FCTUC DEPARTAMENTO DE ENGENHARIA INFORMÁTICA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Master's Degree in Computer Science
Dissertation
Final Report

CloudAid2

Daniel Alberto Guedes Barrigas
barrigas@student.dei.uc.pt

Advisors:
Dr. Jorge Cardoso
Dra. Catarina Ferreira da Silva
Dr. Paulo Melo

Abstract

Since its appearance, cloud services and its market have grown remarkably. Companies have begun to consider cloud services as a real solution to their problems although, with the market growth and the appearance of an enormous quantity of cloud service providers (CSP), when one is confronted with the situation where he/she has to choose a Cloud service, discovering an ideal solution is not an easy task. Questions like *"Which is the best service for my situation?"* or "*I know what I need but there's too many available options, which one should I choose?"* are hard to solve since it's no longer a choice solely based on the price of a service. Organizations nowadays have many different types of requirements that need to be met so they can offer a product that matches the consumer's needs. It becomes an even harder problem to solve when one is in need of a system composed of multiple services, each with different requirements.

In this thesis I present an augmented version of the CloudAid1 prototype, a recommendation system that provides a suitable aggregation of cloud services given a set of requirements. This new version of the prototype aims at overcoming the limitations of the previous prototype, recurring to new *Multi-Criteria Decision Methods* (ELECTRE III, PROMETHEE I and SMAA-2) for higher accuracy and realism on the ordering of the decision problem's results and, a new graph-theory based aggregation algorithm to deal with *incomparability/incomplete information* between alternatives.

It also presents the Linked USDL Pricing API, a Java API that provides an abstraction layer between developers and the semantic models (Linked USDL core and Linked USDL pricing) thus, promoting and improving their usability. Combining the API with *scrapping/parsing* techniques enabled the creation of the *ServiceGatherer*, a simple application that supports the CloudAid2 prototype by providing semantic descriptions, of real services, for processing.

Keywords: Cloud services, Multi-Criteria Decision Methods, Multi-Criteria Decision Analysis, Cloud service aggregation, cloud services recommendation, Linked USDL service description, Dynamic pricing

Contents

List of Figures

List of Tables

1. Introduction

This chapter provides an overall perspective of the project. It's divided in 6 sections: section 1.1 and 1.2 start by situating our project and explaining the main reasons for its creation. Section 1.3 describes the posed problem; section 1.4 presents the approach adopted on [6, 7] to tackle the challenge of recommending an adequate aggregation of cloud services. Section 1.5 presents the main goals of this thesis and finally, on section 1.6, I present a description of the structure of this document.

1.1 Scope

With the rise of the popularity of cloud services, companies have begun to consider outsourcing their IT systems to reduce the complexity and cost of their business. The increasing popularity of such approach lead to the appearance of new service providers and consequently, an increase on the number of available cloud services in the market. This high level of diversity leads, in turn, to the appearance of new service functionalities, the *pay-per-use* models and many other features that companies need to make their product stand out from the competition and, at the same time, go towards the user's needs [9]. Consequently, when a decision maker is confronted with the situation of deciding which service or services to choose, he/she has great difficulty assessing the pros and cons of each possible alternative.

This is where the CloudAid comes in. The CloudAid1 prototype was developed with the purpose of aiding the decision maker with this kind of situation. It's a recommendation system that makes use of semantic web, Multi-Criteria Decision Analysis (MCDA) and algorithmic techniques to assist the decision maker, recommending aggregations of cloud services based on his preferences.

1.2 Motivation

According to a study made by IBM [10], organizations are gaining a competitive advantage by integrating cloud services in their businesses. They're reporting almost double revenue growth and nearly 2.5 times higher profit than companies that are still considering or reluctant about cloud computing. Furthermore, 25% of the companies saw a reduction in IT costs, 55% saw an increase on efficiency and 49% an improvement in employee mobility.

Considering the advantages that come with the integration of cloud services and the complexity involved in the decision making process, we hope to deliver these tools and methods as a mean of aiding decision makers dealing with this situation.

Given the current state of the prototype, several changes are required for it to evolve and become a step closer to the state of a product that can be delivered with confidence to the people who need it. Achieving this goal is the main objective of this thesis.

1.3 Problem description

As the number of cloud services and cloud providers increases, manually browsing the web looking for services that match our needs and then aggregating said services will become almost impossible. For example, imagine that you're the Chief Technology Officer

(CTO) of a relatively new company and that you're responsible for deciding/choosing the components for your company's main product: an online trading platform where users can buy/sell/trade any item they want. You'll need 5 different classes of cloud services: a Load Balancer, a Database, a Back-up Recovery system, a payment gateway and a Server. The first step is to elaborate a set of pre-defined requirements for each class. Following, manually browse the web looking for cloud services (based on the pre-defined requirements) that can be used to create and host the application. Once you've found a list of possible cloud services, the third step consists in identifying the best services from the list of services, using the decision maker's preferences (e.g.: *security is more important than the cost* or, *back-up reliability is more important than processing power*). Finally, the fourth step consists in generating possible aggregation of services (e.g.: using a spreadsheet application like Excel to organize and analyze the data) and, discarding aggregations that don't meet the necessary requirements. This is a manual, time consuming and prone to errors approach. To address this issue, the CloudAid1 prototype was developed.

The CloudAid1 prototype is the work of a previous thesis [6, 7] whose main objective was to provide a set of tools to assist the decision maker choosing a set cloud services. By gathering the details of his preferences and the requirements of the services he's looking for, the CloudAid1 prototype is capable of recommending an adequate aggregation of cloud services. While the results obtained by the CloudAid1 prototype are promising, there are several aspects that need to be improved. As such, this thesis objective is to augment the current CloudAid1 prototype, adding several new features to its modules and improving some of its current functionalities.

1.4 CloudAid – Approach and techniques

The CloudAid1 prototype applies techniques from several areas to achieve the desired aggregations of cloud services. This section's purpose is to give an introduction to the key elements of the CloudAid1 prototype and the main technologies supporting it.

There are three principle aspects that are the key components of the prototype [9], each supported by one or more techniques/technologies:

1. **Service description** – Tackles the problem of how to model and structure the cloud services data. To achieve this, Eng. Jorge Araújo adopted Linked USDL [37], a semantic service description language based in USDL [38, 39]. It's the most comprehensive approach to support the description of real-world services and supports the creation of structured descriptions covering the most relevant characteristics, ranging from technical aspects, to socio-economic concerns or even legal issues [17]. Due to the lack of support in the description of cloud services using a semantic web approach, the need for a taxonomy to map the most common characteristics of cloud services arose. Thus, on [6, 7] was created the *Cloud Taxonomy*, a domain specific ontology which captures cloud service characteristics was developed. Recurring to the *Cloud Taxonomy* and to the Linked USDL, [6, and 7] successfully describes and standardizes cloud services information.

2. **Service selection** – Achieved the desired service description, it's necessary to decide which of the services, based on a set of defined requirements, have more

relevance to the decision maker. Service selection is directly related with this problem. To provide an automated method to answer this question, [6, and 7] opted for a Multi-Criteria Decision Analysis approach (MCDA). MCDA is a discipline that explicitly considers multiple criteria in decision-making environments which, in our case, fits perfectly. Multi-Criteria Decision Methods (MCDM) were integrated into the prototype to obtain ordered lists of alternatives (on the new version of the prototype, these lists are replaced with graphs); these lists are later used to create the desired aggregation of services.

3. **Service aggregation** – This subject is related to the creation of the desired aggregation of services. Taking advantage of the ranked lists created by the MCDM, Eng. Jorge Araújo applied the notion of combinatory tree to generate all possible combinations and developed two algorithms (section 4.5.2) based on *breadth-first search* and *branch-and-bound* techniques to transverse the tree and create the set of admissible aggregated solutions [7,9]. After studying and analyzing these algorithms, a new aggregation algorithm was developed for this version of the prototype (Section 5).

1.5 Objectives and Challenges

This internship proposes challenges at both technical and research levels. It represents the transition from an academic environment into a professional stage and my insertion on a team whose purpose is to provide a set of tools to aid decision makers choosing a proper aggregation of cloud services. It requires an adaptive and responsive attitude from the intern to the problems at hand and the adoption of an engineering perspective to smooth/monitor the progression of the project.

Given the broad scope of the project, the objectives were classified as either Primary or Secondary objectives. Primary objectives are the main focus of the project and where highest effort/time was spent. Objectives classified as Primary possess a higher degree of interest from a scientific point of view. Secondary objectives, on the other hand, are meant to support the Primary objectives or improve the current state of the prototype.

This section presents the main objectives and challenges of this thesis:

1. **[Primary]** – *Dynamic price calculation*: Development of a friendly Java API that enables the use of the Linked USDL core and Linked USDL pricing (created on [6, 7]) models programmatically.
    a. **Challenge** – Detailed analysis of multiple pricing models adopted by different service providers and perform a conceptual/practical validation of both model and API by modeling said services (and their pricing methods).

2. **[Primary]** – *New aggregation algorithm*: The results of the new *MCDM* are different from those used on the earlier version of the prototype (which used *ranked lists*). The new *MCDM* results consider the possibility of incomparability/incomplete information between alternatives thus, a new algorithm needs to be developed to deal with this new characteristic. Section

5.3.3.3 provides a detailed description of these new results and Section 5.4 describes the new aggregation algorithm.

   a. **Challenge** – Study and analysis of the old aggregation algorithms and the problem at hand. Based on this analysis, develop and test a new aggregation algorithm.

3. **[Secondary]** – *Develop a friendly and ergonomic visual graphic interface*: The current graphic interface of the CloudAid1 prototype is textual. It's intended to apply ergonomic design guidelines to create a visual graphic interface that'll facilitate the interaction and usability of the prototype.
   a. **Challenge** – Study an adequate technology to help me create a proper graphic interface for the prototype and apply the necessary changes to the current architecture of the CloudAid1 system.

4. **[Secondary]** – *Apply scrape/parsing techniques to build a cloud services repository*: The repository created on [6, 7] is but a semi-real representation of cloud services. The version of the prototype has a new service repository created with data extracted from real Cloud services.
   a. **Challenge** – Study the provider's services/websites in order to create a *scrapper/JSON Parser* that extracts and structures the service's information using the Linked USDL Pricing API [50].

5. **[Secondary]** – *Integration of new multi-criteria decision methods*: Add new multi-criteria decision methods that consider incomplete information/incomparability.
   a. **Challenge** – Study and inclusion of new *MCDM* into the current prototype.

This concludes the introduction of the work and what's expected from it. Next, the structure of this document is presented.

1.6 Document structure

In this section I present a brief description of the structure of the document and what is discussed/presented on each of its chapters.

- **Chapter 1** – Introduction to the internship, its scope, motivation and main objectives.
- **Chapter 2** – Presents the software engineering methodologies followed to manage and control the development of the product.
- **Chapter 3** – Reviews related literature, comparing and contrasting it to our work.
- **Chapter 4** – Presents a brief analysis of the CloudAid1 prototype.
- **Chapter 5** – Describes in detail the new functionalities/modifications of the CloudAid2 prototype, how they were implemented and introduces other initiatives that emerged during the prototype's development.

- **Chapter 6** – Presents the testing devised to verify the validity of the CloudAid2 prototype.
- **Chapter 7** – Makes a summary of this thesis work, discussing overall findings, future development and research potential.

2. Project Management

Project management is a discipline that centers on planning, motivating, controlling and organize resources to achieve specific goals [18]. To manage and control the development of the new prototype, it was necessary to adopt a project management methodology.

This section handles the description of the adopted project management methodology (section 2.1), presents the team members (section 2.2) and the devised working schedule (section 2.3).

2.1 Methodology

The adoption of a project management methodology is imperative. Since I had the liberty to choose whatever methodology I felt more comfortable with and given the proposed project schedule specified on [19], the decision fell upon the Royce's final model [20, 40].

Waterfall model is sequential process, adopted and modified from the construction model of production companies, which progresses steadily through the following stages: Analysis, Design, Implementation/Production, Testing/Verification and finally, Maintenance. Royce's final model (Fig.1) is an intended improvement of the initial waterfall model which illustrates that feedback can lead to modifications on the work done on previous stages. For example, feedback from the testing phase can lead back to the design phase due to some flaws encountered on the design of the product [20, 40]. It's also recommended to try to involve the customer as much as possible, keeping them informed on the progress of the project and other relevant information [20]. As such, and given that the team members are also the clients on this project, regular meetings were scheduled, more precisely, at least one per week. In each meeting, we discussed the work done throughout the week, findings and debated solutions to the problems at hand. After each meeting, a report about what was discussed, decisions, and other relevant information, was written. This report also included the tasks to be done on the following week(s).

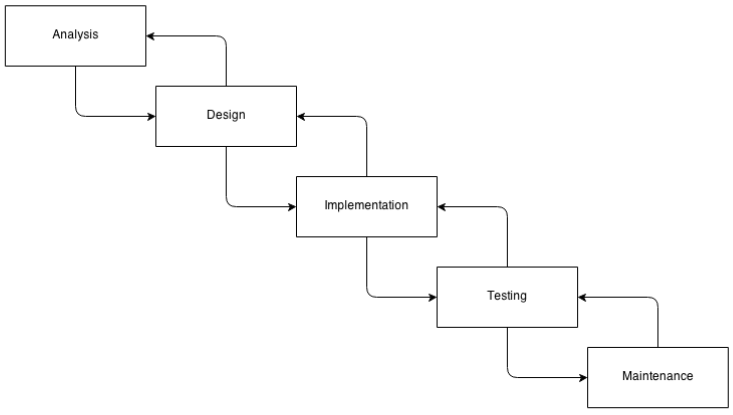


Fig. 1- Royce's final model

2.2 Team

This section presents the members that compose the team responsible for the development of the CloudAid2 project:

- Daniel Alberto Guedes Barrigas – Intern responsible for the development of the CloudAid2 prototype.

- Prof. Jorge Cardoso – Is Associate Professor and joined the Information System Group at the University of Coimbra in 2009. He's a Guest Professor at the Karlsruhe Institute of Technology (KIT). CloudAid2 Advisor, responsible for the validation of the product as well as the documentation produced along with it.

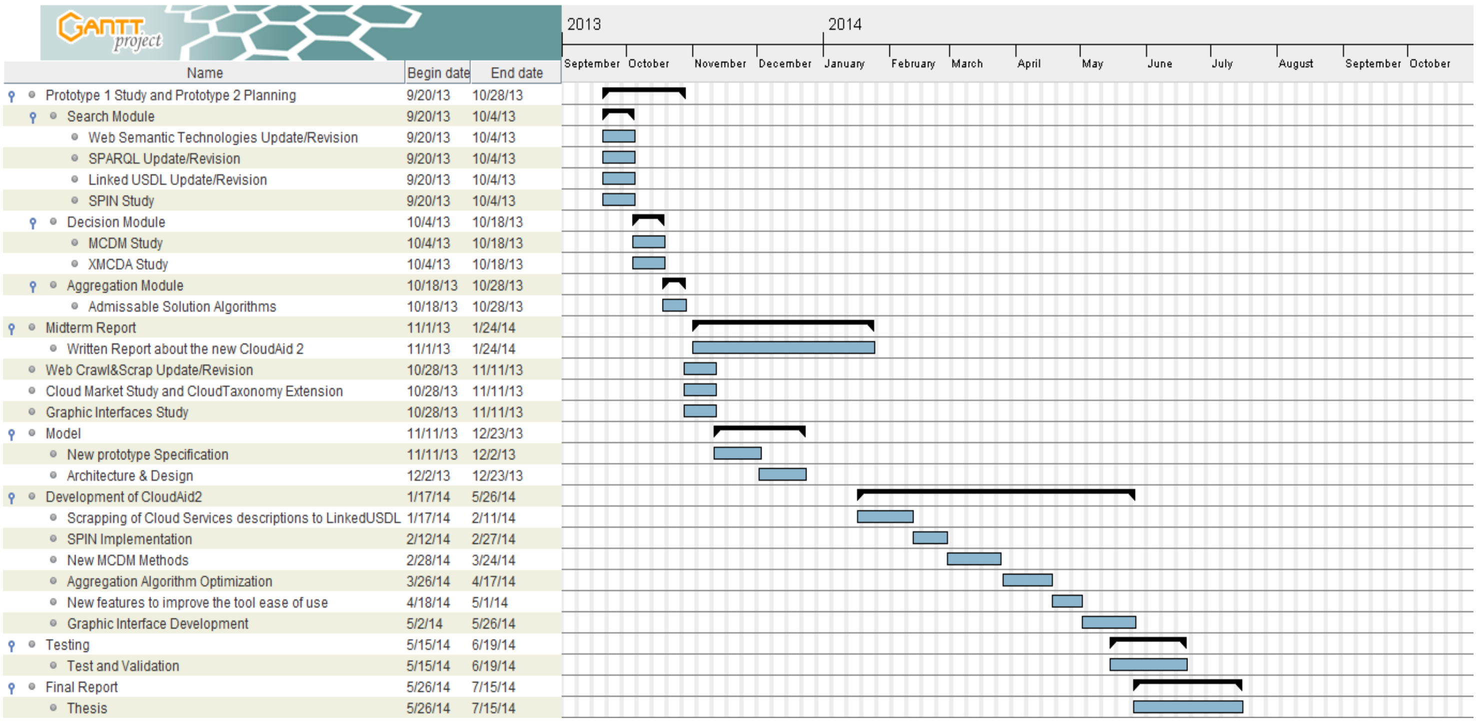
- Prof. Catarina Ferreira da Silva – Is Associate Professor at the Computer Science Department of the Institute of Technology of the Claude Bernard Lyon 1 University, and joined the Service Oriented Computing team of the Research Center of Images and Intelligent Information Systems (France) in 2012. She's also a member of the Information System Group of the Centre for Informatics and Systems of the University of Coimbra (Portugal) since 2009. CloudAid2 Advisor, responsible for the validation of the product as well as the documentation produced along with it.

- Prof. Paulo Melo – Is Associate Professor at the Faculty of Economics, University of Coimbra where he obtained his PhD in Management Science. He has a Bachelor in Computer Science and a Master's Degree in Systems and Automation. CloudAid2 Advisor, responsible for the validation of the product as well as the documentation produced along with it.

2.3 Planning

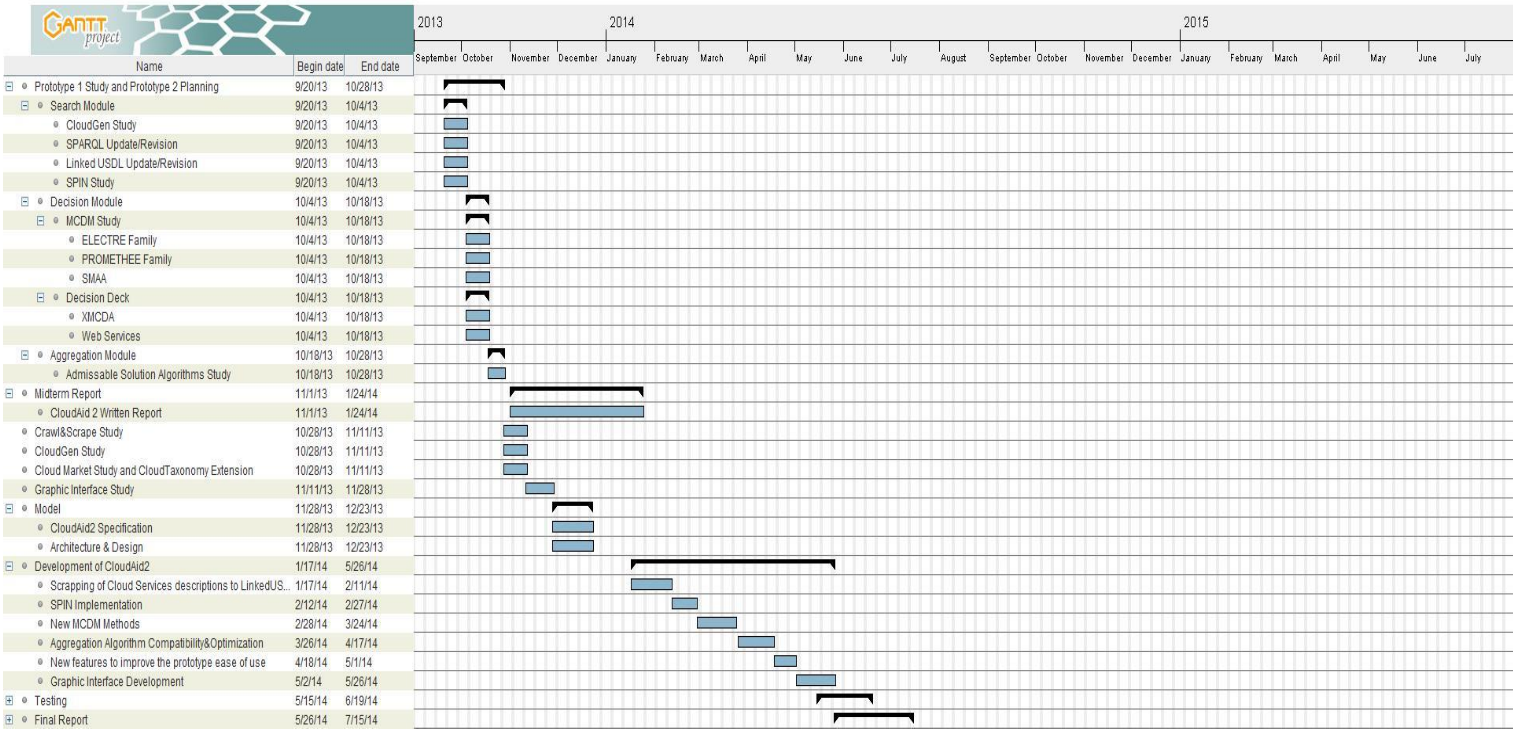
To keep track of the progress of the project and maintain a proper perspective on the work ahead, a working plan was designed. To help me design the plan, I recurred to a Gantt diagram.

