cost estimations service in the current interface. This is done by clicking on the button *Distribution Cost*, a new panel is rendered and the options of candidate offerings are loaded in a select box, according to the workload and performance characteristics previously defined. It is possible to specify constraints and characteristics of the application profile using the interface. In the

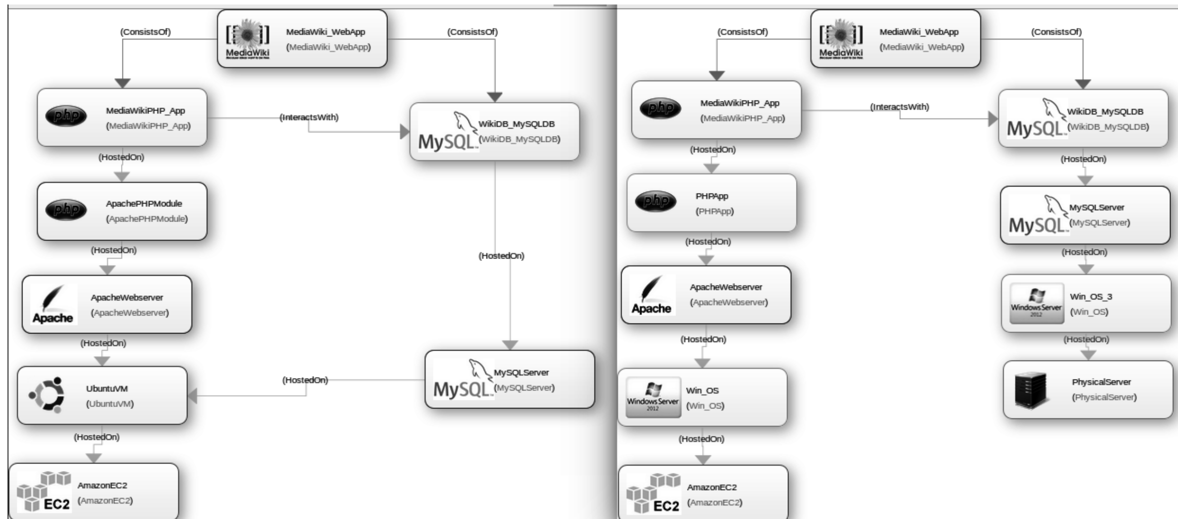


Figure 7.8: View of two viable distributions retrieved through the similarity engine

current case the restriction is that the application must be located in Oregon and that it will have a load of 20000 queries per month and must have 5 servers, when pressing the button *Estimate Cost* the Nefolog service is invoked one more time and the cost of the infrastructure for a 10-months period is displayed. Figure 7.11 present the interface employed to estimate the cost of the distribution.

It is possible now to store the refined solution to the knowledge base, to perform this the option *Persist Topology and Knowledge* is selected and a new panel where is possible to give a name to the new solution is rendered. In this case the viable topology is named as *NewWiki* and it is added to the repository by selecting the option *Save Topology*, a confirmation message is obtained and immediately is possible to persist the non-functional characteristics by clicking on the button *Persist Knowledge*, a new confirmation message is obtained. Since everything went fine, now the new solution is added to the repository of Perfinery and to the knowledge base.

The results provided by the CBR framework have been accurate. It was expected that the applications with IDs 1,2 and 3 were the ones returned, in that order and that was the result obtained. As it was shown in Figure 7.7, the global similarity value for the application 1 was 0.893, for the one with ID 2 was 0.591 and for the application 3 the computed value was 0.364 and this allows to confirm that the similarity measures that are being employed are working in a proper manner. Furthermore and as it was previously mentioned, the similarity values computation was also executed using other software and obtaining the same outcomes.

Another important aspect that should be mentioned is the fact that the options offered by the similarity engine are integrated on a smooth way into Perfinery and they appear to be one of the original functionalities of the system, as it can be observed in the series of figures presented in this chapter. Additionally, it was corroborated that the retrieval and storage of the knowledge as well as the visualization of the solutions were correctly executed, confirming once more that the system is behaving as required.