This section presents three versions of the Gantt diagram: an initial plan (Fig.2), a mid-term plan (Fig.3) and a final plan (Fig.4). The later version derived from the initial plan which was modified as the project moved forward, my knowledge about the CloudAid1 (and related topics) deepened and, as the development of the new prototype progressed.



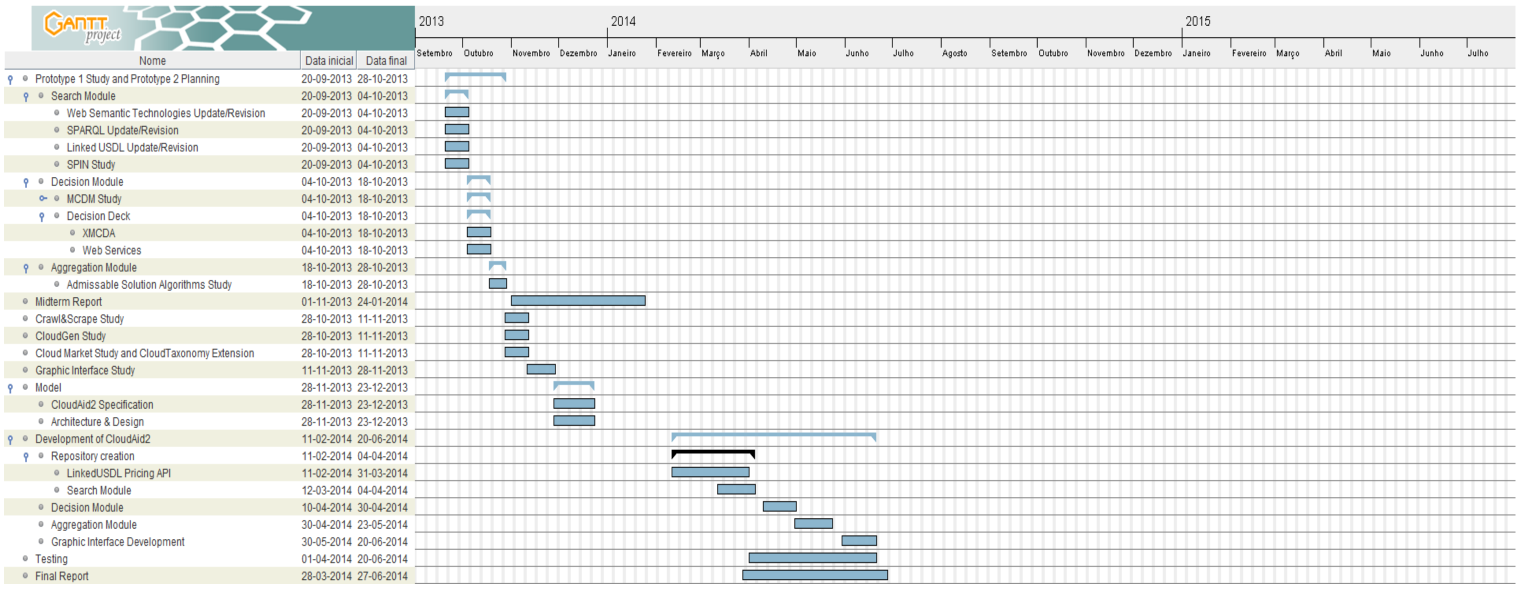
Name	Begin date	End date
Prototype 1 Study and Prototype 2 Planning	9/20/13	10/28/13
Search Module	9/20/13	10/4/13
Web Semantic Technologies Update/Revision	9/20/13	10/4/13
SPARQL Update/Revision	9/20/13	10/4/13
Linked USDL Update/Revision	9/20/13	10/4/13
SPIN Study	9/20/13	10/4/13
Decision Module	10/4/13	10/18/13
MCDM Study	10/4/13	10/18/13
XMCDA Study	10/4/13	10/18/13
Aggregation Module	10/18/13	10/28/13
Admissable Solution Algorithms	10/18/13	10/28/13
Midterm Report	11/1/13	1/24/14
Written Report about the new CloudAid 2	11/1/13	1/24/14
Web Crawl&Scrap Update/Revision	10/28/13	11/11/13
Cloud Market Study and CloudTaxonomy Extension	10/28/13	11/11/13
Graphic Interfaces Study	10/28/13	11/11/13
Model	11/11/13	12/23/13
New prototype Specification	11/11/13	12/2/13
Architecture & Design	12/2/13	12/23/13
Development of CloudAid2	1/17/14	5/26/14
Scrapping of Cloud Services descriptions to LinkedUSDL	1/17/14	2/11/14
SPIN Implementation	2/12/14	2/27/14
New MCDM Methods	2/28/14	3/24/14
Aggregation Algorithm Optimization	3/26/14	4/17/14
New features to improve the tool ease of use	4/18/14	5/1/14
Graphic Interface Development	5/2/14	5/26/14
Testing	5/15/14	6/19/14
Test and Validation	5/15/14	6/19/14
Final Report	5/26/14	7/15/14
Thesis	5/26/14	7/15/14

Fig. 2 - Initial Gantt Diagram



Name	Begin date	End date
Prototype 1 Study and Prototype 2 Planning	9/20/13	10/28/13
Search Module	9/20/13	10/4/13
CloudGen Study	9/20/13	10/4/13
SPARQL Update/Revision	9/20/13	10/4/13
Linked USDL Update/Revision	9/20/13	10/4/13
SPIN Study	9/20/13	10/4/13
Decision Module	10/4/13	10/18/13
MCDM Study	10/4/13	10/18/13
ELECTRE Family	10/4/13	10/18/13
PROMETHEE Family	10/4/13	10/18/13
SMAA	10/4/13	10/18/13
Decision Deck	10/4/13	10/18/13
XMCDA	10/4/13	10/18/13
Web Services	10/4/13	10/18/13
Aggregation Module	10/18/13	10/28/13
Admissable Solution Algorithms Study	10/18/13	10/28/13
Midterm Report	11/1/13	1/24/14
CloudAid 2 Written Report	11/1/13	1/24/14
Crawl&Scrape Study	10/28/13	11/11/13
CloudGen Study	10/28/13	11/11/13
Cloud Market Study and CloudTaxonomy Extension	10/28/13	11/11/13
Graphic Interface Study	11/11/13	11/28/13
Model	11/28/13	12/23/13
CloudAid2 Specification	11/28/13	12/23/13
Architecture & Design	11/28/13	12/23/13
Development of CloudAid2	1/17/14	5/26/14
Scrapping of Cloud Services descriptions to LinkedUS...	1/17/14	2/11/14
SPIN Implementation	2/12/14	2/27/14
New MCDM Methods	2/28/14	3/24/14
Aggregation Algorithm Compatibility&Optimization	3/26/14	4/17/14
New features to improve the prototype ease of use	4/18/14	5/1/14
Graphic Interface Development	5/2/14	5/26/14
Testing	5/15/14	6/19/14
Final Report	5/26/14	7/15/14

Fig. 3 - Mid-term Gantt Diagram



Nome	Data inicial	Data final
Prototype 1 Study and Prototype 2 Planning	20-09-2013	28-10-2013
Search Module	20-09-2013	04-10-2013
Web Semantic Technologies Update/Revision	20-09-2013	04-10-2013
SPARQL Update/Revision	20-09-2013	04-10-2013
Linked USDL Update/Revision	20-09-2013	04-10-2013
SPIN Study	20-09-2013	04-10-2013
Decision Module	04-10-2013	18-10-2013
MCDM Study	04-10-2013	18-10-2013
Decision Deck	04-10-2013	18-10-2013
XMCDA	04-10-2013	18-10-2013
Web Services	04-10-2013	18-10-2013
Aggregation Module	18-10-2013	28-10-2013
Admissable Solution Algorithms Study	18-10-2013	28-10-2013
Midterm Report	01-11-2013	24-01-2014
Crawl&Scrape Study	28-10-2013	11-11-2013
CloudGen Study	28-10-2013	11-11-2013
Cloud Market Study and CloudTaxonomy Extension	28-10-2013	11-11-2013
Graphic Interface Study	11-11-2013	28-11-2013
Model	28-11-2013	23-12-2013
CloudAid2 Specification	28-11-2013	23-12-2013
Architecture & Design	28-11-2013	23-12-2013
Development of CloudAid2	11-02-2014	20-06-2014
Repository creation	11-02-2014	04-04-2014
LinkedUSDL Pricing API	11-02-2014	31-03-2014
Search Module	12-03-2014	04-04-2014
Decision Module	10-04-2014	30-04-2014
Aggregation Module	30-04-2014	23-05-2014
Graphic Interface Development	30-05-2014	20-06-2014
Testing	01-04-2014	20-06-2014
Final Report	28-03-2014	27-06-2014

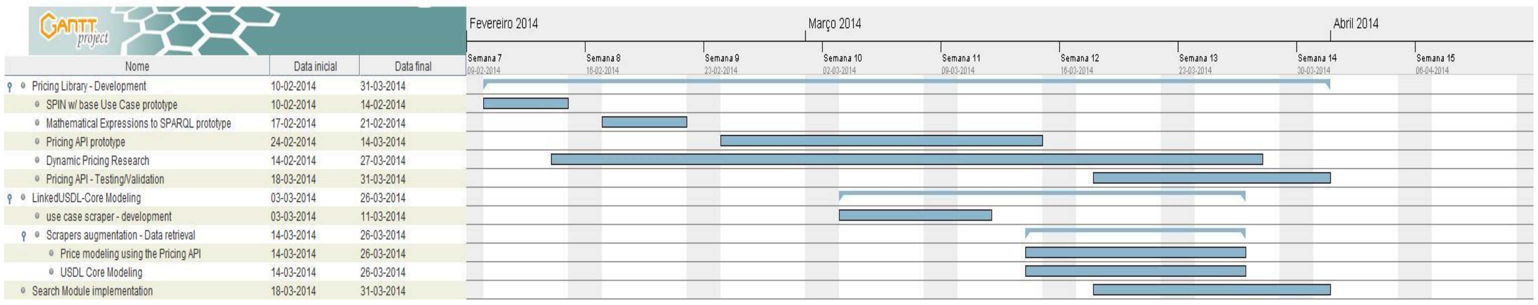
Fig. 4 - Final Gantt Diagram

As shown in the figures, more specific tasks either replaced the old ones or new ones were added, leading to some deviations from the original planning. This is consequence of the study performed along the way or, from the progressive development of the new prototype and the challenges that emerged from it.

The study, analysis and development of the prototype was split among the components of the CloudAid prototype (Fig.4):

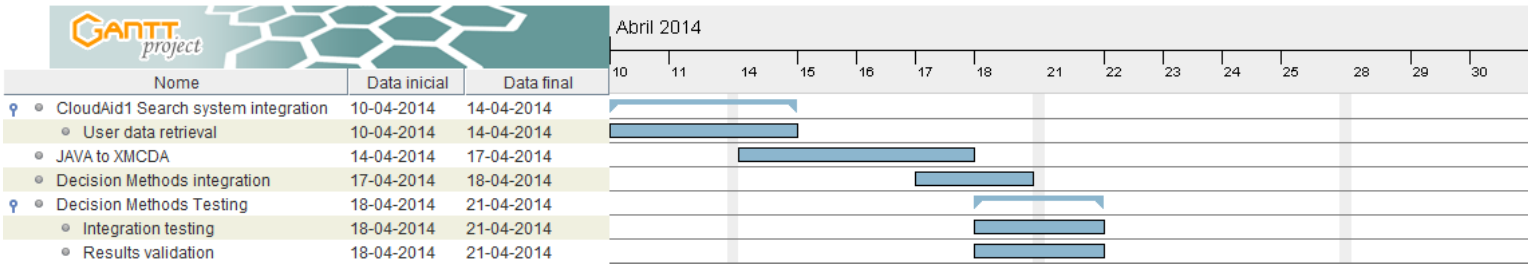
- Repository creation/Search Module
- Decision Module
- Aggregation Module
- Graphic Interface

To simplify the navigation and understanding of the plan, a different Gant Diagram describing the tasks involved on the development of each component was created. Fig.5, Fig.6, Fig.7 and Fig.8 show the planning for each them, respectively.



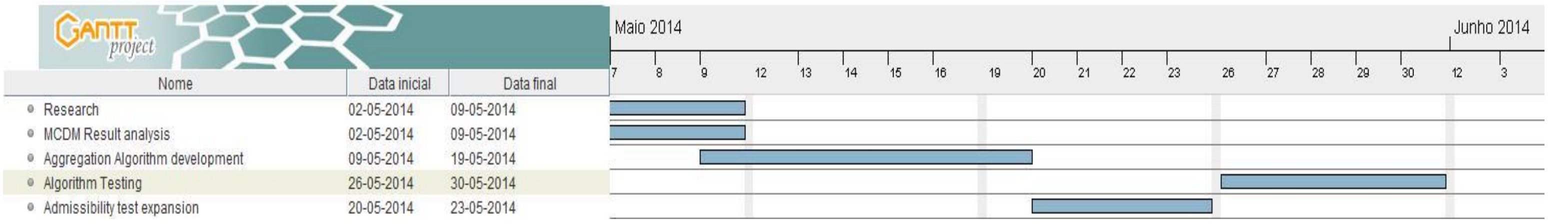
Nome	Data inicial	Data final
Pricing Library - Development	10-02-2014	31-03-2014
SPIN w/ base Use Case prototype	10-02-2014	14-02-2014
Mathematical Expressions to SPARQL prototype	17-02-2014	21-02-2014
Pricing API prototype	24-02-2014	14-03-2014
Dynamic Pricing Research	14-02-2014	27-03-2014
Pricing API - Testing/Validation	18-03-2014	31-03-2014
LinkedUSDL-Core Modeling	03-03-2014	26-03-2014
use case scraper - development	03-03-2014	11-03-2014
Scrapers augmentation - Data retrieval	14-03-2014	26-03-2014
Price modeling using the Pricing API	14-03-2014	26-03-2014
USDL Core Modeling	14-03-2014	26-03-2014
Search Module implementation	18-03-2014	31-03-2014

Fig. 5 - Repository/Search Module development



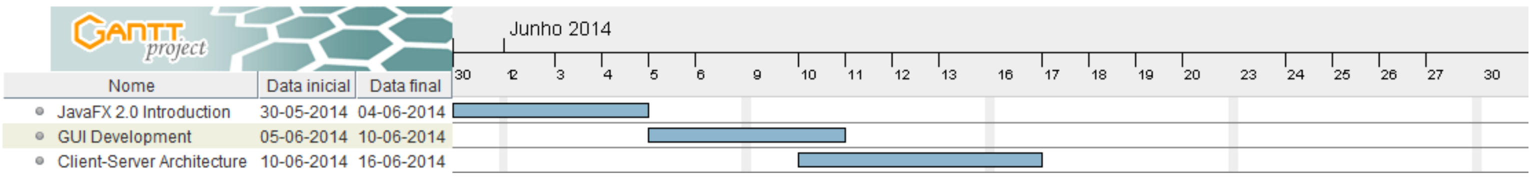
Nome	Data inicial	Data final
CloudAid1 Search system integration	10-04-2014	14-04-2014
User data retrieval	10-04-2014	14-04-2014
JAVA to XMCDA	14-04-2014	17-04-2014
Decision Methods integration	17-04-2014	18-04-2014
Decision Methods Testing	18-04-2014	21-04-2014
Integration testing	18-04-2014	21-04-2014
Results validation	18-04-2014	21-04-2014

Fig. 6 - Decision Module development



Nome	Data inicial	Data final
Research	02-05-2014	09-05-2014
MCDM Result analysis	02-05-2014	09-05-2014
Aggregation Algorithm development	09-05-2014	19-05-2014
Algorithm Testing	26-05-2014	30-05-2014
Admissibility test expansion	20-05-2014	23-05-2014

Fig. 7 - Aggregation Module development



Nome	Data inicial	Data final
JavaFX 2.0 Introduction	30-05-2014	04-06-2014
GUI Development	05-06-2014	10-06-2014
Client-Server Architecture	10-06-2014	16-06-2014

Fig. 8 - Graphic Interface development

The most significant change to the initial plan was due to the inclusion of the Linked USDL Pricing API. Considering the amount of time I had and the time necessary to develop the API, some of the deadlines had to be postponed in order to meet the objectives of this "side" project. Of course, minor deviations from the plan occurred due to the appearance of challenges not only related to the API but also, to the remaining modules of the prototype. While these deviations did happen, focusing on the critical points of the modules and keeping the project's requirements/objectives in mind, I managed to complete the entire development cycle successfully.

3. State of the art

In this section I present a review of literature related to the challenges presented by the CloudAid project. I'll compare and contrast these different approaches with our work, identifying similarities and differences between them, or, I'll describe how their work contributed to the improvement/development of the CloudAid2/Linked USDL Pricing API. The remaining of this section will cover three key scopes of the project:

- Cloud pricing
- Service selection
- Service aggregation

Further comparisons and different approaches about service selection/aggregation can be seen on [6, 7, and 9].

❖ Cloud pricing

The current Cloud market's pricing scene is complex. Pricing methods vary between providers, each with its own characteristics, advantages and disadvantages for both consumer and provider. While every provider has its own pricing method for the services they provide, one can find similarities between them; similarities that can be used to classify/categorize the pricing methods and as such, achieve some degree of coherence/order on the topic.

The authors of [43, 44, 45, and 47] describe some of the most common similarities between pricing methods. Exploiting those similarities, they achieve a similar classification for the pricing methods.

According to [44, 45], every existing method can either be: a Usage/Consumption based method where the customer is charged only by what he/she consumed; a Subscription based method where the user pays a fee (usually monthly) to use the service over a certain period of time or a Market based method where the cost of the service varies according to the status of the market: high demand translates into an increase on the cost of the service whereas, low demand translates into a decrease on the cost said service.

On [43, 70], while the name of the categories under which the methods are supposed to belong to is different, the meaning of each of those categories is very similar to those presented on [44, 45]. According to them, each of the existing pricing methods can be placed under one of the three following categories: Elastic pricing methods, where the user is charged depending on what he/she used (similar to the Usage/Consumption categories identified on [44, 45]); Fixed pricing methods where consumers are charged a fixed fee per month (or any other period) independently of the usage (similar to the Subscription category) and finally, Spot pricing methods where the price of the service depends on the current state of the market/provider's cloud platform (identical to the Market based category).

The issue addressed on [46] is different from the ones addressed by the literature presented so far. It introduces and compares two pricing methods: Instance Based methods where the customer chooses the amount of resources he/she needs and then is billed by the hour, and, *Reserved Pool* methods. It describes the situations in which the customer is more likely to get a better cost/value ratio by choosing one over the other. It captured my interest since it presents a new pricing method that wasn't addressed on the literature presented so far: Reserved Pool pricing. Contrary to the Usage/Fixed pricing methods, with this pricing method the user is buying quantities of resources instead of pre-defined(or custom made) virtual machines. Using these resources, the customer is able to create any number of virtual machines as long as the quantity of resources allocated to the created virtual machines (VM) doesn't exceeds the quantity of bought resources.

Gathering this information, and reading the work performed on [48] by 451Research.com [49], provides a decent picture on the current state of the cloud's pricing scene. From their detailed survey, a taxonomy that classifies every existing method under one of eight possible categories was created: PrePaid VM Access/ Recurrent PrePaid VM Access where the user needs to pre-pay the service in order to use it (Recurrent meaning that the customer commits to pre-paying the cost of the service in arrears, over a certain period of time, instead of pre-paying the full amount); On-Demand where the user is simply charged for what she/he consumed; Reserved Instances which is similar to On-Demand but requires the up-front payment of an initial fee from the customer (in some ways, this fee works as a proof of commitment from the user); Spot Pricing which is the same as the Market-based methods presented earlier; PrePaid Credit access/Recurrent PrePaid Credit which are very similar to the PrePaid VM Access method but, the payment made by the client is transformed into a secondary currency from where usage costs are later debited from; and finally, Recurring Resource Pooling which has already been introduced by [46].

Aside from the 8 category taxonomy, they also presented four different types of bundling. Bundling refers to the practice of combining several products into a single unit and sell that unit for a single price. In any bundled offering, the customer needs to choose from a number of pre-configured bundles whereas, on any unbundled offering, the customer will be charged for every single billable resource. They identified four main bundle types: Fully Bundled, where the customer needs to choose from a number of virtual machines with predefined quantities of CPU, memory, disk and bandwidth (however, bandwidth is usually not chargeable); VM Bundled, where the consumer needs to choose from a number of virtual machines with predefined quantities of CPU, memory and disk, Bandwidth is charged separately; Processor Bundled where customers need to choose from a number of virtual machines with predefined quantities of CPU and memory, Bandwidth and disk are charged separately; and finally, Unbundled where CPU, memory, disk and bandwidth are all charged separately and their quantities are decided by the customer.

Given the volatile nature of the Cloud scene, and in order to ensure that my work addresses the most recent state of the Cloud pricing scene, I decided to narrow down the scope of my research to a maximum of two years old since introducing a pricing method that is no longer in use would bring little value when compared to the introduction and modeling of the ones currently in use. From the work presented on this section, the most detailed and complete work was the survey performed by 451Research.com [49] on [48], The Pricing CODEX. They address and complement the pricing methods described by other literature, giving the current cloud pricing scene a structure.

❖ Service selection

Service selection is the process of selecting the best service from a pool of services; the best being the one that better satisfies the needs of the customer/consumer.