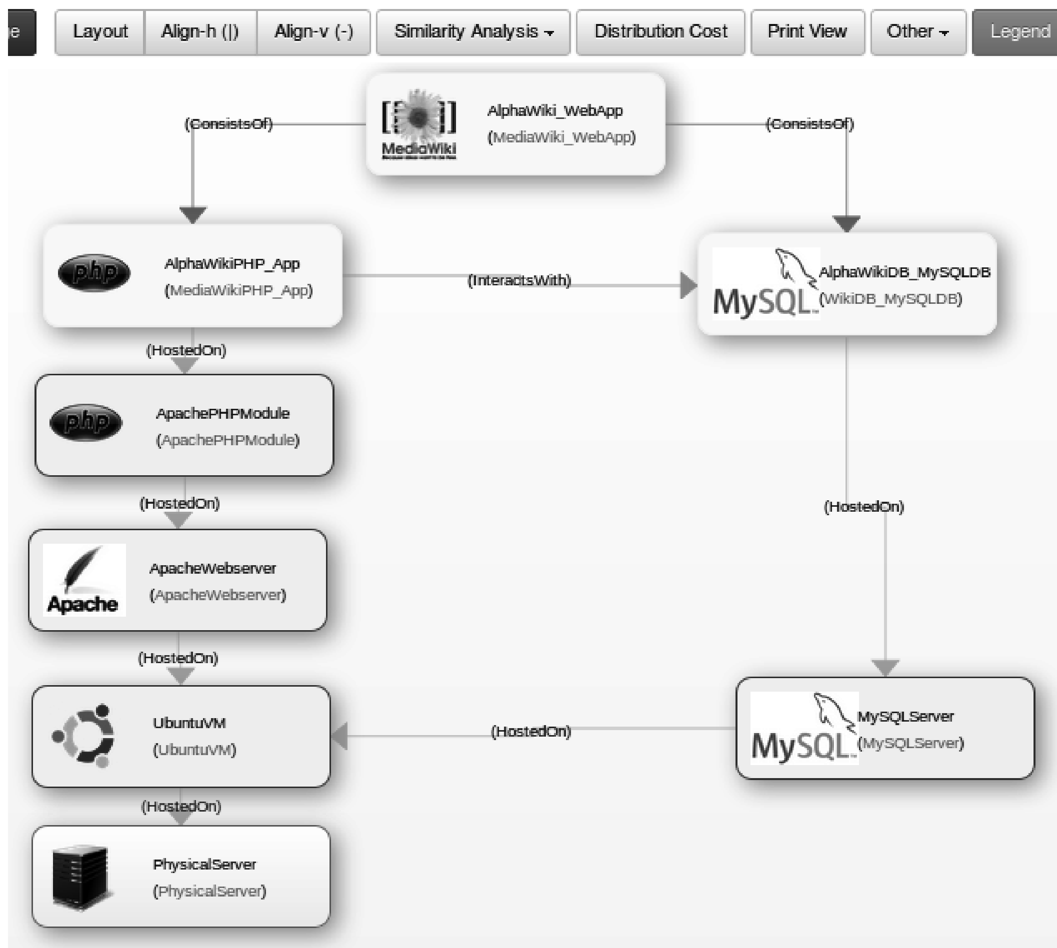


Figure 7.10: Refining a selected viable topology

7.2.3 System Behavior after Adaptation Evaluation

The objective of this stage is to verify that the system behaves correctly after a new case and its solution have been added to the knowledge base. The first two tasks of this phase included the modeling of the α-topology and definition of non-functional requirements, the described application had exactly the same characteristics than the one previously depicted. Subsequently the retrieval of similar applications task takes place, obtaining the results presented in Figure 7.12.

The application with the highest similarity value is the one whose ID is number 10 and it corresponds to the one previously described, its similarity value is 1 whereas the applications with IDs 1 to 3 are returned in the same order and with values of similarity as was expected. A simple inspection of the distribution confirms that the system is returning the solution that was also previously refined. In this way, it is possible to conclude that the system behaves satisfactorily and that the design approaches of the present work offer consistent and accurate results.

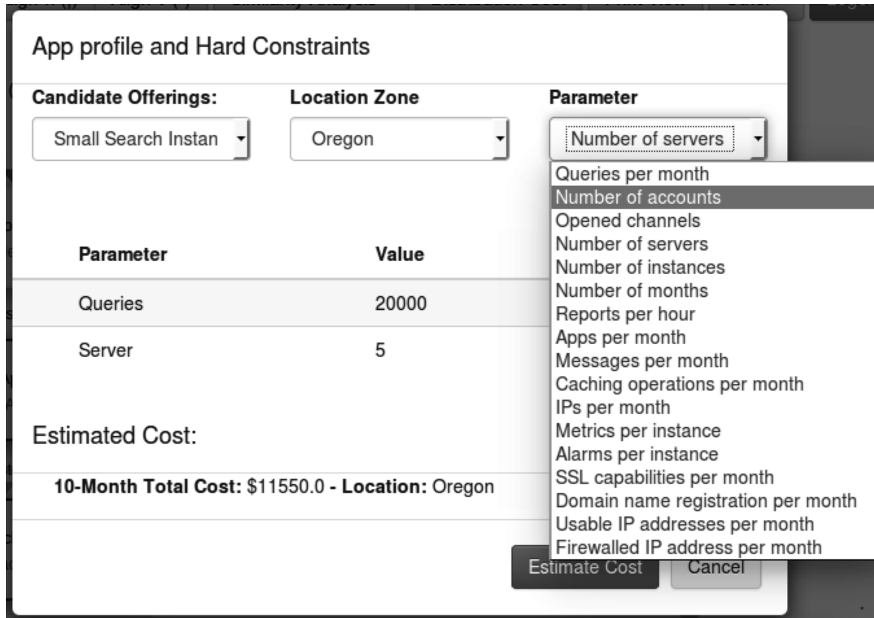


Figure 7.11: Cost calculation interface in Perfinery

Similar applications:

ID	Workload	Performance	Global Similarity	Knowledge	Mu-Topology	Distribution Cost
10	1	1	1	View	Refine	View
1	0.92727274	0.85930806	0.8932904	View	Refine	View
2	0.51666665	0.6663365	0.5915016	View	Refine	View
3	0.325	0.40495777	0.36497888	View	Refine	View

Discover Similar Apps | Close

Figure 7.12: Retrieval of similar applications after a new case and solution have been inserted into the knowledge base

7.2.4 Domain Expert Operations Validation

As indicated in Section 7.1, the evaluation of the operations of *Retrieve and Update Similarity Tables* are performed using the REST API client Postman. Details of the executed requirements and responses are shown below:

- Retrieve Similarity Tables:
  - URL: ../SimilarityEngine/similarity-tables
  - Method: GET
  - Status code: 200 OK. The response contains the list of similarity tables of the system and their entries in JSON format as shown in Figure 7.13.
- Update Similarity Tables:
  - URL: ../SimilarityEngine/similarity-table
  - Method: PUT
  - Parameters: A similarity table with their entries and respective similarity values depicted in XML format with the value of the first entry modified.
  - Status code: 200 OK. It was verified that the correspondent value in the database was updated by retrie

Figure 7.13: Body of the obtained response when invoking the REST function of retrieving similarity tables in Postman

8 Outcome and Future Work

The expansion and advent of the Cloud Computing paradigm has brought a number of benefits but also challenges. Nowadays, it is not just about having an application deployed and running in the cloud but also fully exploiting the advantages that the cloud technologies offer. This is not a trivial task and requires a considerable amount of effort.

The components of an application can be deployed in a variety of ways among different offerings. Nevertheless, there is a lack of support when it comes to assist the developers in the task of selecting the right cloud services to host the different application components and that fulfill their requirements. Therefore, this work aims to provide the necessary support to application developers in the tasks of designing the topology of one application to partially or fully deploy it in the cloud. In order to implement such a tool, the Case-Based Reasoning with Similarity Retrieval paradigm was employed.

To achieve the objectives of this thesis, it was necessary to perform an analysis of the state of the art. Moreover, basic concepts, theories, models and relevant technologies used or extended in the scope of this work are explained and described in Chapter 2. Analysis of previous and ongoing researches with similar objectives as the ones of this thesis were also conducted. Two examples of the studied solutions are SMICloud and PatEvol, their strengths, weaknesses and the challenges that their designs implied were evaluated in order to learn from them, as Chapter 3 presents. The analysis of the state of the art and the study of the existing solutions allowed us to develop the concepts and the methodology that guides the entire development of this thesis. Chapter 4 describes them as well as the characteristics of the knowledge of the applications that is captured, how that knowledge should be used to identify potential viable topologies and the guidelines that the design of the solution must follow.

Subsequently, the architecture of the proposed system was designed focusing on its components, their interactions and the set of operations that a user can perform. All these considerations were presented in Chapter 5 and using them alongside with the specifications introduced in Chapter 4, it was possible to proceed with a prototypical implementation of the system. The different functionalities of the CBR Framework were exposed through a RESTful API and details about the use of the functions, data format of parameters and returned answers are described in detail in Chapter 6 as well as implementation considerations of the different algorithms for computing the similarity measures. Furthermore, details of the modification of the Perfinery modeling tool as well as of the usage of the Nefolog service were also depicted.

It was necessary to define a methodology to validate the implemented prototype, which is presented in Chapter 8. An evaluation by means of a case study was performed and the obtained results allowed us to conclude that the use of the CBR approach, with the similarity measures that were defined in this work, can effectively assist to the application developers

in the decision making process of choosing a viable topology. However, the quality of the results depends not only on a good selection of characteristics and similarity measures but also on the quality of the cases of the knowledge base, therefore the formation of the case base is a task that should be performed with a high level of meticulousness and detail.