M. Sun, T. Zang, X. Xu, and R. Wang [5] present an innovative consumer-centered selection method based on Analytic Hierarchy Process (AHP) theory. They take advantage of the AHP theory for quantifying qualitative or semi-quantitative service parameters into weight values, which will later be used for the service ranking calculations. The ranking procedure is based on NxN matrixes (one for each defined criteria), where N is the number of alternatives, that represent pair-wise comparisons between alternatives. Combining these matrixes with other calculated parameters, it's possible to calculate a final weight for each possible alternative. It's important to note that the ranking method is also based on the AHP theory. Through this weight value they can identify the alternative that better satisfies the needs of the consumer. An interesting topic is the support for a scenario where the decision maker isn't a single entity but a group with multiple and different preferences. In this case, weights are attributed to each entity of the group and the final results will take into account every preference of each element of the group.

On [3], Hussain and Rehman provide a comparative study involving IaaS (in this case, thirteen different IaaS services) and use MCDM techniques to select the best one based on their performance values (gathered and supplied by a third party monitoring service, *Cloud Harmony Compute Unit*) over 5 pre-defined criteria. The main objective of their work is to prove that different MCDM may lead to different results over the same input of data. To prove this, they based their study on two different types of MCDM: Multi-Attribute Utility Theory methods (MAUT) and Outranking methods.

This work is mainly focused on result analysis, they don't concern themselves with service description/representation nor do they consider an interaction with a possible decision maker, they solely focus on the MCDM components and their results analysis. Seven different methods were studied: Min-Max method, Max-Min method, Compromise Programming, TOPSIS method, ELECTRE II, PROMETHEE I and AHP. This work proves worthwhile since it uses a scenario with real data to prove, as was expected, that different MCDM can lead to different conclusions. The CloudAid2 prototype will support different types of MCDM thus, it's important to be aware of this issue since the user should also be aware of this possibility.

The authors of [4] bring a new and interesting approach to the selection of cloud services. Instead of using the average or the real-time performance of a cloud service (over *C* criteria) to determine its ranking, they take into consideration their dynamic nature which results from elasticity and on-demand provision of computing resources. Using a third party tool to capture services information on *C* criteria (for this study they considered CPU, Memory, I/O operations and Cost) over different periods of time, applying the TOPSIS method to rank the services on each time slot and by finally combining the results, they're able to provide a final ranking of the services. This approach allows for a more reliable cloud service selection than the one achieved following the common approach (taking only in consideration current QoS values), by taking into consideration temporal variations of the QoS parameters.

On [2], a more direct and personalized approach is considered. They use a MACBETH based model to help the decision maker (DM) select the best cloud service based on a set of evaluation criteria and verify its validity on a real life scenario. The model is divided in three steps: first, they structure their data model by identifying the evaluation criteria for the problem at hand and asking the DM to define “neutral” and “good” references for each criteria. Secondly, they use the MACBETH semantic categories to judge the differences in attractiveness between the two levels of performance (neutral and good). Afterwards, they build a value function for each criterion and use the MACBETH weighing procedure to assess the weights for each of the criteria. Finally, when every criteria related data has been set, the third step begins: inserting the performance data of each service and use the MACBETH method to calculate the ranking of the services.

While this approach proves useful and reliable, it requires a high level of interaction with the decision maker. The CloudAid2 and CloudAid1 system aim to diminish this heavy interaction, providing a simple and automatized way to select and aggregate cloud services.

The work on [1] uses a different approach to deal with unquantifiable evaluation criteria of a multi-criteria decision problem. They designed a 9 steps model based on combined fuzzy theory and a modified version of VIKOR method, which deals with various types of incommensurable and conflicting criteria, to provide a valid ranking of the services. This enables the selection of a cloud service using different types of unquantifiable criteria, which we can relate to the qualitative criteria in our work.

While the usage of a fuzzy model proves to be a reliable option, this work follows a different approach: by defining a numerical distance between linguistic terms (e.g.: numerical distance between the concepts “*Very High”* and *“High”*) it is equally capable of handling unquantifiable criteria.

❖ Service Aggregation

Service aggregation is a relatively new concept thus, a low degree of information is to be expected. Despite this lack of information, it’s heavily related to an older concept: service composition [41, 42]. Both aim at creating new services by putting together smaller parts (i.e. services) but, while service composition imposes a partial order of execution and hard technical constrains (i.e. matching input and outputs or dealing with behavior exceptions), service aggregation relaxes these condition(s) and focuses on softer technical aspects. For example, suppose that the customer is looking for a combination of services with a total cost lower than *y*, or for the possible combinations between a database, a back-up service and a load balancer where at least two of the three need to be located in Tokyo. Service aggregation will take into consideration these soft constraints in order to create the desired aggregation of services, discarding every other possible combination that does not comply with them [9].

Regarding the composition of services, [1] describes the challenges and difficulties that arise when trying to create an automated aggregation of services, pointing out that each of these usually possesses a complex description. Description that might recur to provider specific concepts to describe its attributes, increasing even further the complexity of service discovery/selection and composition.

Considering this issue, [2, 3] focus on approaches developed to facilitate and promote an automated aggregation of services.

On [3], Debajyoti Mukhopadhyay and Archana Chougule present twelve approaches created in order to tackle the problem of service discovery. These methodologies are based on existing technologies from several fields, like semantic web or information retrieval, and use/create new data standards (standards like WSDL/UDDI or creating domain specific ontologies) to deal with the highly ambiguous syntax used to describe the services and, at the same time, optimize search results.

Going a step further, [2] presents a reference model for a fully/semi-full automated composition of services, based on WSDL[1], WS-BPEL [2]and WS-CDL[3]. Basically, the reference model is composed of three key entities: the Web Service Orchestrator (WSO), responsible for managing the execution of the application in-house; the Web Service (WS) (agents designed to perform specific job(s)), and the Web Service Choreographer (WSC), which is in charge of monitoring the interaction behaviors between different WSO's (WSO's from different organizations for example). WSO's communicate with the WSC via their observable views. The standards WSDL and WS-BPEL are used to describe and formalize the communications between each of the components while WS-CDL is used to describe the Web Service Choreographer. Regarding the actual composition of the services, it describes two of the most conventional/used approaches when dealing with this kind of situation/problem (Petri-Nets and Process algebra) and how they can be combined with the reference model.

This thesis proposes a similar solution to the ones described on the literature presented so far: it adopts a mature and comprehensive description languages to reduce the complexity and ambiguity of cloud service descriptions and combines the usage of multi-criteria decision methods with an efficient aggregation algorithm, to select and create (in an automatized way) suitable aggregation of services for the customer.

The two aggregation algorithms developed on [9] are a good approach for the problem at hand however, both algorithms aren't capable of dealing with incomparability/incomplete information between alternatives because they rely on an ordered list (best to worst) of alternatives to perform the aggregations. Having an ordered list of alternatives is not always possible. In fact, in multi-criteria decision problems, the higher the number of criteria involved in the problem, the higher is the probability of being presented with a semi-ordered list rather than an ordered one.

The algorithm I propose in this thesis relies on graph theory to structure preferences between alternatives and tackle situations where it's impossible to tell which alternative is better than the other (meaning they're incomparable). Using this approach, the algorithm successfully overcomes this issue thus, achieving more realistic results.

As mentioned earlier, further discussion and analysis of other different approaches can be found on [6, 7].

[1] http://www.w3.org/TR/wsdl
[2] http://en.wikipedia.org/wiki/Business_Process_Execution_Language#WS-BPEL_2.0
[3] http://www.w3.org/TR/ws-cdl-10/

4. CloudAid1

In this chapter I present an overview of the CloudAid1 prototype. I'll present each element of the system, their objective, how they work and interact with each other. This is but a brief explanation of the prototype to provide an overall idea of how the previous version of the prototype works; a more detailed description about it can be found on Appendix B.

Section 4.1 provides an overall view of the system and its components. From section 4.2 to section 4.6 I'll describe each of these component with further detail, presenting the reason why they were created and their objective. This analysis of the CloudAid1 was supported by [6, 7] and a personal study of the prototype.

4.1 Overall Architecture

The CloudAid1 application was built using a Model-View-Controller (MVC) approach. With the help of Fig.9, which was created using the *Fundamental Modeling Concepts* (FMC) notation, we can visualize a high level representation of the internal components of the system.

FMC complements the software-description achieved by UML, providing a set of tools to describe the system's structures, communication channels and internal flow [21]. It enables the description of architectural components leaving the software specification to UML. It's important to note that several diagrams presented on this document don't follow a strict FMC notation in order to provide a higher degree of detail on the system's architecture (while it's not strictly followed they're heavily based on it).

As we can see on Fig.9, the CloudAid1 prototype is composed of by five modules that are coordinated by a sixth model (the *Controller*) whose main objective is to monitor and control the flow of execution, making sure that each of the components receives the data they need to do their job. I'll make a brief introduction for each of the components to provide a global idea of how the system works from the beginning. A more detailed description of the modules will be presented on the following sections.

- **UI** – This component is the 'View' on the MVC model. It's the interface between the user and the application, responsible for capturing/presenting information from/to the user.

- **Controller** – 'Controller' in the MVC model. It's responsible for mediating information transactions between the UI and the rest of the components. It's also responsible for initializing and controlling the system and its execution.

- **Composite Service Architecture** (CSA) [7] **Evaluator** – Its part of the 'Model' in the MVC model. Responsible for evaluating and preparing the CSA data. The data is inserted by the user thus, some measures must be taken in order to guarantee that there are no problems with it.

- **Search Module –** It's part of the 'Model' in the MVC model. Responsible for retrieving alternatives that match the user's requirements from the *TripleStore*.

- **Decision Module –** Component from the 'Model' in the MVC model. Responsible for ranking the alternatives of certain *Service Template*.

- **Aggregation Module –** It's part of the 'Model' in the MVC model. Responsible for computing the aggregated solutions. Aggregated solutions are composed by one alternative from each *Service Template*.

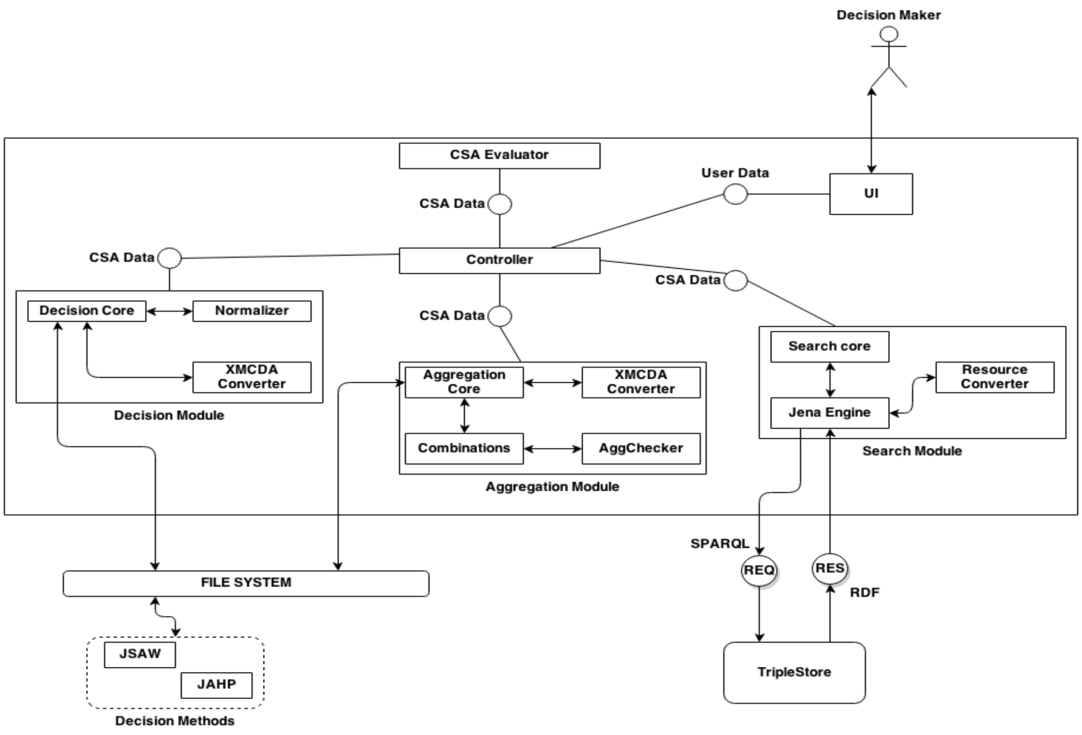


Fig. 9 – CloudAid1 – High Level Architecture

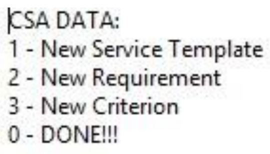
4.2 The Controller and the User Interface (UI)

4.2.1 User Interface

UI stands for *User Interface* and as the name suggests, it's the point of interaction between the user and the application. It's where the user inserts the data required by the application and defines the characteristics he desires for his composite service aggregation (*CSA*). We can categorize the data required by the application on four topics:

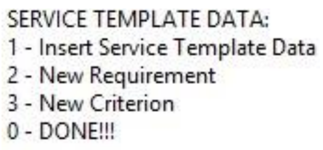
- *Service Template* – Represents an element of the aggregated solution.
- *Requirements* – Features of the solution desired by the decision maker.
- *Criteria* – Parameters used to evaluate the alternatives.
- *Preferences* – The decision maker's preferences regarding the criteria.

Let's see some screenshots of the current UI and how this data is captured by the CloudAid1 prototype.



```
CSA DATA:
1 - New Service Template
2 - New Requirement
3 - New Criterion
0 - DONE!!!
```

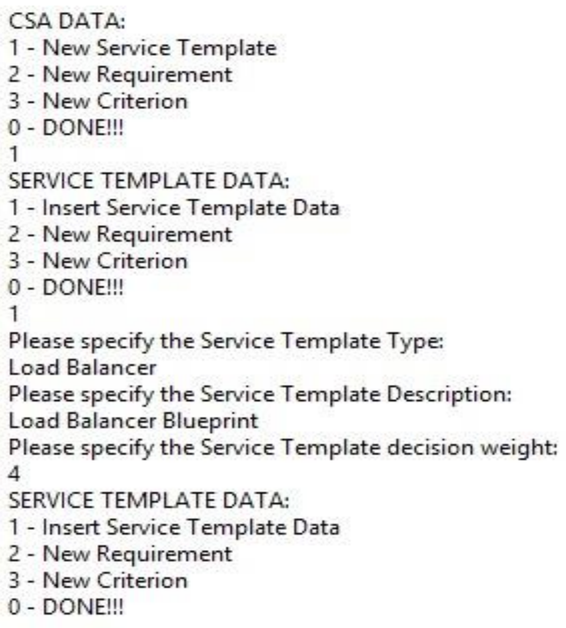
Fig. 10 - CloudAid1 - CSA Menu



```
SERVICE TEMPLATE DATA:
1 - Insert Service Template Data
2 - New Requirement
3 - New Criterion
0 - DONE!!!
```

Fig. 11 - CloudAid1 - Service Template Menu

Fig.10 shows the menu that is first presented to the user. From here, the user is capable of creating new *Service Templates* (option 1), global Requirements (option 2) and global Criteria (option 3). Fig.11 shows the prompted menu when the user chooses to create a new *Service Template*; from here, the user can insert create its respective Requirements and Criteria. Fig.12 presents the creation of a new *Service Template*.



```
CSA DATA:
1 - New Service Template
2 - New Requirement
3 - New Criterion
0 - DONE!!!
1
SERVICE TEMPLATE DATA:
1 - Insert Service Template Data
2 - New Requirement
3 - New Criterion
0 - DONE!!!
1
Please specify the Service Template Type:
Load Balancer
Please specify the Service Template Description:
Load Balancer Blueprint
Please specify the Service Template decision weight:
4
SERVICE TEMPLATE DATA:
1 - Insert Service Template Data
2 - New Requirement
3 - New Criterion
0 - DONE!!!
```

Fig. 12 - CloudAid1 - New Service Template

4.2.2 Controller

The *Controller* is the main component of the CloudAid 1 prototype. It's responsible for managing the whole system, making sure that every other component receives the data they need and, at the same time, it controls the flow of the execution process. Fig.13 presents a diagram, created using an informal notation, that'll help us get a clearer image of the Controller's objective and the flow of the execution of the CloudAid1 application.

As we can see in the picture, this module acts a data dispatcher for the other modules in a sequential order. After one module has done its job, the flow of execution goes back to the controller which, will initiate the next step in the process. Knowing this, and after a brief analysis of the Fig.13, we can identify the core steps of the execution process:

1. Fetch CSA from the user
2. Evaluate CSA – CSA Evaluator
3. Search for alternatives that match the user's needs – Search Module
4. Rank the alternatives of each Service Template – Decision Module
5. Create aggregated solutions – Aggregation Module

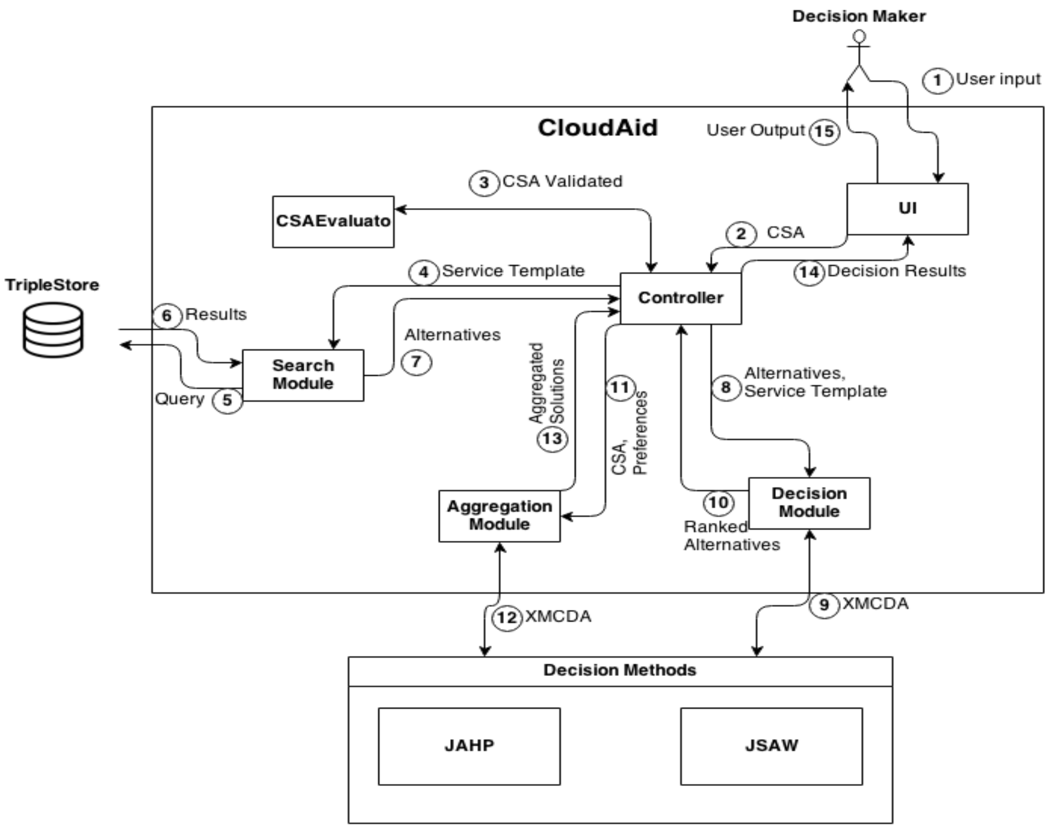


Fig. 13 - CloudAid1 - Controller's flow of execution

It's important to note that every module has its own execution process that will be explained on the following sections but for now, the main objective of this section is to provide an overall perspective of the system and how it works.

Controlling the flow of execution and acting as a data dispatcher for the other modules is but one of the four tasks the controller is in charge of:

1. Environment setup
2. Choose Decision Method
3. Manage the flow of execution
4. Establish a link of communication between the UI and the other modules.

Task 2 and 3 are sequential while task 1 is executed only once and task 4 is executed at the same time as tasks 2 and 3.

- **Environment setup** – This task is responsible for instantiating the rest of the system's components that is, it "creates" the UI followed by the Search Module (Section 4.3), then the Decision Module (Section 4.4) and finally the Aggregation Module (Section 4.5).

- **Choose Decision Method** – The CloudAid1 prototype supports two different Multi-Criteria Decision Methods: Simple-Additive-Weighting (SAW) and Analytic Hierarchic Process (AHP). To decide which of the methods should be used a simple question is asked to the user: "Are you comfortable giving weight to the criteria?" In case he answers 'y' SAW will be used otherwise, AHP will be used.

- **Manage flow of execution** – Once each of the modules has been instantiated (task 1) and the decision method has been chosen (task 2), the application can start its main objective: find aggregated solutions of cloud services that go towards the user's needs. This process starts by evaluating the CSA created by the user (performed by the CSAEvaluator module). If it succeeds, the Controller can start the next stage: Search for alternatives. This step is performed by the Search Module and it searches alternatives for each *ServiceTemplate* on the CSA. Once the search module finishes retrieving the alternatives, it's necessary to rank them according to the defined criteria. This step is performed by the Decision Module. When every Service Templates has its corresponding alternatives ranked, the final step of the application begins: compute the aggregated solutions. This final step is performed by the Aggregation Module.

- **Link of communication** – This is the final responsibility of the controller. The Controller is the communication link between the different modules of the system and, between the UI and the application. Every time a module needs to interact with the user, it calls the Controller who in turn invokes the proper functionality on the UI to communicate with the user. When the user finishes his interaction, the controller is then in charge of transferring the inserted information back to the module. Fig.14 shows how this request is processed at an architectural level.

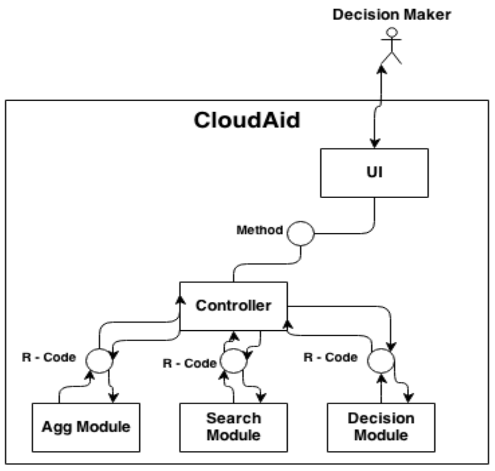


Fig. 14 - CloudAid1 - Model-View Communication

4.3 Search Module

This module is responsible for finding alternatives that match the decision maker preferences. It performs three major tasks:

a. Divide the exclusive requirements from the non-exclusive requirements
b. SPARQL Query construction.
c. Fetch matching alternatives based on the exclusive requirements
d. Enrich the found alternatives with the service offering attributes

The reason a distinction between exclusive and non-exclusive requirements is made is because of the impact they have on the results. Exclusive requirements are those that actually work as a "filter" on the search mechanism since these represent conditions imposed by the decision maker, alternatives that don't obey them will be discarded.

Once we have every *ServiceOffering* resource, we can move to the last step of the module: resource conversion. To ease the following steps of the CloudAid, the information from the semantic model is "loaded" into Java classes which, are later passed onto the following modules. The class responsible for the TTL/RDF to Java conversion is the *ResourceConverter* class. Using the returned resources from the earlier search on the TripleStore, it is able to extract the remaining information (its features and their corresponding values) related to the *ServiceOffering* from the semantic model.
Once these steps have been completed the module's job is done and, all that is left to do is to return the results to the *Controller* in order to proceed with the flow of execution.

Fig.15 presents the module's execution flow from an architectural point of view, showing in an ordered way the steps described on this chapter.

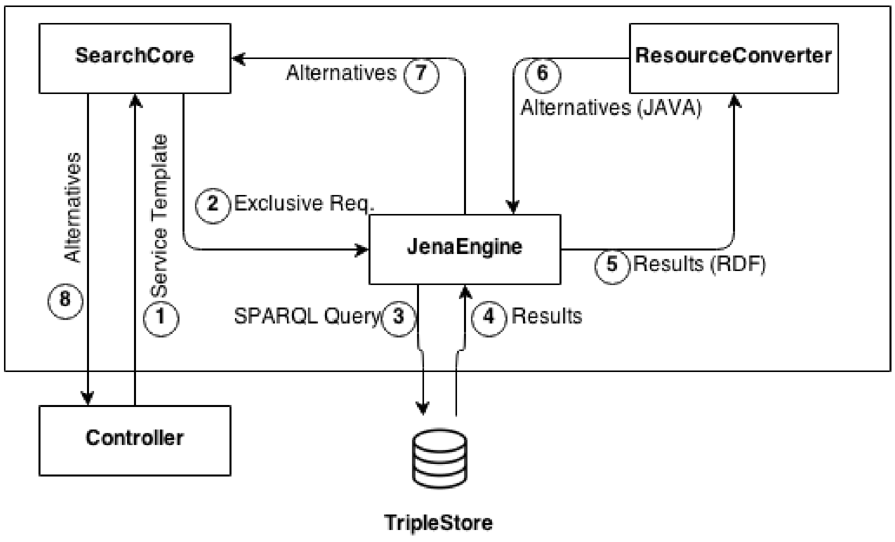


Fig. 15 - CloudAid1 - Search Module

4.3 Decision Module

After retrieving the Service Template's alternatives, it's time to rank them according to the criteria defined by the Decision Maker. Here is where the Decision Module comes in. It retrieves the criteria, the user's preferences and asks other relevant information to produce a valid ranking list of the alternatives for a particular service template. This list has the form of a typical ranked list where the head of the list is the best alternative and the last one the worst. This ranking is accomplished using *Multi-Criteria Decision Methods* (MCDM) that receive data inserted by the Decision Maker, process it, and return the desired ranked list.

To get a correct result from the *MCDM*, it's necessary to normalize the data (section 4.3.1) and then, transform it into XMCDA[4] format (section 4.3.2). Once the problem is in XMCDA format, it's published into a pre-defined directory. External applications that are monitoring the directory will fetch the files that are newly written, calculate the ranking of the alternatives and publish the results on another pre-defined directory. This directory is, in turn, monitored by the Decision Module. It will fetch them as soon as they're created and map the results to Java Objects (section 4.3.3).

We can split the Decision Module in 5 steps:

1. Data normalization – Normalization is needed since data from the Service Template can belong to different intervals, adding undesirable noise and behaviors to the methods calculations.
2. Express the problem in XMCDA format – After normalizing the data, the problem should be described in XMCDA format (section 4.4.2).
3. Publish XMCDA file – Once it's described in XMCDA, it's written into a file on a pre-defined directory for the external *MCDM* application.
4. Read and Transform the Results – After publishing the problem, the module will monitor a pre-defined directory waiting for the results to be published. Once they're published, they'll be read and transformed into Java objects.

[4] http://www.decision-deck.org/xmcda/

5. **Obtain and Sort the ranked List** – The results from the *MCDM* may not be ordered correctly thus, before returning the results onto the Controller, a descending ordering of the list is performed.

Fig.16 describes the complete flow of execution of the Decision Module.

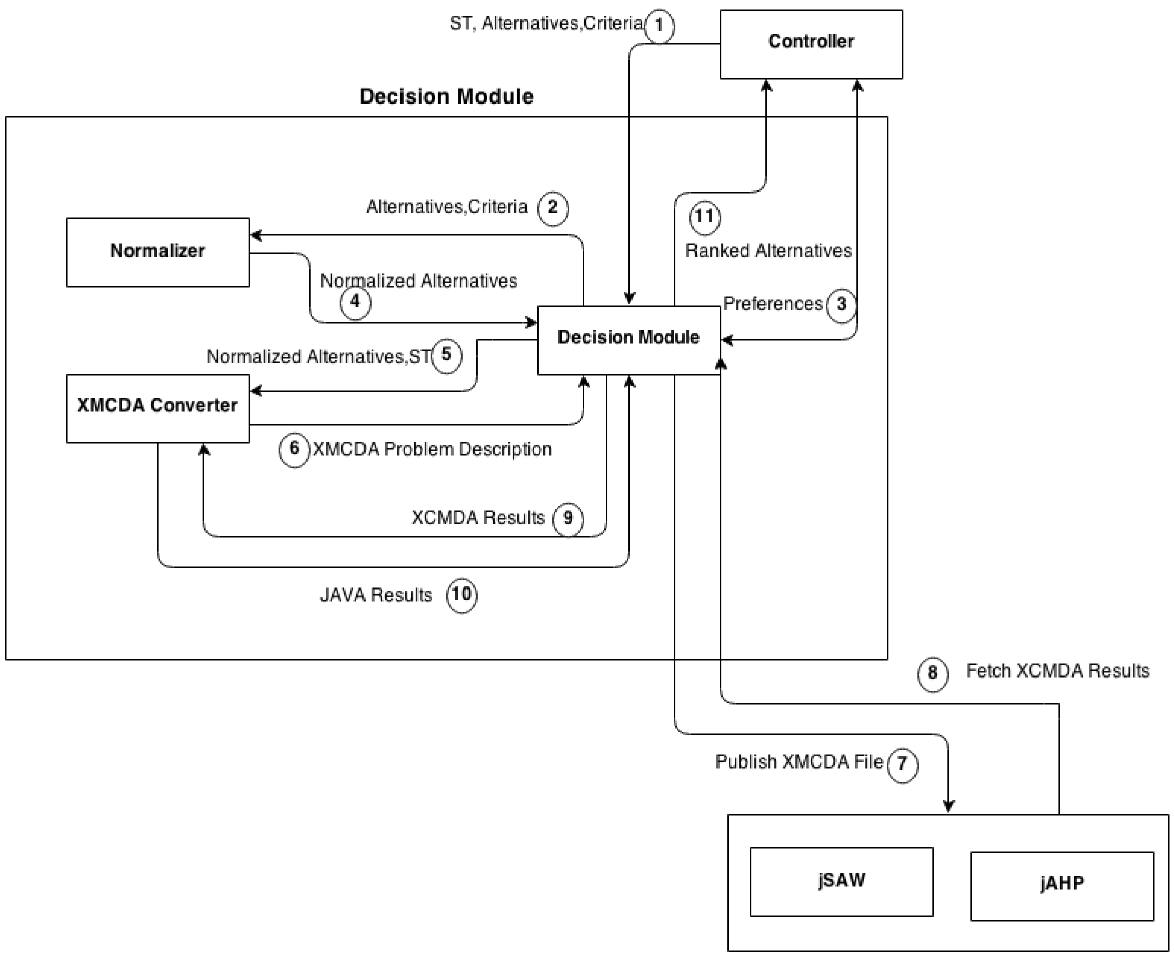


Fig. 16 - CloudAid1 - Decision Module Architectural Flow

4.3.1 Data Normalization

The data normalization process consists in a series of mathematical calculations done by the Java class *Normalizer* to map the data (related to the defined criteria) on the current Service Template into the [0, 1] interval. Let's see an example, imagine we have a Service Template *S* with two alternatives A_1 and A_2, A_1 with attributes [*Price* = 40, *MemorySize = 1024*] and A_2 with [*Price* = 130, *MemorySize* = 4096]. Now suppose the criterions have the following weights: *Price* = 5 and *MemorySize* = 2. With this example we can see that the criteria *MemorySize*, despite having a lower weight than *Price*, will overwhelm the influence of *Price* on the calculations [7]. With the help of the *Normalizer* Java class we can avoid this type of situations.

4.3.2 XML Encoding of Multi-Criteria Decision Aid Data

XML Multi-Criteria Decision Analysis (XMCDA) is an XML based data standard developed by Decision Deck [28, 29] to describe Multi-Criteria Decision problems [23]. A Java library called J-XMCDA is also provided by the Decision Deck to help us create and manipulate information described in XMCDA. Every XMCDA related operation is performed by the Java class *XMCDAConverter* including operations related to the reading and writing of files.

4.3.3 Decision Methods

CloudAid1 uses external applications to solve the multi-criteria decision problem. To establish a communication link between the applications and the prototype, a file based approach was proposed. When in need of the external applications, a file is written on a directory monitored by them. The results obtained by the external applications are transmitted back to the prototype using the same approach but through a different directory. Fig.17 shows a visual representation of these transactions. To assist in this process, the *FileChecker* class was created. This class listens to events on the specified directories and calls the corresponding methods to handle them.

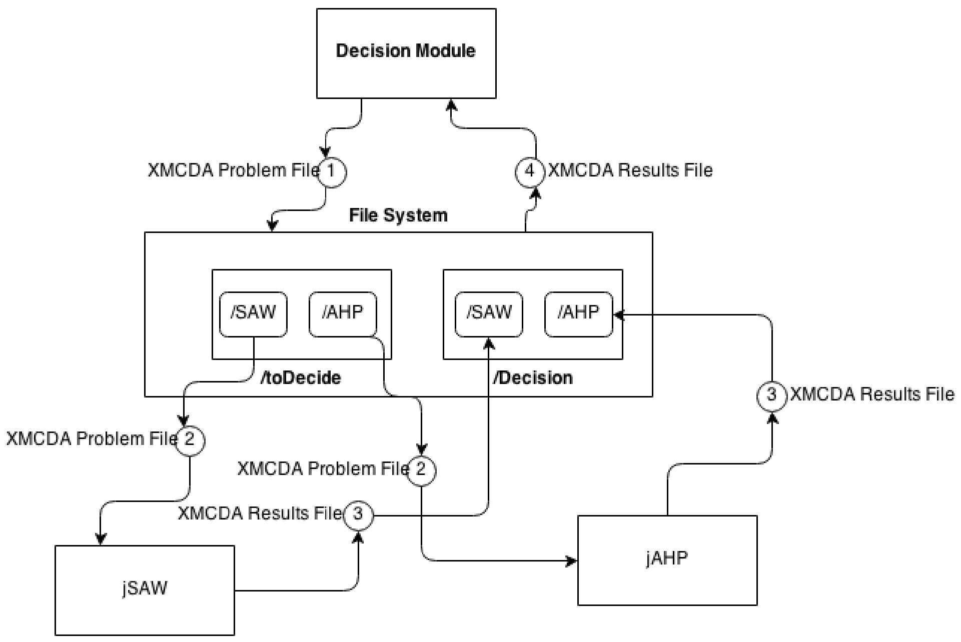


Fig. 17 - CloudAid1 - File Communication

The CloudAid1 prototype supports two different *Multi-Criteria Decision Methods*: Simple Additive Weight (SAW) and the Analytic Hierarchic Process (AHP).

4.4.3.1 Simple Additive Weight

SAW is a simple multi-criteria decision method that relies on criterion weighting to determine the ranking of the alternatives [24]. Criterion weighting consists in the definition of an importance degree for every criterion in the Service Template. These values are at the core of the method and therefore are mandatory.

4.4.3.2 Analytic Hierarchic Process

Contrary to the SAW method, the AHP method is based on comparisons between data in order to extract the importance weights and perform the necessary calculations to derive the desired ranked list [25]. When I refer to comparisons, I'm referring to something like "Alternative A_1 is better than Alternative A_2" or "Criterion C_1 is more important than Criterion C_2".

4.4 Aggregation Module

Once the decision module finishes ranking the alternatives of each *Service Template*, we move to the Aggregation Module. This is the final step performed by the CloudAid1 prototype; after this step, a ranked list of admissible aggregated solutions that match the Decision Maker's requirements will be passed onto the Controller.

The purpose of the aggregation module is to exploit the ranked list of alternatives of each *Service Template* to create possible aggregations of services. It's necessary to point out that each aggregated solution is composed by one alternative from each *Service Template*. Let's consider an example: assume we have a *CSA* composed by three *Service Template*'s A, B, C and their corresponding ranked lists of alternatives RL_A, RL_B, RL_C. If we denote A_i has an alternative from RL_A with rank *i*, a possible aggregated solution would be: $A_1B_1C_1$. Fig.18 presents a visual representation of this example.

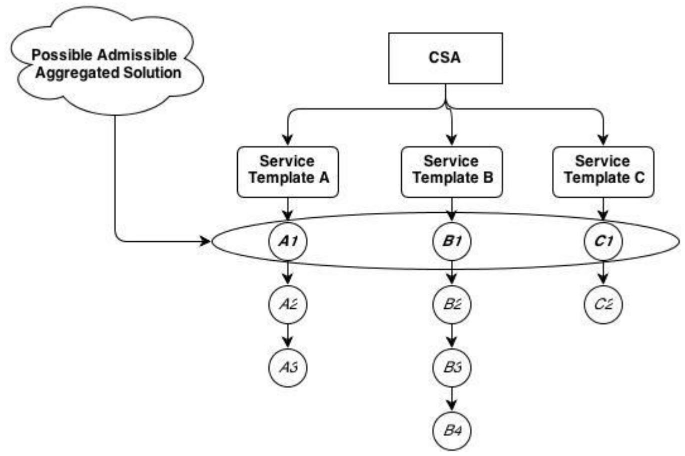


Fig. 18 - CloudAid1 - Aggregated Solution Example (Zi are alternatives with rank i on list Z = {A,B,C})

The Java class responsible for computing these aggregated solutions is the *Combinations* class. It implements two algorithms that compute all the possible Admissible Aggregated Solutions based on the ranked lists of each *Service Template*. One version of the algorithm is able to deal with incomparability between alternatives (on a very limited scale) while the other doesn't consider this particular yet important aspect of multi-criteria decision problems. I'll cover this topic with further detail on section 4.5.2. Admissible Aggregated Solutions are solutions that match the Decision Maker global constraints or, in other words, requirements that were defined at the *CSA* level. Section 4.5.1 will provide a better description on this subject.

4.5.1 Admissibility test

When considering every possible combination of alternatives from each *Service Template*, we must keep in mind that not every combination might be a valid one. For an aggregated solution to be considered valid, and therefore to be considered as possibility for the final solution list, needs to pass a series of tests first; these series of tests are performed by the Java class *AggChecker*.

These tests depend on the global constraints (requirements) set by the decision maker when creating his CSA. Consider an example where we have two alternatives for two different *Service Template*'s; a possible global requirement might be that every alternative needs to be compatible with Linux OS. If any alternative fails to support this type of operating system, the solutions fails the test and will be considered inadmissible. On CloudAid1, the only restriction supported is the global price; when the Decision Maker defines a global Price requirement he's telling us that every aggregated solution's cost needs to be inferior or equal to some value set by him.

4.5.2 Aggregated Solutions Algorithms

As mentioned earlier, an aggregated solution is composed of one alternative from each *Service Template* on the CSA. Following this line of thought, we can see that the number of possible aggregated solutions will grow exponentially depending on the number of *Service Templates* and corresponding alternatives.

To avoid this issue, the algorithms take advantage of the sorted ranked lists provided by the Decision Module. Let's assume we have three *Service Templates* A, B, C, with their corresponding sorted ranked lists of alternatives. Let's denote A_i as an alternative from *Service Template* A with rank *i*. With the knowledge that $A_i > A_k$, $i < k$ (alternative *a* is better than alternative *b* if *a* is ranked above *b*) we can say that any aggregated solution $A_iB_kC_l > A_oB_pC_q$ if $i,k,l < o,p,q$. In this case, $A_jB_kC_l$ dominates $A_oB_pC_q$ hence, there's no need test $A_oB_pC_q$ since its overall performance value will be lower.

Combining this knowledge with combinatory tree theory, the algorithms generate all possible combinations. They're based on the *Breadth-First Search algorithm* together with the notion of the *Branch and Bound* technique in order to transverse the tree of solutions and at the same minimize the computational weight of the problem. As mentioned earlier, there's two versions of the aggregation algorithm: *algorithm1* and *algorithm2*. In *algorithm1*, every time a new possible solution is discovered, it simply compare it with the already found admissible solutions to check for dominance. If there's dominance found between the node being tested and an already found solution, this node can be discarded along with its children. However, if no dominance is found, the node is added to the list of admissible solutions and its children discarded.

Algorithm2 is a bit different; by considering the possibility of incomparability between the alternatives (also known as a partial ordered list) it no longer assumes that when an admissible aggregated solution is found its children can be discarded. It checks for incomparability between the node and its children; if any children is incomparable with its father (which is a solution that passed the admissibility test), it'll be place into the *queue* for later testing.

Fig.19 shows the flow of execution of the Aggregation Module on an architectural level.

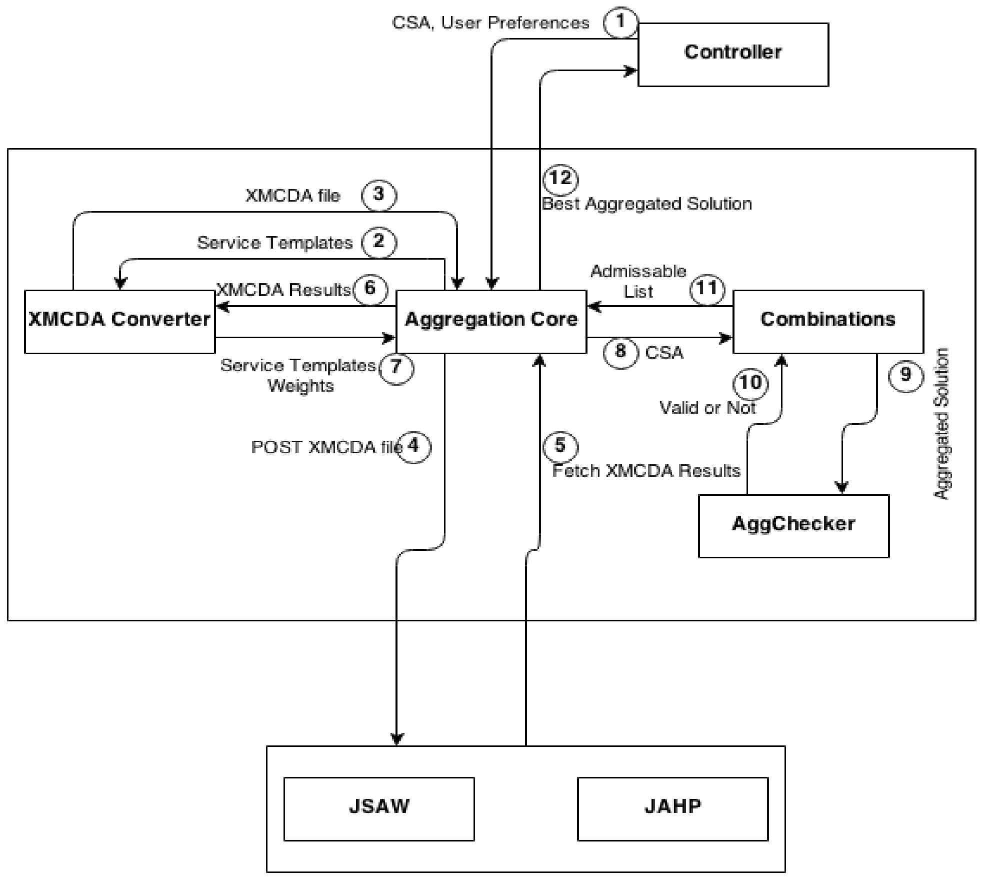


Fig. 19 - CloudAid1 - Aggregation Module Architectural Flow

In a summarized way, these are the main steps of the Aggregation Module:

1. **Get *Service Templates* weights** – After receiving the CSA from the *Controller*, it's necessary to retrieve the importance weights from the *Service Templates* in order to compare and rank the aggregated solutions. There are two ways of getting these weights: the Decision Maker explicitly defines them when creating his CSA or, use JAHP and the Decision Make's preferences in order to calculate the *Service Template* weights.
2. **Find admissible aggregated solutions** – This is the main objective of the module: compute the possible aggregated solutions. Every time a new solution is found, the admissibility test is applied to verify if the solution meets the requirements necessary to enter the list of admissible solutions.
3. **Find the best Aggregated Solution** – When every solution has been found it ranks the solutions using the importance weights and the SAW method.