In the future, the system could also offer means to allow to developers or experts define their own similarity measures and weighting of attributes, which will provide higher levels of flexibility to the system. Currently, the ranking of the solutions is executed in terms of the result of the similarity computations, it would be also possible to use *utility functions* to rank them and in that way the applications and solutions appearing on higher positions are not only the ones that mathematically are the most similar, but also the ones that have been useful and have met the needs of other users. The integration with a monitoring system and the implementation of knowledge aggregator components give place to further enrichment of the knowledge base since more information about performance and workload metrics could be captured and therefore an improvement of the quality of the results might be possible. It would be also feasible to integrate the system with a provisioning engine such as *Vinothek*, part of the OpenTOSCA environment, in this way it would be possible to facilitate to the developers the tasks of instantiating the applications in the cloud and consequently a better support in the whole design process of application topologies could be provided.

Bibliography

[AA05] J. Arthur and S. Azadegan. Spring Framework for rapid open source J2EE Web Application Development: A case study. In *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, 2005 and First ACIS International Workshop on Self-Assembling Wireless Networks. SNPD/SAWN 2005. Sixth International Conference on*, pages 90–95. IEEE, 2005.

[ADC10] M. Alhamad, T. Dillon, and E. Chang. Conceptual SLA framework for cloud computing. In *Digital Ecosystems and Technologies (DEST), 2010 4th IEEE International Conference on*, pages 606–610. IEEE, 2010.

[AFG+10] M. Amburst, A. Fox, A. Griffith, R. Katz, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of Cloud Computing. *Communications of the ACM*, 53:50–58, 2010.

[AGSLW14] V. Andrikopoulos, S. Gómez Sáez, F. Leymann, and J. Wettinger. Optimal distribution of applications in the cloud. In *Advanced Information Systems Engineering*, pages 75–90. Springer, 2014.

[AJP13] A. Ahmad, P. Jamshidi, and C. Pahk. A framework for Acquisition and Application of Software Architecture Evolution Knowledge. *ACM SIGSOFT Software Engineering Notes*, 38:1–7, 2013.

[AP94] A. Aamodt and E. Plaza. Case-based reasoning: Foundational issues, methodological variations, and system approaches. *AI communications*, 7(1):39–59, 1994.

[ARSL14] V. Andrikopoulos, A. Reuter, S. G. Sáez, and F. Leymann. A GENTL Approach for Cloud Application Topologies. In *Service-Oriented and Cloud Computing*, pages 148–159. Springer, 2014.

[BBKL14a] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. TOSCA: Portable Automated Deployment and Management of Cloud Applications. In *Advanced Web Services*, pages 527–549. Springer, New York, January 2014.

[BBKL14b] U. Breitenbücher, T. Binz, O. Kopp, and F. Leymann. Vinothek-A Self-Service Portal for TOSCA. In *ZEUS*, pages 69–72, 2014.

[BGPCV12] L. Badger, T. Grance, R. Patt-Corner, and J. Voas. Cloud Computing Synopsis and Recommendations, 2012.

[BINF12] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *Proceedings of the 34th International Conference on Software Engineering*, pages 419–429. IEEE Press, 2012.

[BRS+01] R. Bergmann, M. M. Richter, S. Schmitt, A. Stahl, and I. Vollrath. Utility-oriented matching: A new research direction for case-based reasoning. In *Professionelles Wissensmanagement: Erfahrungen und Visionen. Proceedings of the 1st Conference on Professional Knowledge Management. Shaker*, 2001.

[CS93] M. Calzarossa and G. Serazzi. Workload characterization: A survey. *Proceedings of the IEEE*, 81(8):1136–1150, 1993.

[CS09] A. Chhabra and G. Singh. Knowledge-Based Modeling Approach for Performance Measurement of Parallel Systems. *International Arab Journal of Information Technology (IAJIT)*, 6(1), 2009.

[Din16] H. Ding. Persistence and Discovery of Reusable Cloud Application Topologies. Master's thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, 2016.

[DKVR09] X. Dimitropoulos, D. Krioukov, A. Vahdat, and G. Riley. Graph annotations in modeling complex network topologies. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 19(4):17, 2009.

[Fie00] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter. *Cloud Computing Patterns*. Springer, 2014.

[FRL13] S. Frey, C. Reich, and C. Lüthje. Key performance indicators for cloud computing SLAs. In *The Fifth International Conference on Emerging Network Intelligence, EMERGING*, pages 60–64, 2013.