5. CloudAid2

This chapter provides a detailed view of the new version of the prototype I implemented, the CloudAid2, and other two side projects (the Linked USDL Pricing API and the Service Gatherer) that were essential for its development.

I begin by introducing the Linked USDL Pricing API [53] and explaining the reasons behinds its creations (Section 5.1). Following, on section 5.2, I introduce the Service Gatherer, an application that is responsible for *feeding* the prototype's TripleStore[5]. On section 5.3, I describe in detail the modifications and additions to the CloudAid1 prototype and finally, on section 5.4 I present the CloudAid2 graphical user interface (GUI).

5.1 Linked USDL Pricing API

A major objective of this work was to introduce the Linked USDL pricing model developed on [9] by Eng. Jorge Araújo. While the model seemed to be a good approach to tackle the Cloud pricing scene, there wasn't anything alongside it to help others describe services, using the Linked USDL Pricing and Linked USDL core models, in a simple way. Besides this lack of support, the model itself lacked some form of validation; I had to make sure that this model was flexible enough to model different service offerings from different service providers. To solve these issues, two solutions were proposed:

- Develop a friendly Java API to help developers describe services using the Linked USDL Core [17] and Linked USDL Pricing model [9] (Section 5.1.1).
- Take advantage of the taxonomy developed on [47] and of the Linked USDL Pricing API to perform a conceptual validation the Linked USDL pricing model (Section 5.1.2) and, at the same time, validate the Linked USDL Pricing API's functionalities.

On the next sections, I'll describe the Linked USDL pricing API, how it works and the technologies supporting it.

5.1.1. Overview

The development of the pricing API was firstly suggested by Dr. Jorge Cardoso has a follow up of an idea that had emerged throughout the development of the CloudAid1. The idea was to work with Eng. Jorge Araújo in order to develop a friendly Java API that would help developers design applications/systems able to structure services information using the Linked USDL core and Linked USDL pricing models in a simple way. This was a good opportunity to work alongside a professional and a good approach to solve the lack of support and usability of the Linked USDL pricing model. This way, we'd be able to reduce the gap between developers and the Linked USDL pricing model, providing a mean to programmatically interact with it.

After some consideration about the amount of time and effort it would be required to develop the API and, comparing its advantages with its disadvantages, I decided to move forward with the idea. Thus, I and Eng. Jorge Araújo worked together throughout the next month in order to develop the first version of the Linked USDL Pricing API, which can be followed on [50].

[5] http://www.w3.org/2001/sw/wiki/Category:Triple_Store

To use the Linked USDL Pricing API, the developer just needs to follow the following steps (a detailed explanation of the model is available on [6,7]):

1. Import the Linked USDL Pricing API into his Java project.
2. Import, instantiate and populate the Java Models with the corresponding data.
3. After populating the models, call the *writeToModel()* function to create the semantic description of the service.

On the following sections, I'll explain in detail how the pricing API and its functionalities work. Section 5.1.2 starts by explaining the Java models and their role on the API, followed by Section 5.1.3 where I'll explain the mapping between the Java models and their Triple representation. On section 5.1.4 I'll explain the reasoning behind the dynamic price calculation and how it can be achieved using the API. Finally, on Section 5.1.5, I'll describe the conceptual validation of the Linked USDL pricing model and how it was executed with the Linked USDL pricing API.

5.1.2. Linked USDL pricing API Java Models

The Linked USDL Pricing API is composed of eleven main Java models. These eleven models are at its core and are the containers for the services information. Looking at Fig.20, we can see how these models relate to each other; this diagram is but a simpler version of the class diagram presented on **Appendix C**.

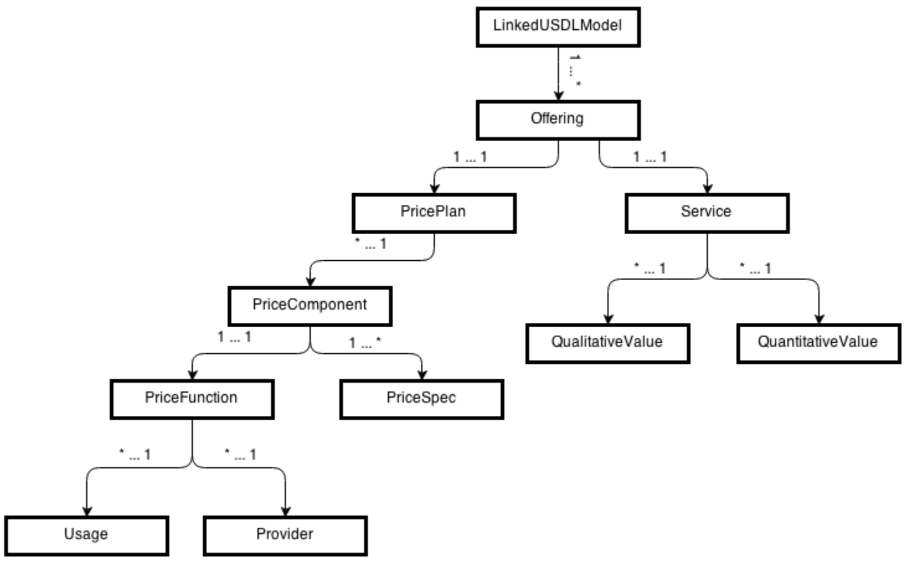


Fig. 20 - Pricing API Java models

Looking at Fig.20, we can see that most of these objects represents an object of either the Linked USDL core model [17] or Linked USDL pricing model [6,7] (except the Linked USDLModel class). Everything related, and including, to the *PricePlan* class are objects of the Linked USDL pricing model. The *Offering* and *Service* class are objects from the Linked USDL core model while the *PriceSpec*, *QualitativeValue* and *QuantitativeValue* classes are objects from the GoodRelations ontology[6].

[6] http://semanticweb.org/wiki/GoodRelations

Following is a brief explanation of the roles of each data model:

- **Linked USDLModel** – Container of the *Offering* class and consequently, every other model. It's also capable of performing reading and writing (from Java to triples, and vice-versa) operations regarding the offerings objects it's currently holding.

- **Offering** – Java class that represents an instance of the *ServiceOffering* object from the Linked USDL core [17]. Regarding the Linked USDL pricing model, this Java class (or from the Triple-oriented point of view, this Triple object) is the linking point between the Linked USDL pricing model and the Linked USDL core model.

- **Service** – Class that represents an instance of the *Service* object from the Linked USDL Core model. It's the container for the *QualitativeValue* and *QuantitativeValue* classes.

- **QuantitativeValue** – Class that is used to model quantitative features of the service. Quantitative features are those whose value can be quantified like MemorySize (RAM), CPU Speed, Disk size, etc.

- **QualitativeValue** – This class, contrary to the *QuantitativeValue* class, is used to model the qualitative features of the Service. Qualitative features are those whose value is a concept with some meaning attached to it, like the brand of a CPU (CPUType) or the name of the operating system.

- **PricePlan** – Represents an instance of the *PricePlan* object from the Linked USDL pricing model. It holds every piece of information necessary to calculate the cost of the Offering's Service. Each *PricePlan* can be composed of one or more *PriceComponents*.

- **PriceComponent** – Represents an instance of the *PriceComponent* object from the Linked USDL pricing model. This object/class represents the billable (or deductions) attributes of the Service. For example, the cost per hour of a VM from Amazon EC2 could be modeled using a single *PriceComponent*; in case there are other billable attributes, like an extra fee for running a different operating system, this could and should be modeled using a different *PriceComponent*. The final cost of the Offering will be the sum of its billable attributes and subtraction of its deductions. A relevant feature of this class is that is capable of modeling both static and dynamic prices. This class is the container for both *PriceSpec* and *PriceFunction* class and, as their name implies, *PriceSpec* is the class responsible for modeling static prices while the *PriceFunction* class is responsible for modeling mathematical expressions that calculate the cost of the service based on some values provided by the user. I'll provide a more detailed explanation on the calculation of dynamic prices, and how they can be obtained using the API, on Section 5.1.4.

- **PriceSpec** – Represents an instance of the *PriceSpec* object from the GoodRelations ontology. This class/triple object represents a static pricing value.

- **PriceFunction** – Represents an instance of the *PriceFunction* object from the Linked USDL pricing model. Using this class, developers can define mathematical expressions to calculate a personalized price based on values provided by the user/customer (*Usage* variables) and some pre-defined values (*Provider* variables). Currently, this class supports two types of expressions:

    - **Simple Expressions** – It's a simple mathematical expression to calculate the cost of the service/service feature, e.g.: NumberOfMonths (usage variable) * VMCostPerHour (provider variable)
    - **Composed Expressions** – Mathematical expressions whose behavior is controlled by IF-ELSEIF-ELSE conditions. The behavior is a simple mathematical expression. E.g.: IF(COND1) ; BEHAVIOR1 ~ ELSEIF (COND2) ; BEHAV2 ~ ELSE BEHAV3. On section 5.1.3.1 I'll explain how I'm able to transform this type of expression into its corresponding SPARQL Query.

- **Usage Variable** – Represents an instance of the *Usage* object from the Linked USDL pricing model. This class represents the variables whose value is only known by the customer/user; we can look at it as an empty slot that needs to be filled with a value provided by the user. A good example of this type of variable is the amount of time that the user is going to be using the provider's service.

- **Provider Variable** – Represents an instance of the *Provider* object from the Linked USDL price model. It represents a function variable whose value is known à priori. It's usually a fee for a certain attribute of the service, like the cost per hour of the virtual machine or the price of a different operating system.

To use these Java models, the developer just needs to import them into his/her project and populate them with either Strings, Doubles or one of the above Java models. While these models are the core of the Linked USDL Pricing API, they need many secondary classes to achieve their semantic representation.

5.1.3. Java to Triples and Vice-Versa

On this section I'll explain how the API maps its Java models into a Triple representation and, the reverse process. Using the service's Java description, it can transform the data contained within the models into their corresponding semantic description.

Section 5.1.3.1 describes how the API maps the Java models information into its Triple representation while section 5.1.3.2 will cover the reversing process.

5.1.3.1. From Java to Triples

Once populated, the developer can transform his populated *Offering* objects into a Triple[7] representation following one of two possible ways:

- Call the *writeToModel()* function from the *LinkedUSDLModel* class. This will transform every *Offering* object on the *LinkedUSDLModel* instance into its Triple representation.
- Call the *writeToModel()* function from the *Offering* class. This way, we provide to the developer a means to transform only one object of the *Offering* class into a triple representation.

To achieve the desired Triple representation, besides the information on the Java models, the API relies on three third party libraries and some secondary classes. Their purpose is to assist in the transformation of the Java objects into their Triple representation.

These three libraries are:

- **Jena API** [8]– A TripleStore engine.
- **TopBraid SPIN API**[9] – Library developed on top of the Jena API that enables the use of the SPIN [11, 12 and 13] model programmatically.
- **Symja**[10] – A mathematical expression parser.

Calling the *writeToModel()* function initiates a chain reaction throughout the Java models, calling the *writeToModel()* function from each Java model introduced on the previous section. Instead of developing a centralized system that would transform every Java object into a Triple representation, we decided to scatter this task among the corresponding Java models. As such, each of the models is in charge of converting itself from its Java representation into its Triple representation. For example: calling the *writeToModel()* function from the *Offering* class will create a Resource object from the Jena API. Following, the method will populate with the *Resource* with the standard attributes of the *Offering* instance, Strings and/or Integers, and will call the *writeToModel()* function of its Java model attributes. In the case of the Offering class, it has a *PricePlan* instance and a *Service* instance. To get their Triple representation, it needs to call their *writeToModel()* function: *Service.writeToModel()*. This concept is the same for every Java model of the pricing API.

Every *writeToModel()* function uses both Jena API library and the secondary classes mentioned at the beginning of the section. These secondary classes are Java Enums that work as an interface between the ontologies and the Jena API. For example, to use the

[7] http://www.w3.org/RDF/
[8] https://jena.apache.org/
[9] http://topbraid.org/spin/api/1.2.0/
[10] https://code.google.com/p/symja/wiki/MathExpressionParser

property *type* of the *rdf* ontology, the developer needs to import and use the *RDFEnum* Java *enum*. Calling the *RDFEnum.RDF_TYPE.getProperty()* returns a Property[11] resource from the Jena API representing the *rdf type* property.

Currently, the API has interfaces for the following ontologies:

- *Cloud Taxonomy*
- *FOAF*
- *GoodRelations*
- *RDF*
- *RDFS*
- *Linked USDL Core*
- *Linked USDL Price*

Each of these Enums has the most commonly used properties of the ontologies as well as the ones necessary for the development of the CloudAid2 project.

In the case of the *Cloud Taxonomy*, there isn't a *getProperty()* function but a *getConceptResource()* function. Since the *Cloud Taxonomy* is but a collection of Cloud concepts, this function returns a JENA Resource object that represents the corresponding Cloud concept. For example, calling the *CLOUDEnum.LOCATION.getConceptResource()* returns a Resource object that should be linked to a *rdf type* property (this is mainly used in the QualitativeValue and QuantitativeValue classes to state which is the type of service feature that is being modeled).

While every *writeToModel()* function uses the Jena API extensively, only the *PriceFunction*'s *writeToModel()* function uses the TopBraid SPIN and Symja libraries. These libraries are extremely important for the Linked USDL Pricing API and are used in the following order:

- **Symjya** – Used by the *MathExp2SPARQL* class. This class is responsible for parsing the mathematical expression inserted by the user and returning its corresponding SPARQL representation.

- **TopBraid SPIN** – Used by the *PriceFunction's writeToModel()* and *readFromModel()* functions. Responsible for transforming the SPARQL query received from the MathExp2SPARQL class to its SPIN [11, 12 and 13] representation and vice-versa.

The *MathExp2SPARQL* class receives a String with the mathematical expression inserted by the user and creates its AST (Abstract Syntax Tree)[12] description. Traversing through the AST, it's able to identify each member of the expression and create the corresponding SPARQL Query.

Having the SPARQL Query of the mathematical expression is just one of the two steps necessary to successfully model the *PriceFunction* instance into its Triple representation. Step two consists in the transformation of the SPARQL Query returned by the *MathExp2SPARQL* into its SPIN objects representation. This is where the TopBraid

[11] https://jena.apache.org/documentation/javadoc/jena/com/hp/hpl/jena/rdf/model/Property.html
[12] http://en.wikipedia.org/wiki/Abstract_syntax_tree

SPIN library comes in. TopQuandrant[13] provides a set of examples alongside their TopBraid SPIN API where they demonstrate its usage and its transformation process. By studying these examples, I was able to reproduce the conversion from SPARQL into its SPIN object representation which, is later connected to the *PriceFunction*'s Jena Resource.

Once every Java model has been transformed into its Triple representation, the semantic description of the service is done.

5.1.3.2. From Triples to Java

The Linked USDL Pricing API is also capable of reading and loading the information of a semantic model into its corresponding Java models.

Like in the writing scenario, there's also two possible ways of reading the data contained in the model:

- Read every *Offering* from the Semantic model using the *createFromModel()* function of the *LinkedUSDLFactory* class.
- Read one *Offering* object from the semantic model using the *Offering.readFromModel()* function.

The reading process is exactly the same for both cases but, while the *readFromModel()* function receives a Jena *Resource* of the type *Offering* (that is, it receives a Jena *Resource* object from where it's supposed to fetch the *Offering's* information), the *createFromModel()* function receives the path of the directory where the semantic model is located and loads every *Offering* on that model automatically.

This *reading* process follows the same methodology adopted on the *writing* process: each of the Java models is responsible for extracting its own information from the model. It takes advantage of the same libraries and classes except, now it executes the process in the opposite direction. For example, to extract the information from a *PriceComponent* Jena Resource, the *PriceComponent's readFromModel()* function uses the same *Enums* described on the previous section alongside the *hasProperty()* and *getProperty()* methods from the Jena's *Resource* object. This way, it's possible to extract the necessary information to populate its Java attributes, whether they're Strings, Integers, Doubles or Java classes (if the instance needs to read another Java model from the semantic model to populate one of its attributes, it just has to call its *readFromModel()* function).

Regarding the *PriceFunction* object, only the TopBraid API is needed since what the developer usually needs is the SPARQL Query (or function, as one wishes to call it) and not its SPIN representation, which, due to its complex description, yields little value.

5.1.4. Dynamic Pricing

Using the Linked USDL pricing API, it's possible to model services whose price is dynamically calculated using mathematical expressions. These mathematical expressions use the values from its *Provider* and *Usage* variables to calculate the cost of the service.

As previously mentioned, *Provider* variables are variables whose value is known *à priori* while, *Usage* variables only have their value known during execution. This is because they're variables whose value needs to be supplied by the person/entity that is

[13] http://www.topquadrant.com/

going to use the service. For example, a very common *Usage* variable used nowadays on the Cloud market is the number of hours/days/months the client is going to be using/renting the service. Depending on its value, the cost for the same service can be very different between different consumers.

The development of a semantic model able to support dynamic price calculation and the usage of said model are two different things. As long as the *PriceComponents* from the *Offering's PricePlan* use a mathematical expression to calculate the cost of the service, we're dealing with a dynamic semantic model.

The main objective of the Linked USDL Pricing API is to provide a simple way to create and use dynamic semantic models (namely the Linked USDL pricing model). To achieve this goal, there're two steps that need to be performed:

- Create a generic semantic model using the Linked USDL API. When I refer to generic semantic models, I'm referring to the same concept used on Object Oriented Programming where developers create Classes (templates) which are later instantiated and populated with data. Regarding the dynamic price calculation, it's exactly the same.

- Import the generic model created and populate its *Usage* variables. After providing values for the model's *Usage* variables, the developer ends up with an instance of the generic model, but with some information about its *Usage* variables.

Once the values have been set for each of the *Offering's PriceFunction's Usage* variables, all that is left to do is call the *writeToModel()* function once again to write our instance of the generic model into a new Triple representation.

After that, to obtain the final cost for the *Offerings* of the model, the developer just needs to load the instance of the model and call the *calculatePrice()* function from each of its *Offering*'s. This function iterates over the *PriceComponents* of the *Offering's PricePlan* and:

- Checks if it's a Deduction or a regular PriceComponent.
- Regardless of the type, check if we're dealing with a dynamic or static *PriceComponent/Deduction*.
- In case we're dealing with a static *PriceComponent*, simply add/subtract its value to the *PricePlan's* final value; else, execute the SPARQL function and only then add/subtract its result to the *PricePlan's* final value.

5.1.5. Linked USDL Pricing model validation

To assess the flexibility of the model developed by Eng. Jorge Araújo and, at the same time, test the functionalities, possibilities and limitations of the new API, I proposed a challenge that takes into consideration the survey performed by *451Research* [48]. While their study is mainly focused on *IaaS* services, it's a good and solid starting point for the purpose at hands: verify if both the Linked USDL Pricing API and the pricing model can tackle today's cloud pricing scene (or at least, a portion of it).

Aside the pricing *taxonomy*, they also present the list of 52 Cloud providers that were studied, and afterwards classified, in the survey. As each one of these use cases was classified into one of the 8 possible categories, I modeled (using our API) at least one service offering from the most popular provider of each category.

To choose the best (most popular) provider from each category, I developed a *script* in python that extracts the number of page hits, returned by Google's search engine, for a specific search about the cloud services of the selected provider (the full results can be seen on **Appendix D**). After a brief analysis of the results, I selected the "winner" for each of the eight different pricing methods. Table.1 presents the chosen use cases.

Pricing method	Bundling method	Provider
Recurring Resource Pooling	PB	VMWare vCloud Hybrid[14]
PrePaid Credit	VB	Micros. Azure Virtual Machines[15]
PrePaid Subscription Credit	UB	CloudSigma Cloud[16]
PrePaid VM	PB	IDC Frontier Cloud[17]
On Demand	VB	Amazon EC2[18]
Spot Pricing	VB	Amazon EC2[10]
Reserved Instance	VB	Amazon EC2[10]
Recurring PrePaid VM	FB	Arsys Dedicated Servers[19]

Table 1 - Modeling use cases

These use cases cover every aspect of the CODEX pricing *taxonomy*. It considers the eight different types of pricing methods and the four different types of bundling identified on the survey (Fully Bundled, Processor Bundled, VM Bundled, and Unbundled).

Once selected, I performed a detailed study of their services and how their pricing was calculated. Identifying the service features, studying their pricing methods and identifying their billable attributes was the first step of the challenge. Afterwards, it was necessary to model each of the selected service offerings using the Linked USDL Pricing API and consequently, the Linked USDL Pricing model.

Every service offering was successfully modeled and tested using the API, its results being validated either by using tools provided by the Cloud providers themselves (e.g. Amazon EC2 calculator[20]) or by manually performing the necessary mathematical operations to calculate the expected cost of the service.

These service modeling examples are included in the API's *tar ball* which, can be downloaded from [50, 51]. They're good examples of the API's functionalities/capabilities and a good way of introducing the Linked USDL Pricing API to other developers that might be interested in it. Alongside the API and its modeling examples, [50] has a *wiki* section where interested parties can find and read further documentation about the API, how to use it and future plans.

[14]http://www.vmware.com/files/pdf/vchs/VMware_vCloud_Hybrid_Service_Purchasing_and_Subscription.pdf
[15] http://azure.microsoft.com/en-us/offers/commitment-plans/
[16] http://www.cloudsigma.com/
[17] http://en.www.idcf.jp/cloud/
[18] https://aws.amazon.com/ec2/
[19] http://www.arsys.net/
[20] http://calculator.s3.amazonaws.com/index.html

5.2 CloudAid2 – Service Gatherer

One of the proposed objectives for this thesis was to extract and model data from real cloud services. Thus, as soon as the API was capable of modeling cloud services information using the Linked USDL core and Linked USDL pricing models, I moved forward to the creation of the CloudAid2's service repository. I designed and implemented the Service Gatherer [53], which is an application that combines *scrapping/parsing* techniques to extract data from the provider's websites, and structures it using the Linked USDL Pricing API.

Taking advantage of the service modeling described on section 5.1.5, I developed a few *scrappers/parsers* for the following use cases:

- Arsys dedicated servers[21]
- Amazon:
    - Elastic Compute Cloud (EC2)[22]
    - Relational Database Service (RDS)[23]
    - Glacier[24]
    - Elastic Load Balancing[25]

With the Arsys dedicated servers situation, its information is extracted by a Java class called *Arsys*. Using jsoup[26] to parse the provider's webpage, extracting the *offerings* information becomes simple:

1. I used *Google Chrome's inspector* to get the *CSS Path* of the *html* element that is containing the desired data.
2. Having its *CSS* Path, I can fetch the *Element*[27] from the *html* source using the *select()* function from the json-simple library. Afterwards, iterate over its contents and extract the data.

For example, it's relatively common to use *html table*'s[28] to structure and present service's information to the user in a *webpage*; in this situation, we just need to fetch the table, iterate over its rows (*Elements*) and process the data.

Of course, each *scrapper/parser* requires a preliminary study of the *html webpage*, the information contained within it and the service's pricing method to create a proper semantic representation. While it's not a particularly complex task, it can be very time consuming.

Table.2 presents the service's features modeled by the *Arsys* class using the Linked USDL Pricing API.

[21] http://www.arsys.net/
[22] http://aws.amazon.com/ec2/
[23] http://aws.amazon.com/rds/
[24] http://aws.amazon.com/glacier/
[25] http://aws.amazon.com/elasticloadbalancing/
[26] http://jsoup.org/
[27] http://jsoup.org/apidocs/org/jsoup/nodes/Element.html
[28] http://www.w3schools.com/html/html_tables.asp

Arsys Dedicated Servers	
***CloudTaxonomy* feature**	**Type**
Data IN External	Quantitative
Data IN Internal	Quantitative
Data OUT External	Quantitative
Data OUT Internal	Quantitative
Transferrate	Quantitative
CPU Cores	Quantitative
CPU Speed	Quantitative
MemorySize	Quantitative
DiskSize	Quantitative
StorageType	Qualitative
Feature	Qualitative
Monitoring	Qualitative
Language	Qualitative
Platform	Qualitative
Security	Qualitative
Web	Qualitative
Console	Qualitative
GUI	Qualitative
Unix	Qualitative
Windows	Qualitative
Price	Quantitative

Table 2 - Arsys Dedicated Servers features

Amazon's situation is a little bit more complex. Their services descriptions can also be found in *HTML webpages* however, their pricing information is kept separately and, it's expressed in JSON [26, 27] format, not *HTML*. As such, the modeling of their service offerings is composed of two steps:

1. **Extraction and creation of the service's features from the *HTML webpage* using Jsoup and the Linked USDL Pricing API** – *scrapping* performed by a function or class specifically created for the purpose of generating the *Service* instances without their pricing components. E.g.: *AmazonRDSBaseServices* Java class.
2. **Extraction of the pricing information by parsing the JSON files with the help of the *json-simple* [29] Java library** – *Parsing*, just like the *scrapping*, is performed by a function or a class specifically created for the job. It creates the *Offering* instance, linking together the *Service* object created on *step 1* and a pricing description. E.g.:*AmazonRDSMySQL* Java class.

These two steps apply for every Amazon use case. First, I create a description of the service without its pricing components; afterwards, I *parse* their JSON[30] files, extracting the pricing information and adding it to an *Offering* instance that contains a *Service* instance. For example, the method *baseServices()* from the *AmazonRDSBaseServices* class returns an *ArrayList<Service>*. Each element in the *ArrayList* contains the description of a

[29] https://code.google.com/p/json-simple/
[30] http://aws-assets-pricing-prod.s3.amazonaws.com/pricing/ec2/mswinSQLWeb-ri-heavy.js

service without a pricing component (e.g.: Amazon's *db.m3.medium*[31] instance). These *Service* instances are used by the *parsing* classes (e.g.: *AmazonRDSMySQL* class) which extracts and models the pricing information from the JSON files, creates the *Offering* instance and finally links together a *Service* and a pricing description (a *Price Plan*).

This approach simplifies the modeling process since a single *Offering* may have multiple prices depending on its features (e.g.: Amazon's *db.m3.medium* has different *fees* depending on its location).

Table.3 presents the service's features modeled by the Amazon's *scrappers/parsers*.

EC2		**RDS**		**Glacier**		**Load Balancing**	
CloudTaxnomy	Type	*CloudTaxnomy*	Type	*CloudTaxnomy*	Type	*CloudTaxnomy*	Type
Location	Qual.	*I/O Ops.*	Quant.	*Location*	Qual.	*Location*	Qual.
Unix	Qual.	*Storage Cap.*	Quant.	*Feature*	Qual.	*Feature*	Qual.
Bit64	Qual.	*Back-Up Rec.*	Qual.	*SSL*	Qual.	*Protocol*	Qual.
Bit32	Qual.	*Monitoring*	Qual.	*Encryption*	Qual.	*SSL*	Qual.
CPUCores	Quant.	*CPUCores*	Quant.	*Durability*	Quant.	*Load Balancing*	Qual.
CPUSpeed	Quant.	*CPUSpeed*	Quant.	*Storage Cap.*	Quant.	*Reliability*	Qual.
MemorySize	Quant.	*MemorySize*	Quant.	*API*	Qual.	*Scalability*	Qual.
DiskSize	Quant.	*Feature*	Qual.	*PUT Req.*	Quant.	*Data Processed*	Quant.
StorageType	Qual.	*Performance*	Quant.	*POST Req.*	Quant.	*Console*	Qual.
Feature	Qual.	*Location*	Qual.	*DELETE Req.*	Quant.	*Unix*	Qual.
Performance	Qual.	*Platform*	Qual.	*LIST Req.*	Quant.	*Windows*	Qual.
CPUType	Qual.	*Price*		*GET Req.*	Quant.	*Price*	Quant.
Platform	Qual.			*Price*	Quant.		
Data IN Ext.	Quant.						
Data IN Int.	Quant.						
Data OUT Ext.	Quant.						
Data OUT Int.	Quant.						
Price	Quant.						

Table 3 - Amazon's service offerings features

It's important to note that every *parser/scrapper* class uses a local copy of the sources of information (JSON files and *HTML webpages*) instead of issuing a request through the *web* for a "fresh" copy every time we wish to generate new service descriptions. Choosing this approach instead of the latter seemed best for two reasons:

1. Ensure that every *scrapper/parser* works as intended – *HTML webpages*/JSON files are constantly being updated. Most of the changes are usually related to the content of the document rather than its structure however, structural changes may occur. If they do, they'd probably turn the *scrapper* obsolete, forcing an update to cope with these changes or even the creation of a new one.

2. Enable the possibility of generating new Linked USDL descriptions based on real service descriptions. Keeping a local copy ensures that we can re-create

[31] http://aws.amazon.com/rds/details/

the services descriptions with the same, or new, information any time we want.

Every *scrapper/parser* supports content modifications that is, as long as the structure of the *HTML/*JSON files remains intact or there aren't any errors (e.g.: the *scrapper/parser* is expecting an *Integer* and the new inserted value is a *String*), the *scrappers/parsers* will be able to generate the new Linked USDL descriptions.

The current Service Gatherer generates approximately 8500 service descriptions (~450Megabytes) from real cloud services information, including a dynamic modeling of their pricing description, with the help of the Linked USDL Pricing API. Section 5.3.2 will address the dynamic aspect of the Linked USDL service descriptions with further detail.

The Service Gatherer application can be followed on [53] and the full dataset can be found on [52].

5.3 CloudAid2 – Prototype

This section describes the CloudAid2 prototype in detail. I'll be focusing on the additions/changes made to the prototype, describing its new functionalities and what had to be modified to successfully achieve the proposed requirements/objectives. While section 4 provides a brief overview of the previous version of the prototype, accompanying the reading of this section with Appendix B is advised for it would ease and enhance its understanding.

On section 5.3.1, I start by introducing the new *Controller*, explaining its new functionalities and the architectural changes to cope with a Client-Server architecture. From section 5.3.2 to 5.3.4, I'll introduce the changes/additions made to the Search, Decision and Aggregation modules respectively and on Section 5.3.5, I'll introduce the Graphic User Interface (GUI[32]) of the prototype.

5.3.1. Controller

Like in the previous prototype, the *Controller* class is still in charge of monitoring and controlling the application's flow of execution however, its communication link with the Graphic User Interface (GUI) no longer exists thus, simplifying its execution. The new Controller is in charge of the following tasks:

1. **Initialize each of its modules** – When instantiated, the controller is responsible for initializing the Search Module, the Decision Module and the Aggregation Module.
2. **Wait for GUI requests/responses** – Once every module has been initialized, it waits for the GUI to write the JSON Request on a pre-defined directory (Section 5.3.1.1).
3. **Control the application's execution flow** – When a request is received, the controller is in charge of reading the request and initiating the process to find the desired aggregated solutions (Section 5.3.1.2).

[32] http://www.britannica.com/EBchecked/topic/242033/graphical-user-interface-GUI

4. **Send data request to the GUI** – Every time a module needs extra information to continue the process, it asks the Controller to create and send a request to the GUI (Section 5.3.1.1).

Task 1 is executed only once while task 2, 3 and 4 are executed every time it's necessary.

5.3.1.1. Client-Server Communication

This section focuses on the architectural changes that had to be made to ensure that both the *Controller* and the *GUI* are independent of each other. As such, anything related to the old *UI* module was removed and their communication is now established via JSON[33] files written on specific directories.

Due to time constrains, I was unable to implement a real *web* communication between the GUI and the Server. While it was a personal objective, it wasn't part of the initial list of objectives and it would be interesting from a usability perspective, with the inclusion of the Linked USDL Pricing API, it had to be left aside and considered for future work. Despite its inexistence, their communication and exchange of information is established following a traditional *web* scenario through *requests-responses*[34].

Choosing JSON as a data-interchange format enables the communication with any type of GUI no matter the technology it was developed on.

To create a JSON description of the Java data models, I followed a simple approach that allowed me to focus on other relevant issues of the prototype: Gson[35], a Java library that can convert Java Objects into their JSON representation, and from their JSON representation back to Java Objects.

Figure.21 is a simple sequence diagram created to highlight the 3 possible points of interaction with the GUI and therefore, the decision maker. Each of these steps encompasses a *writing* operation to send the JSON data and a *reading* operation to receive back the response from GUI:

1. When the Search Module is requesting values for the pricing variables of the found alternatives (Section 5.3.2)
2. When there's qualitative criteria defined on the *Service Template* (Section 5.3.3)
3. When the Aggregation Module finishes computing the admissible solutions and it's time to send the results to the GUI.

In both scenarios, *reading* and *writing*, each of the modules recurs to secondary Java models, created specifically for external communication, to contain the information they wants to share. These models are in turn passed onto the Controller (through a *static* method) which uses the *gson.toJSON()*[36] function to transform these Java models into their

[33] http://json.org/
[34] http://en.wikipedia.org/wiki/Request-response
[35] https://code.google.com/p/google-gson/
[36] https://google-gson.googlecode.com/svn/trunk/gson/docs/javadocs/com/google/gson/Gson.html

JSON representation. Finally, it writes the JSON representation of the model into file on a pre-defined directory.

Table.4 presents a list of the Java models and methods that each model uses for external communication.

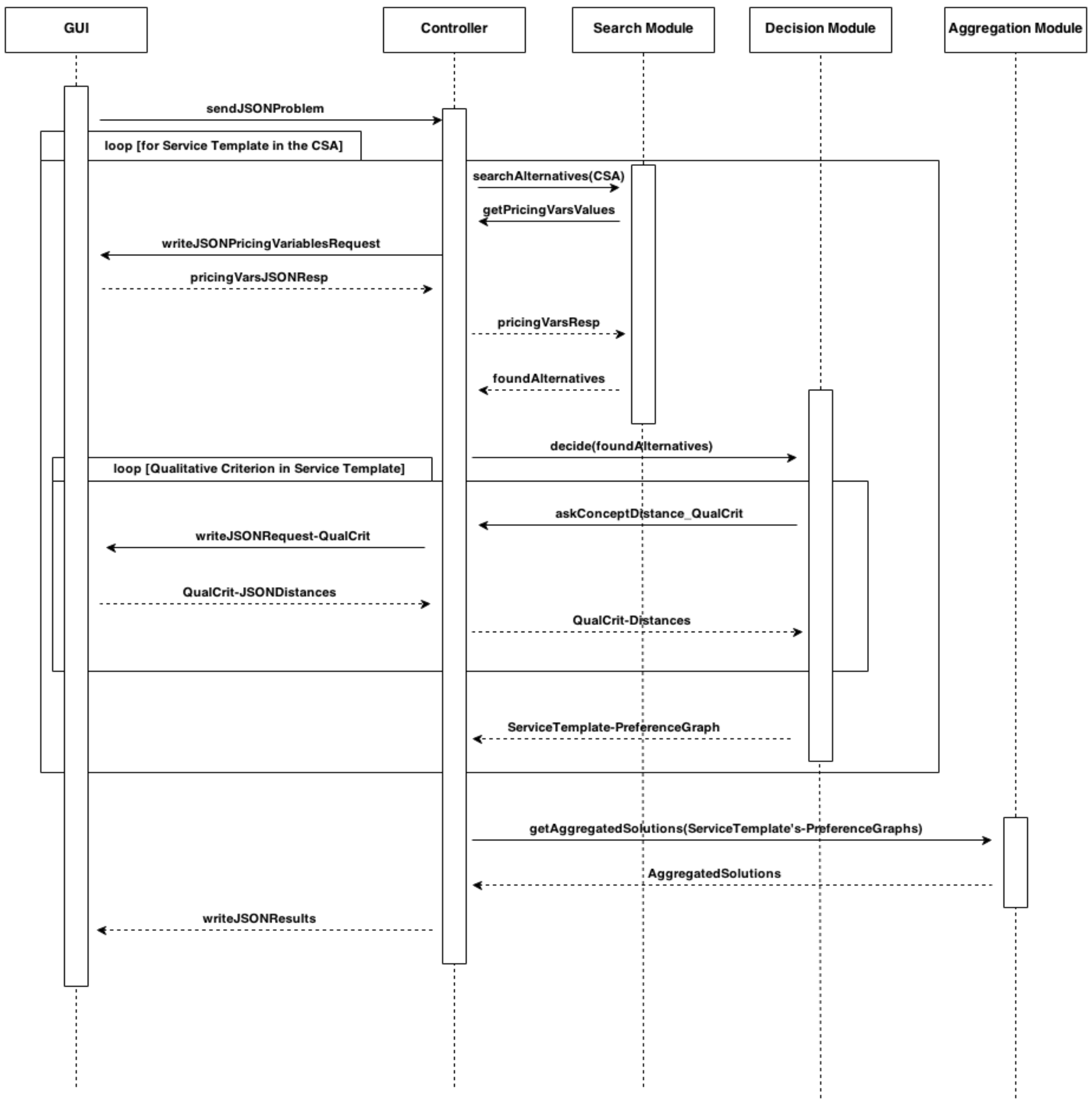

GUI
Controller
Search Module
Decision Module
Aggregation Module
sendJSONProblem
loop [for Service Template in the CSA]
searchAlternatives(CSA)
getPricingVarsValues
writeJSONPricingVariablesRequest
pricingVarsJSONResp
pricingVarsResp
foundAlternatives
decide(foundAlternatives)
loop [Qualitative Criterion in Service Template]
askConceptDistance_QualCrit
writeJSONRequest-QualCrit
QualCrit-JSONDistances
QualCrit-Distances
ServiceTemplate-PreferenceGraph
getAggregatedSolutions(ServiceTemplate's-PreferenceGraphs)
AggregatedSolutions
writeJSONResults

Fig. 21 - Simple external communication sequence diagram

Search Module – *step 1*		**Decision Module – *step 2***		**Aggregation Module – *step 3***	
Java Model	***Method***	**Java Model**	***Method***	**Java Model**	***Method***
PriceVariable	*requestVariablesInfo()*	*ConceptDistance*	*requestDistancesInfo()*	*AggregationComponent*	*sendResults()*
PricingVariables		*DistancesContainer*		*AggregatedSolution*	
				AggregationSolutions	

Table 4 - Controller's methods and Java models for external communication

I won't go into much detail about the models or what they're need for, I leave that for the following sections. What's important to note is that each model is composed only of simple Java attributes, like String, Integers or Lists to ease the JSON conversion. Once one of the modules calls its method, these models will be converted into their JSON description and written into a specific directory of the GUI.

Once the file is written into the GUI's directory, the Controller launches a *WatchService*[37] that is in charge of monitoring a pre-defined directory of the Server, waiting for the *response*. When a new JSON file is written on the directory, the *WatchService* loads the content of the file back into its Java containers using the *gson.fromJSON()*[38] function and, returns them to the corresponding module in order to resume the process.

5.3.1.2. Execution flow

Figure.21 is a simple sequence diagram that was created to highlight the points where external communication is expected, it doesn't show the internal execution of the *Controller*. *Algorithm 1* shows the entire process executed from the moment that the request is received until the aggregated solutions are found and sent to the user.

```
CSA ← readJSONRequests()
ok ← EvaluateCSA(CSA)
if ok then
     for ServiceTemplate ∈ CSA
           foundAlternatives ← Search(ServiceTemplate)
           if foundAlternatives not empty then
                decisionResults ← Decide(foundAlternatives)
                add decisionResults to ServiceTemplatesPreferenceGraphs
           else
                exit(1)
           end if
      end for
      aggregatedSolutions                                      ←
FindAggregations(ServiceTemplatesPreferenceGraphs)
     sendResults(aggregatedSolutions)
else
    exit(2)
end if
```

Algorithm 1 – *Controller's execution flow*

[37] http://docs.oracle.com/javase/tutorial/essential/io/notification.html
[38] https://google-gson.googlecode.com/svn/trunk/gson/docs/javadocs/com/google/gson/Gson.html#fromJson(com.google.gson.JsonElement, java.lang.Class)

The algorithm 1 starts by receiving a JSON description of the *CSA* (which contains the problem description). Following, it evaluates the *CSA* recurring to the *CSAEvaluator* module. Regarding the *CSAEvaluator* module, no significant changes were made to it since there were no objectives that explicitly required such modifications.

If the *CSAEvaluator* "says" that everything is ok with the *CSA*, it can start the searching of alternatives for its *ServiceTemplates*. Once alternatives have been found for a *ServiceTemplate*, it moves onto the *Decision Module* to get the alternative's *preference graph*.

Once every *ServiceTemplate* has been processed, each will have its alternative's preference graph and is now possible to move onto the last step of the prototype: computing the aggregated solutions. To do so, it passes every preference graph to the *Aggregation Module* which, will compute the admissible aggregated solutions and then return them back to the *Controller* which is responsible for sending them to the GUI.

Figure.22 combines both *Algorithm 1* and Figure.21 to help one visualize the flow of execution of the CloudAid2.

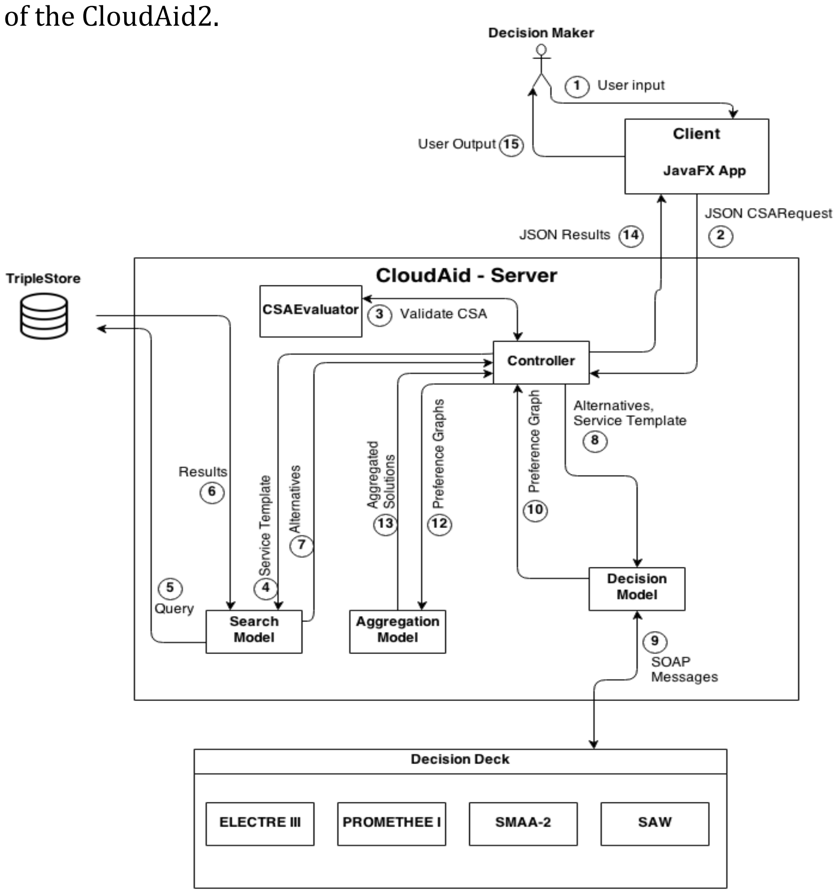


Fig. 22 - CloudAid2's execution flow

5.3.2. Search Module

Once the *CSA* has been validated by the *CSAEvaluator*, the *Controller* relies on the *Search Module* to find alternatives for each *ServiceTemplate* on the *CSA*.