[FZRL08] I. Foster, Y. Zhao, I. Raicu, and S. Lu. Cloud computing and grid computing 360-degree compared. In *Grid Computing Environments Workshop, 2008. GCE'08*, pages 1–10. Ieee, 2008.

[GALS14] S. Gómez, V. Andrikopoulos, F. Leymann, and S. Strauch. Design Support for Performance Aware Dynamic Application (Re-)Distribution in the Cloud. *IEEE Transactions on Services Computing*, 8(2):225–239, December 2014.

[Gan15] K. Ganguly. Performance Aware Cloud Application Topology Enrichment. Master's thesis, University of Stuttgart, Faculty of Computer Science, Electrical Engineering, and Information Technology, 2015.

[GG15] T. Gabel and E. Godehardt. Top-Down Induction of Similarity Measures Using Similarity Clouds. In *Case-Based Reasoning Research and Development*, pages 149–164. Springer, 2015.

[GLZ+10] C. Gong, J. Liu, Q. Zhang, H. Chen, and Z. Gong. The characteristics of cloud computing. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 275–279. IEEE, 2010.

[GVB11] S. K. Garg, S. Versteeg, and R. Buyya. SMICloud: A framework for comparing and ranking cloud services. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 210–218. IEEE, 2011.

[HZ11] N. Hurley and M. Zhang. Novelty and diversity in top-n recommendation–analysis and evaluation. *ACM Transactions on Internet Technology (TOIT)*, 10(4):14, 2011.

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann. Winery–a modeling tool for TOSCA-based cloud applications. In *Service-Oriented Computing*, pages 700–704. Springer, 2013.

[LZM98] T. W. Liao, Z. Zhang, and C. R. Mount. Similarity measures for retrieval in case-based reasoning systems. *Applied Artificial Intelligence*, 12(4):267–288, 1998.

[Mas11] M. Masse. *REST API design rulebook*. " O'Reilly Media, Inc.", 2011.

[MB02] B. Mougouie and R. Bergmann. Similarity assessment for generalizied cases by optimization methods. In *Advances in Case-Based Reasoning*, pages 249–263. Springer, 2002.

[MC11] J. M. Merigó and M. Casanovas. A new Minkowski distance based on induced aggregation operators. *International Journal of Computational Intelligence Systems*, 4(2):123–133, 2011.

[NLPVDH12] D. K. Nguyen, F. Lelli, M. P. Papazoglou, and W.-J. Van Den Heuvel. Blueprinting approach in support of cloud computing. *Future Internet*, 4(1):322–346, 2012.

[PvdH11] M. P. Papazoglou and W.-J. van den Heuvel. Blueprinting the cloud. *IEEE Internet Computing*, (6):74–79, 2011.

[RAS+14] R. G. Rocha, R. R. Azevedo, Y. C. Sousa, E. d. A. Tavares, and S. Meira. A case-based reasoning system to support the global software development. *Procedia Computer Science*, 35:194–202, 2014.

[Reu13] A. Reuter. *An extensible application topology definition and annotation framework*. PhD thesis, University of Stuttgart, 2013.

[RR08] L. Richardson and S. Ruby. *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[RRV15] A. Ramírez, J. Romero, and S. Ventura. An approach for the evolutionary discovery of software architectures. *Information Sciences*, 305:234–255, 2015.

[She03] M. Shepperd. Case-based reasoning and software engineering. In *Managing Software Engineering Knowledge*, pages 181–198. Springer, 2003.

[Sta03] A. Stahl. *Learning of knowledge-intensive similarity measures in case-based reasoning*. PhD thesis, Universität Kaiserslautern, 2003.

[top13] Topology and Orchestration Specification for Cloud Applications Version 1.0. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html`, November 2013.

[UPVS09] G. Urdaneta, G. Pierre, and M. Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, 2009.

[uRHH14] Z. ur Rehman, O. K. Hussain, and F. K. Hussain. Parallel cloud service selection and ranking based on QoS history. *International Journal of Parallel Programming*, 42(5):820–852, 2014.

[wik15] Wikipedia Statistics German. `https://stats.wikimedia.org/EN/TablesWikipediaDE.htm`, 2015.

[WL01] S. Weibelzahl and C. U. Lauer. Framework for the evaluation of adaptive CBR-systems. In *Proceedings of the 9th German Workshop on Case-Based Reasoning*, pages 254–263. Citeseer, 2001.

[XAo13] M. Xiu, V. Andrikopoulos, and others. The Nefolog & MiDSuS Systems for Cloud Migration Support. *Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Technical Report*, 8, 2013.

[ZCB10] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.

All links were last followed on January 28, 2016

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, January 28, 2016 ______________________________

Jhonny Vladimir Pincay Nieves