To find the corresponding alternatives, the *Search Module* performs the following tasks:

1. Extract the exclusive requirements from the *CSA*.

2. Build the exclusive requirement's SPARQL query and perform the search over the TripleStore.
3. Use the Linked USDL Pricing API to load the alternatives information from the TripleStore.
4. Send a request through the *Controller* to the GUI to obtain the values for the pricing variables of the alternatives found.
5. Receive the response from the *Controller* and use the Linked USDL Pricing API to obtain the cost of each alternative.
6. Perform a price filtering if there's a price *Requirement* on the *ServiceTemplate*.

A major change on the *Search Module* when comparing it to the previous prototype is the inclusion of the Linked USDL Pricing API [50] to deal with the dynamic component of the Linked USDL service descriptions. Including the Linked USDL API eases the interaction of the ClouAid2 with the semantic side of the project thus, allowing me to focus on the logical behavior of the module.

Regarding task 1 and 2, their behavior is the same as on CloudAid1. The module starts by executing the *getExclusiveRequirements()* function to get the exclusive requirements from the *ServiceTemplate* being processed. Once they've been identified, it executes the *queryBuilder()* function to build the SPARQL query of the exclusive requirements. While their behavior is the same, the source code had to be reworked from scratch.

Once task 1 and task 2 have been completed, the module ends up with a *ResultSet*[39] that contains every *ServiceOffering Resource*[40] that matches the *ServiceTemplate's* requirements. **Appendix E** shows the SPARQL query built for *ServiceTemplate1* from the scenario presented on section 6.2.

On CloudAid1, the cost of each alternative was imbedded on the semantic description of the service as a feature with a static value. On CloudAid2, to obtain the price of the alternatives from the new semantic descriptions, the decision maker has to provide extra information related to the service he's looking for; more precisely, he/she needs to specify some *usage* parameters such as the time he's going to be using the service, or the amount of data he's expecting to receive on the machine, so the module can get cost of the alternatives and move onto the next step.

Considering this and the introduction made on section 5.1, on task 3 the *Search Module* uses the *Offering.readFromModel()* function to load the information from each Jena *Resource*, found on task 2, from the TripleStore.

Once task 3 is complete, the *Search Module* requests the *usage* information from the decision maker. The external communication related to this task was introduced on the previous section so I won't be going over it again however, this task encompasses an extra step that is yet to be addressed: getting the *Usage* variable's details from the *Offering* instances and, populate the *PriceVariable* and *PricingVariables* models for the external communication.

[39] http://jena.apache.org/documentation/javadoc/arq/com/hp/hpl/jena/query/ResultSet.html
[40] https://jena.apache.org/documentation/javadoc/jena/com/hp/hpl/jena/rdf/model/Resource.html

The models themselves are very simple, *PriceVariable* has a (*String*) name, a (*String*) description and a (*Integer*) value while the *PricingVariables* has a single *List<PriceVariable>*. Iterating over the *Offering* instances, one easily extracts their *Usage* variables information and populates the models.

Once the *PricingVariables* instance is populated, the *Search Module* calls the *requestVariablesInfo*() function from the Controller to send the JSON request and then stays on hold, waiting for its response. In this case, what the module is expecting is a new value for the *value* attribute of each *PriceVariable* instance. Once the *Controller* delivers the response back to the *Search Module*, the latter uses the method explained on section 5.1.4 to get the cost of the alternatives: it inserts the values provided by the Decision Maker on their *Usage variables*, creates their semantic description and then uses the *calculatePrice()* function to get the cost of each *Offering*.

Having their cost, the last step is to verify if there is a price requirement on the *ServiceTemplate*. As I mentioned earlier, on CloudAid1 the price was included on the semantic description of the service so filtering by price was possible at a SPARQL level however, on CloudAid2 the price is calculated dynamically. To cope with this situation, filtering service offerings by price ("At most, I'm willing to pay *y* for the service" or "At least, the service I'm looking for needs to cost *y*") is only performed once the cost of every alternative has been calculated. One can look at it as a second level filtering that is applied after the first level filtering (SPARQL query), where the *Search Module* discards every alternative that doesn't match every other (exclusive) requirement on the *ServiceTemplate*.

Once task 6 is finished and before returning the results back to the *Controller*, the *Search Module* encapsulates the remaining *Offering* instances in a new Java model (Figure.23). This Java model is quite useful since it enables me to append extra information that is not included on the *Offering* object and that might be of use for the following modules, the *Controller* or even the decision maker.

Table.5 presents the *Usage* variables for each use case presented on section 5.2.

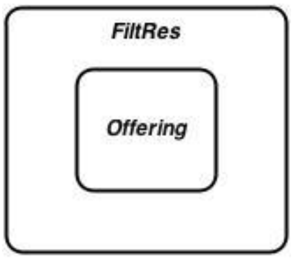


Fig. 23 - Offering
object encapsulation

Amazon EC2	Amazon RDS	Amazon Glacier	Amazon Load Balanc.
Name: *gbout (Gb)* **Description**: Number of Gb the decision maker expects to send out from Amazon EC2 to the Internet.	**Name**: *gbout (Gb)* **Description**: Number of Gb the decision maker expects to send out from Amazon RDS to the Internet.	**Name**: *gbToStore (Gb)* **Description**: Gb of data that the decision maker is expecting to store on Amazon Glacier	**Name**: *gbin (Gb)* **Description**: GB of data the decision maker is expecting to receive on the instance per month
Name: *usagehours (Hours)* Description: Number of hours that he'll be using Amazon EC2.	**Name**: *usagehours (Hours)* **Description**: Number of hours that he'll be using Amazon RDS.	**Name**: *retrieveData (Gb)* **Description**: Gb of data that the decision maker is expecting to retrieve from Amazon Glacier	**Name**: *amazonnlb (unit)* **Description**: Number of load balancers the decision maker needs
	Name: *IOPSWanted (unit)* **Description**: Number of I/O operations the decision maker expects to need/require per second.	**Name**: *retrievalTime (Hours)* **Description**: The amount of time that Amazon Glacier has to retrieve the quantity of data specified on the *retrieveData* variable	**Name**: *usagehours (Hours)* **Description**: Number of hours that he'll be needing the load balancers
	Name: *GBStorageWanted (Gb)* **Description**: Number of Gb the decision maker needs for his database.	**Name**: *NRequests (unit)* **Description**: Total number of Upload/Retrieval requests the decision maker is expecting to perform on Amazon Glacier.	
	Name: *usagedays (Days)* **Description**: Number of days that he'll be keeping the data on Amazon Glacier.	**Name**: *usagedays (Days)* **Description**: Number of days that he'll be keeping the data on Amazon Glacier.	
	Name: *NumberOfMonths (Months)* **Description**: Number of months that he'll be using Amazon RDS.	**Name**: *gbout (Gb)* **Description**: Number of Gb he expects to send out from Amazon Glacier to the Internet.	
Arsys Dedic. Servers			
Name: *NumberOfMonths (Months)* **Description**: Number of months that he'll be using Arsys dedicated servers.			

Table 5 - Repository's Usage Variables

5.3.3. Decision Module

Once the *Search Module* finishes its job, the *Controller* moves onto the next step: the *Decision Module*. Independently of the changes made to the module, its purpose remains the same: it provides an ordering (or semi-ordering) to the alternatives found by the *Search Module*.

To achieve this, it recurs to *Multi-Criteria Decision Methods (MCDM)*, the decision maker's preferences and the criteria he/she defined when creating his/her *CSA*.

Regarding the decision maker's preferences, unlike its earlier version, every piece of information that might be relevant for the decision process is requested when the decision maker is creating his/hers *CSA*. For example, on CloudAid1, once the decision process began, it would ask the user if he/she had a preferable value for each criterion he/she had defined earlier when creating the CSA. This no longer occurs, if the decision maker has a preferable value, he/she needs to define it beforehand. The only information that is requested during the module's execution is related to qualitative values distances, e.g.: the numerical distance between *high* performance and *moderate* performance.

Regarding the *MCDM* component, CloudAid1 included two different methods: the Analytic Hierarchic Process (AHP) and the Simple Additive Weight (SAW). For the new prototype however, new *MCDM* that take into consideration incomparability/incomplete information between alternatives were a requirement. As such, new methods were integrated into the module:

- ELECTRE III
- PROMETHEE I
- SMAA-2

Each of the methods has its own data requirements, different workflows and provide different results however, independently of the method used, there are several steps that are always executed:

1. **Data normalization** – The data normalization is performed in the same way as in the CloudAid1 but, with two minor distinctions:

   a. A new parameter was included into the criterions: *thresholds*, more precisely, *preference* threshold, *indifference* threshold and *veto* thresholds [14, 15 and 16]. To normalize these new attributes, a simple mathematical expression is used: *MIN(MAX(l/(a-b),0),1*), where *l* is the threshold value provided by the decision maker, *a* the minimum value of the criterion and *b* the maximum value of the criterion.
   b. CloudAid1 considered three types of attributes: Numerical, Non-Numerical binary and Non-Numerical non-binary. Regarding non-numerical attributes, the distinction between binary and non-binary required that either the user had a previous knowledge about the information on the TripleStore or the attribute in question had only two possible values (0 and 1) [7]. To not only simplify the prototype, but also to optimize the decision itself, this distinction between binary and non-binary was removed. It may happen that, despite having only two possible values, both of them are not the ideal value for the decision maker. E.g.: The decision maker is looking for a service with *performance = Excellent* but, amongst the found alternatives there's only two possible values: *moderate* and *high*. While it is, indeed, a binary attribute, neither of them is the *best/ideal* value so neither of them should be represented by "1", which is the highest numerical value a qualitative value can have. To request the concept distances from the decision maker, it uses method similar to the one used by the

Search Module to request the *Usage* variables values however, it uses the *ConceptDistance, DistancesContainer* Java models and the *requestDistancesInfo()* method instead.

2. **Express the decision problem into XMCDA format** (section 5.3.3.1) – Once normalized, the data needs to be described into XMCDA format before it can be passed onto the *MCDM*.
3. **Encapsulate the XMCDA decision problem in a SOAP message and send it to the corresponding *web-service*** (section 5.3.3.2) – After describing the decision problem into XMCDA format, the *Decision Module* encapsulates it in a *SOAP* message and starts the chosen *MCDM*'s workflow.
4. **Receive the *MCDM* results and create the alternatives' preference graph** (section 5.3.3.3) – Once the *MCDM's* workflow ends, the *Decision Module* has to process the received XMCDA results in order to create the alternatives' preference graph (more precisely, its adjacency list) that will be used by the *Aggregation Module* to compute the aggregated solutions.

5.3.3.1. Decision Deck, XML Encoding of Multi-Criteria Decision Aid Data

The decision deck project aims at developing tools that provide Multiple Criteria Decision Aid methods that can be of help to anyone who needs them, whether they are teachers who need the material for didactic purposes or practitioners who may need MCDA tools to support real world decision problems. To achieve this, the group implemented several Multi-Criteria Decision Methods on the programming languages they felt more comfortable with and made them accessible to everyone through web-services, using SOAP messages and XMCDA format [28,38] (Section 5.3.3.2).

To express the decision problem into XMCDA format, I recurred to the J-XMCDA [41] library, a Java library provided by the Decision Deck group to deal with XMCDA transformations. Every operation related with XCMDA data manipulation is wrapped in the *XMCDAConverter* class.

On CloudAid1, Eng. Jorge Araújo had control over both sides of the module, the prototype's side which was responsible for creating and sending the problem over to the *MCDM* and, the external *MCDM* applications that processed the problem sent by the prototype. As such, there wasn't a big concern regarding the XMCDA *tags* (from the *MCDM* external applications perspective) used to map the problem's data. On CloudAid2, greater care had to be given to the XMCDA data description since it replaced the old external *MCDM* applications for *MCDM* made available through *web services* hosted by the Decision Deck group. These *web services* are expecting a decision problem described into a XMCDA file with a pre-defined structure.

Currently, there's only one *MCDM* that doesn't require criterion weights for its calculations: the SMAA-2[42] method. Every other method requires an importance weight associated to every criterion in every *Service Template*.

[41] http://www.decision-deck.org/xmcda/library.J-XMCDA.html

[42] http://www.decision-deck.org/ws/wsd-smaa2-jsmaa.html

Table.6 presents a list of the methods in the *XMCDAConverter* class that are used by the *Decision Module* to express the decision problem into XMCDA format, their XMCDA tags and their description.

Method	XMCDA Tag
createAlternatives(alternatives)	alternatives, alternative
Description: Creates the list of alternatives in the XMCDA object.	
createCriteria(ServiceTemplate)	criteria,criterion,scale,quantitative,preferenceDirection
Description: Creates the criteria list in the XMCDA object. To each criterion is assigned a preference direction and its thresholds, if there's any.	
createWeights(ServiceTemplate)	criteriaValues,criterionValue,criterionID,value,real
Description: Creates the criterion's weights list in the XMCDA Object.	
createAlternativesValues(alternatives,critID's)	performanceTable,alternativePerformances,alternativeID, performance,criterionID,value,real
Description: Creates a list of the alternatives values for each defined criterion in the XMCDA object.	

Table 6 - XMCDAConverter methods

Once the decision problem has been normalized and described in XMCDA, the module is ready to move onto the next step: encapsulate the XMCDA problem in a *SOAP* message and execute the chosen *MCDM's* workflow.

5.3.3.2. SOAP encapsulation and *MCDM* workflow

Now that the decision problem has been described into XMCDA format, we just need to encapsulate it in a *SOAP* message and send it to the Decision Deck *MCDM's web services*. **Appendix F** presents an example of the *SOAP* messages exchanged.

Currently, CloudAid2 supports four different *Multi-Criteria Decision Methods*: ELECTRE III[43], PROMETHEE I[44], SAW[45] and SMAA-2[41]. Each of these methods has its own workflow that needs to be executed sequentially in order to obtain the results we're looking for. Figures 24, 25, 26 and 27 present the workflows for each method, respectively. These figures were created with the help of *diviz*[30, 31] , an application developed by the Decision Deck group that provides a graphical interface to ease and improve the usability of their *MCDM*.

An interesting aspect is that once the decision problem is expressed into XMCDA, it can be used by either the CloudAid2 prototype or the *diviz* application. Doing so also provides a decent validation for the prototype's XMCDA descriptions.

[43] http://www.decision-deck.org/diviz/workflow.methodElectre3.html
[44] http://www.decision-deck.org/diviz/workflow.methodPromethee.html
[45] http://www.decision-deck.org/ws/wsd-weightedSum-PyXMCDA.html

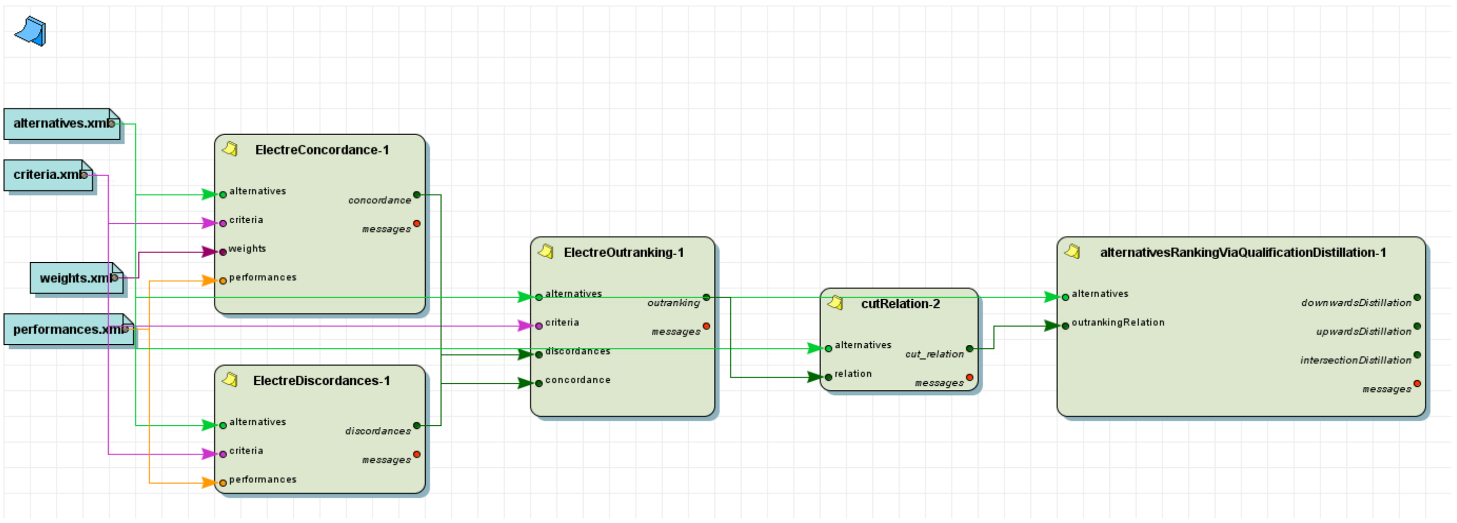


Fig. 24 - ELECTRE III workflow

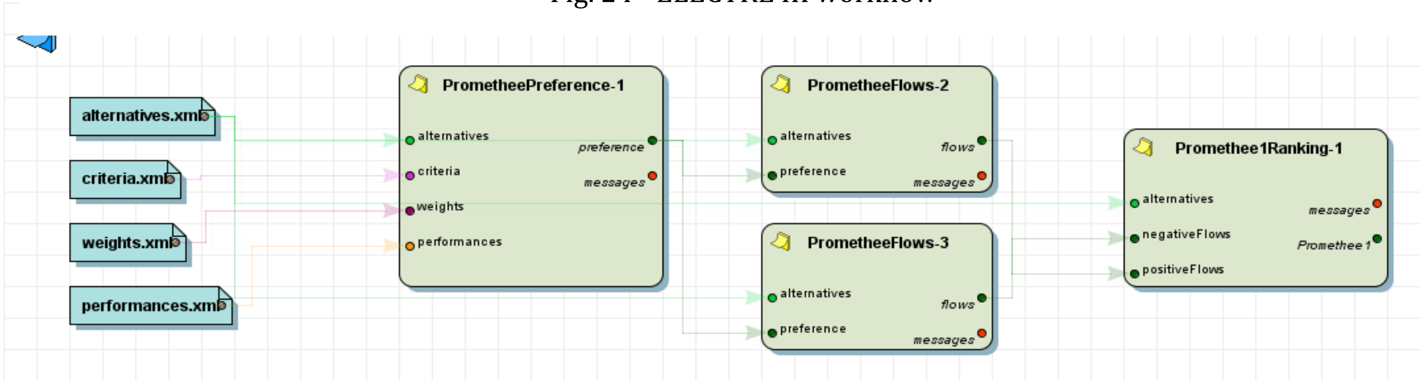


Fig. 25 - PROMETHEE I workflow

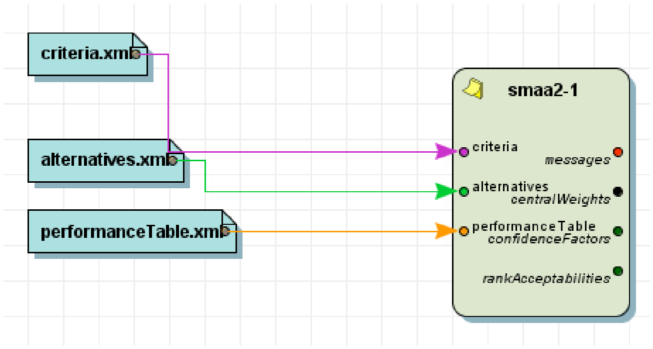


Fig. 26 - SMAA-2 workflow

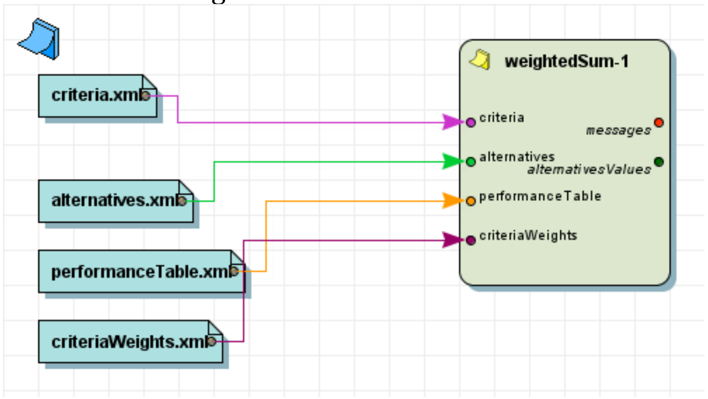


Fig. 27 - SAW workflow

To execute each method's workflow and obtain their results, a specific Java class for each of them was created:

- *ELECTRE*
- *PROMETHEE*
- *SMAA*
- *SAW*

Each of these classes has a static method named *solve()* that initiates the execution of their workflows and, depending on the chosen method, the workflow's complexity differs. For example, ELECTRE III is the most complex *MCDM* method integrated on the prototype as it's composed of 5 *web services*. Each of these *web-services* needs to be executed sequentially (let's call them *steps*) and each needs the results from the previous *step* to perform its calculations and move onto the next *step*.

Of course, depending on the complexity of the submitted problem, the methods execution may vary from seconds or minutes to hours. To work around this problem, they implemented a ticket mechanism that enables users to terminate their current connection to the *web service* and retrieve their results later using the ticket sent back in the response. For example, when we submit a request to the *ElectreConcordance-1* (*step 1* of the ELECTRE III method), the *web service* sends a response containing a ticket (a String with alphanumeric characters) that we can use later to fetch the results.

To simplify the development of the external communication, everything related to the *SOAP* message encapsulation, connection establishment and ticket/response management was wrapped in the *DecisionDeckSOAPClient* Java class. Doing so eases the understanding of the code, keeping the logical execution of the method from the technical aspects of the communication itself. Table.7 presents the most important methods of the class and a brief description of them.

***DecisionDeckSOAPClient* methods**
sendProblem(XMCDAfiles,XMCDAalias,WebServiceName)
Description: Establishes a *SOAP* connection with the *web service* and submits the XMCDA request.
createSOAPRequest(XMCDAfiles,XMCDAalias)
Description: Used by the *sendProblem* method to encapsulate the XMCDA files into a *SOAP* message.
getProblemResponse(WebServiceName)
Description: Establishes a *SOAP* connection with the *web service* and submits a *SOAP* message to request the solution of a previously submitted problem.
getProblemSolutionMessage(TicketID,TicketNumber)
Description: Method used by the *getProblemResponse* method to create the *SOAP* message that requests the solution for a submitted problem. The message contains the ticket number and ticket id of the request.
parseSubmittedResponse(SOAPResponse)
Description: Method that parses the response received from the *web service* and verifies if the submitted problem was successfully solved.

Table 7 - DecisionDeckSOAPClient Java class methods

Using the *DecisionDeckSOAPClient* class, executing the *MCDM* workflows becomes simple. Algorithm 2 presents, in a simple way, how the *PROMETHEE* class executes PROMETHEE I's workflow.

```
DecisionDeckSOAPClient client = new DecisionDeckSOAPClient();
do
    preference = step1(XMCDAfiles,client)
while(preference == null)

if(preference != null)
    ArrayList<XMCDA> cont = new ArrayList<XMCDA>();
    cont.add(files.get(0));//alternatives XMCDA
    cont.add(preference);
    do
       positiveFlow = step2(cont,client)
    while(positiveFlow == null)
endif

if(positiveFlow != null)
    ArrayList<XMCDA> cont = new ArrayList<XMCDA>();
    cont.add(files.get(0));//alternatives XMCDA
    cont.add(preference);
    do
       negativeFlow = step3(cont,client)
    while(negativeFlow == null)
endif

if(positiveFlow != null && negativeFlow != null)
     ArrayList<XMCDA> cont = new ArrayList<XMCDA>();
     cont.add(files.get(0));
     cont.add(positiveFlow);
     cont.add(negativeFlow);
     do
        promethee1Ranking = step4(cont,client)
     while(promethee1Ranking == null)
endif

return promethee1Ranking
```

Algorithm 2 – PROMETHEE I workflow execution

The *solve()* method of the PROMETHEE I executes four *steps*, as depicted on Fig.25. Each *step* function uses the *DecisionDeckSOAPClient* instance (client) to submit the problem to the corresponding *web service* (e.g.: *step4* of PROMETHEE I submits the problem to the *Promethee1Ranking-1 web service*) and to retrieve its results.

Every other *MCDM* workflow's execution is similar to the one depicted on Algorithm 2, but adapted to its situation. Once their workflow execution terminates, we'll have a XMCDA description of the results of the chosen method.

Every every XMCDA file, be it the initial problem description, intermediary results or the final result, are stored on a file located on a pre-defined directory (server side). For example, the *PROMETHE* class will store every XMCDA file used on its calculations on "*PROMETHEE/Request_TimeSent*", a local directory of the server. This way, it's possible to re-use results, check for inconsistencies or even to use them with *diviz* application for results comparison/validation.

5.3.3.3. Results processing

Once task 3 is finished, the *Decision Module* moves onto the next, and final, task: the creation of the alternatives preference graph. A preference graph describes the relations between each pair of alternatives, telling us which one is preferred.

Contrary to its predecessor, ranked lists are no longer capable of describing the results of the new *Multi-Criteria Decision Methods* since the concept itself implies a complete ordering of alternatives, where we know at all times which alternative is preferred to which. Achieving a complete ordering is not always possible, a situation might occur where we're unable to tell which alternative is preferable to which thus, classifying them as incomparable. If two or more alternatives are incomparable with one another, they need to "share" the same *rank* for they need to be treated equally.

Recurring to graphs rather than ranked lists, tackling this type of situations becomes simpler.

Fig.28 presents a simple example of a preference graph.

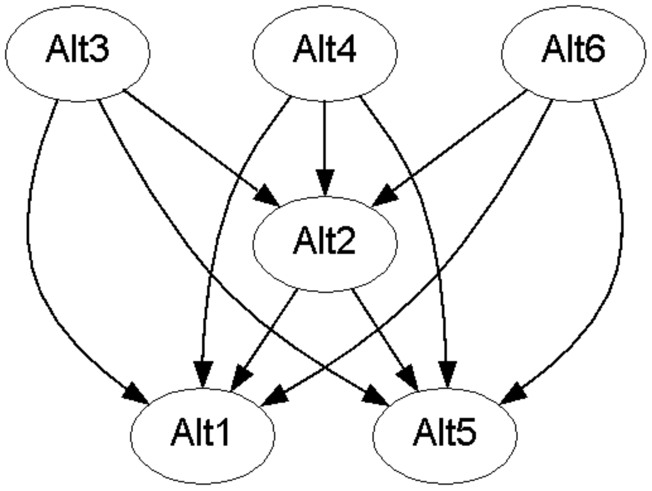


Fig. 28 - Preference Graph example

In a ***complete transitive preference graph*** (from now on it'll be referred simply as preference graph) there's only two possible situations for each pair of alternatives: Let *x* and *y* be two random alternatives from a preference graph

- If a connection (arc) *e = (x, y)*, where *x* is the *head* and *y* the *tail*, exists, then *x* is preferable to *y*.

- If a connection (arc) *e = (x, y)* doesn't exist, then *x* and *y* are incomparable and should be treated equally (strictly speaking, it's more accurate to say that no information about their preference relation can be inferred).

Looking at Fig.26, we can say that Alt3, Alt4 and Alt6 are incomparable but, all three are preferable to Alt2, Alt1 and Alt5. The same applies to Alt1 and Alt5 however, neither of the two has *outgoing* arcs which means that no matter our choice, these two should always be our last resort for they aren't preferable to any other alternative.

The challenge was how to transform each *MCDM's* result into a preference graph like the one depicted on Fig.28.

Before I move to the preference graph creation, first, it's necessary to introduce the results of each method and the class used to represent a *node* of the graph: *GNode*.

GNode is a Java class that contains the number of *outgoing* connections (number of alternatives that are inferior to the alternative in question), the number of *incoming* connections (number of alternatives that are superior to the alternative in question), an *ArrayList<GNode> preferableTo,* that contains the alternatives to which it has an existing connection (the ones upon which it's preferable to), and a second *ArrayList<GNode>, incomparableWith,* that contains the alternatives that are incomparable with the alternative in question. Fig.29 illustrates the GNode encapsulation regarding the other data models of the prototype (note that while *Offering* is extensively used by the prototype, it's a model from the Linked USDL Pricing API).

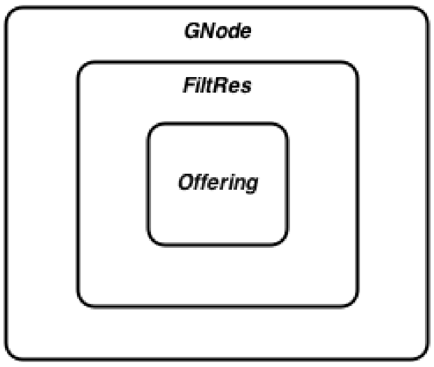


Fig. 29 - GNode encapsulation

PROMETHEE I and ELECTRE III results

The results from PROMETHEE I and ELECTRE III are equal, what differs is the calculations they perform to achieve said results. Once their workflows execution ends, the XMCDA returned by the *web service* contains the preference relations for each alternative. For example, using Fig.28, both PROMETHEE I and ELECTRE III results would look something like the following:

- Alt3 is preferable to Alt2
- Alt3 is preferable to Alt1
- Alt3 is preferable to Alt5
- Alt1 is preferable to Alt4
- ...
- Alt2 is preferable to Alt1
- Alt2 is preferable to Alt5

In this case, extracting preference relations for each pair of alternatives is simple, there's no need to include intermediary steps to do so:

- An arc/connection *e = (x, y)* is mapped as adjacency_matrix[x, y] = 1 and adjacency_matrix[y, x] = -1. E.g.: *adjmatrix[Alt3,Alt2] = 1, adjmatrix[Alt2,Alt3] = -1*;
- The absence of an arc e = (x, y) is mapped as adjacency_matrix[x,y] = adjacency_matrix[y,x] = 0. E.g.: *adjmatrix[Alt3,Alt4] = adjmatrix[Alt4,Alt3] = 0*;

Creating the corresponding preference graph from the adjacency matrix is simple: we just need to create an instance of GNode for each alternative, iterate over the matrix's lines and columns, and update the information of the GNode's instance.

For example, considering Fig.28 again, its adjacency matrix would be:

	Alt1	**Alt2**	**Alt3**	**Alt4**	**Alt5**	**Alt6**
Alt1	0	-1	-1	-1	0	-1
Alt2	1	0	-1	-1	1	-1
Alt3	1	1	0	0	1	0
Alt4	1	1	0	0	1	0
Alt5	0	-1	-1	-1	0	-1
Alt6	1	1	0	0	1	0

Iterating over the adjacency matrix, a populated GNode instance for Alt3 would look something like the following:

- Outgoing_arcs – 3
- Incoming_arcs – 0
- PreferableTo – Alt2, Alt5 and Alt1
- IncomparableWith – Alt4 and Alt6
- FiltRes – Alternative3

Once every alternative has their corresponding GNode instance, the preference graph is built. At least, for the results obtained by the PROMETHEE I and ELECTRE III methods.

SMAA-2 results

Unlike PROMETHEE I or ELECTRE III, SMAA-2 returns the probability that each alternative has to be placed on a specific *rank*. For example, consider a *Service Template* to which the *Search Module* found 5 matching alternatives; a possible SMAA-2 result would be something like:

Alternative\Rank	1º	2º	3º	4º	5º
Alt1	10%	15%	25%	35%	15%
Alt2	0%	36%	39%	12%	13%
Alt3	30%	17%	47%	5%	1%
Alt4	4%	42%	28%	19%	7%
Alt5	65%	12%	5%	18%	0%

Table 8 - SMAA-2 result example

To create an adjacency matrix and therefore, a preference graph, from these results, we provide an ordering to the alternatives by placing each alternative on its *rank* with highest probability thus, creating preference relations between alternatives. In the ideal scenario, we end up with a fully ordered list of size N, where N is equal to the number of alternatives and, where there is only one alternative per *rank*. Of course, it's a lot more common to be faced with the situation where two or more alternatives end up sharing the same rank. For example, looking at Table.8, Alt2 and Alt3 would both be placed third place in the *ranked list* thus, becoming a semi-ordered list.

Regarding this placement of alternatives, there's still one particular issue to address: Alternative N may have very similar (or even equal) probabilities for multiple *ranks*. For example, looking at Table.8, should we place Alt2 second or third? Is 3% difference enough to place Alt2 in third place? It'd be more accurate to say that Alt2 has a ~50% chance of being placed second or third. To identify this type of situations, an *indifference threshold* [14, 15 and 16] was included into the placement method: Let,

- ***d*** be the percent threshold of indifference between two alternatives;
- ***ß*** the list of possible ranks for Alternative N, where N ranges from 1 to the total number of alternatives;
- ***P(y), y ∈ß***, the probability of *rank **y***;
- ***P(x), x ∈ß,*** the probability of *rank x*, *x* being the rank with highest probability in ***ß;***

If ***P(x) - P(y) < d***, Alternative N will be randomly placed on either rank *x* or *y*. Going back to the Alt2 and Alt3 scenario, with ***d*** = 4%, Alt2 would have an ~50% chance of being placed third thus, sharing the rank with Alt3 or, being placed second and share the rank with Alt4. Table.9 presents a possible placement for the alternatives.

Rank (*d=4%*)	Alternative
1º	Alt5
2º	Alt2,Alt4
3º	Alt3
4º	Alt1

Table 9 - SMAA-2 semi-ordered list example

Once the semi-ordered (or fully ordered) ranked list is constructed, we can create the adjacency matrix of the preference graph. Each alternative on rank *m* is incomparable with every other alternative on the same rank, preferable to every alternative placed on ranks lower than *m* and inferior to alternatives placed on ranks higher than *m*. We can relate this information to the relations mapped by 0, 1 and -1, respectively, in the adjacency matrix of the preference graph; we just need to iterate over the list, starting at the top and build the adjacency matrix.

Having the adjacency matrix, we build the preference graph just like we did with ELECTRE III's and PROMETHEE I's results.

SAW results

SAW's results processing method and SMAA-2's results processing method are very similar, they simply differ in the actual result of the *MCDM* itself: while SMAA-2 returns the probability that each alternative has of being placed on a specific rank, SAW returns a *performance* value for each alternative that represents how good the alternative is over the defined criteria. The higher the *performance* value, the better the alternative. Table.10 presents an example of a possible result obtained by the SAW method.

Alternative	**Performance**
Alt1	0.84
Alt2	0.14
Alt3	0.28
Alt4	0.71
Alt5	0.28

Table 10 - SAW results example

Using the alternatives *performance* value (*pv*), we're capable of creating a semi-ordered (or fully ordered) list, just like we did with the SMAA-2 method and its *ranking* probabilities. In this case however, the placement method is simpler since there's only one *performance* value per alternative thus, removing the "need" for the *indifference threshold* mentioned earlier (we could still apply it over *performance values* between different alternatives; e.g.: AltN(*pv) = 0.15* and AltM(*pv*) = 0.17 could be considered incomparable and be placed on the same rank). To create the alternatives ordering, we start by placing the alternative(s) with the highest *performance* value first on the list (alternatives with equal *performance* values are grouped together and placed on the same *rank*), remove it/them from the group of alternatives that still need to be placed on the list and, repeat these two steps until there are no more alternatives left to placed (ranked). Table.11 presents the semi-ordered list obtained from the results depicted on Table.10 (semi-ordered because Alt3 and Alt5 have equal *performance* values thus, being placed on the same *rank* and marked as incomparable. Should Alt5 have a different *performance* value from every alternative, we'd be dealing with a fully ordered list instead of a semi-ordered list).

Rank	Alternative
1º	Alt1
2º	Alt4
3º	Alt3,Alt5
4º	Alt2

Table 11 - Table.10 semi-ordered list description

Once the list is constructed, we can create the corresponding adjacency matrix just like we did on SMAA-2: Each alternative on rank *m* is incomparable with every other on the same rank, preferable to every alternative placed on ranks lower than *m* and inferior to alternatives placed on ranks higher than *m*. From the adjacency matrix we're able to create the alternatives preference graph.

Each method has its results described on XMCDA. To transform these results into a more comfortable data structure for the adjacency matrix construction/extraction, several methods were added into the *XMCDAConverter* class. Table.12 presents these new methods, the data structures they use to map the XMCDA results and the methods that use said data structures to create the adjacency matrix/preference graph.

DecisionCore.method, uses →	***XMCDAConverter.method, creates →***	***Java class***
PROMETHEE I and ELECTRE III		
getComparabilityGraphSolution(CR, dir)	*processComparabilityResults(res, st)*	*ComparabilityResult*
Description: Uses the adjacency matrix created by the *processComparabilityResults* to create a preference graph where each node is an instance of GNode. **Parameters**: *CR* – ComparabilityResult instance that contains an adjacency matrix. *Dir* – Local directory path to save the DOT[46] description file of the graph.	**Description**: Creates the adjacency matrix of a preference graph from the ELECTRE III's and PROMETHEE I's XMCDA results. **Parameters**: *st* – Service Template being evaluated. *res* – XMCDA results returned by the *MCDM's* web service.	**Description**: Container for the adjacency matrix created by the *processComparabilityResults* method.
SMAA-2		
getSMAAGraphSolution(SR, delta, dir)	*processSMAAResults(res, st)*	*SMAAResults,SMAAResult*
Description: Uses the list that contains the alternatives and their corresponding probabilities for each rank, to create the adjacency matrix of the preference graph. Afterwards, it creates the preference graph from the adjacency matrix. **Parameters**: *SR* – SMAAResults instance that contains the list of alternatives and their *ranking* probabilities. Delta – Maximum Indifference threshold between rank probabilities of an alternative. *dir* – Local directory path to save the DOT[45] description file of the graph.	**Description**: Creates a list of alternatives and their corresponding *rank* probabilities, from SMAA-2's XMCDA results. **Parameters**: *st* – Service Template being evaluated. *res* – XMCDA results returned by the *MCDM's* web service.	**Description**: *SMAAResult* – Contains an alternative (FiltRes instance) and its ranking probabilities. *SMAAResults* – Contains a list of *SMAAResult* instances.

[46] http://en.wikipedia.org/wiki/DOT_(graph_description_language)

SAW		
getSAWGraphSolution(SR,dir)	*processSAWResults(res, st)*	*SAWResult,SAWResults*
Description: Uses the list of alternatives and their *performance* values to create the preference graph's adjacency matrix and therefore, the preference graph. **Parameters**: SR – SAWResults instance that contains the list of alternatives and their *performance* value. *dir* – Local directory path to save the DOT[45] description file of the graph.	**Description**: Creates a list of alternatives and their corresponding *performance* value from SAW's XMCDA results. **Parameters**: *st* – Service Template being evaluated. *res* – XMCDA results returned by the *MCDM's* web service.	**Description**: SAWResult – Contains an alternative (FiltRes instance) and its *performance* value. SAWResults – Contains a list of SAWResult instances.

Table 12 - XMCDA conversion and preference graph construction

The final result of the *Decision Module* and therefore, the result that is sent back to the *Controller*, is always a preference graph where each node is an instance of **GNode**. In fact, it's more accurate to say that what is built/returned by the *Decision Module* is the preference graph's adjacency list rather than the graph itself. Through the adjacency list, the *Aggregation Module* is able to traverse the graph and compute the aggregated solutions (Section 5.3.4).

Before returning the results back to the *Controller*, the module executes these three final steps, independently of the chosen method:

1. **Create a DOT description of the preference graph and write it into a file** – DOT is a simple plain text language used to create graph descriptions that are easy to read for both humans and machines [54]. Once the preference graph is described in DOT language, the module writes this description into a file located on the same directory as the XMCDA files. These DOT descriptions can, in turn, be loaded by applications like *Graphiz*[47] or *ZRGViewer*[48], which are capable of "understanding" this descriptions and create a graphic representation of the graph (E.g.: Fig.28).

2. **Sort the preference graph's adjacency list in ascending order, using the number of incoming arcs** - Using *Collections.sort()*[49], sort the following components in ascending order, using the number of incoming arcs from each of its GNode instances, as the sorting criteria (Section 5.3.4 addresses this step with further detail):
   a. The adjacency list
   b. Every GNode's *preferableTo* and *incomparableWith* lists.

3. **Detect and remove possible cycles in the graphs** – The preference graphs generated from SMAA-2's and SAW's results can't contain cycles in the graph since every alternative that is on a given rank *m* will have connections to, at maximum, *(N-y-z)* alternatives where: *N* is the total

[47] http://www.graphviz.org/
[48] http://zvtm.sourceforge.net/zgrviewer.html
[49] http://docs.oracle.com/javase/6/docs/api/java/util/Collections.html

number of alternatives (nodes); *y* the number of alternatives on *rank m; z* the number of alternatives placed above rank *m*; and *(N-y-z)* the number of alternatives that are inferior to *y*. Basically, *y* will never be preferable to any alternative placed on a rank higher than *m*. With ELECTRE III and PROMETHEE I, this is not the case. We cannot guarantee that there won't be any cycles on the graph once it has been built. As such, I included an efficient algorithm to detect elementary cycles on directed graphs, based on Tarjan's algorithm[50], developed by Donald B. Johnson [55]. I won't be going into much detail about the algorithm or how it was implemented for it falls out of the scope of the report however, it's important to note its time complexity, finding every cycle within the graph in *O(n)* time. If it does find any cycles, the module is responsible for marking every node in the cycle as incomparable with every other alternative in the cycle.

Once these 3 steps have been concluded, the module terminates and returns the sorted adjacency list back to the *Controller*. Once every *Service Template* has been processed, it can move onto the next step of execution: computing the aggregated solutions.

5.3.4. Aggregation Module

Contrary to the *Search Module* and *Decision Module*, the *Aggregation Module* is executed only once per each *CSA*, has shown in Algorithm.1. The *Search Module* finds suitable alternatives for each *Service Template* and the *Decision Module* creates the preference graph's adjacency list for the alternatives found by the *Search Module*. The *Aggregation Module* in turn, uses the preference graphs returned by the *Decision Module* to compute the results the decision maker is looking for – the *admissible* aggregated solutions.

Like in the previous prototype, every *admissible* aggregated solution is composed of one alternative from each *Service Template*. Fig.30 presents an example of an aggregated solution for a *CSA* with three *Service Templates*.

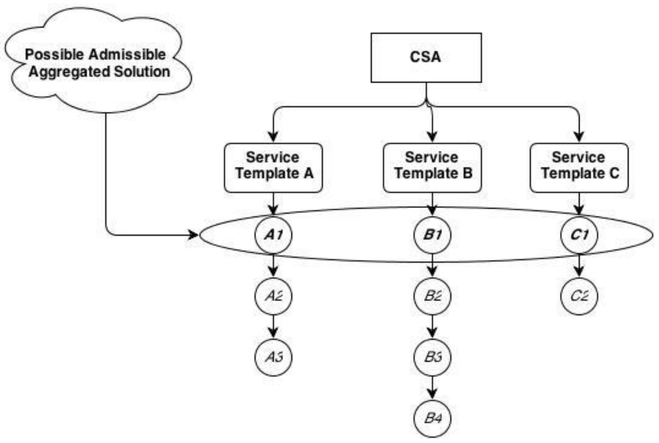


Fig. 30 - Aggregated Solution Example

[50] http://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm

To extract these aggregated solutions from the alternatives preference graph, the *Aggregation Module* uses an algorithm that was developed in this thesis.

Aggregation Algorithm

As previously stated, the aggregation algorithms developed on CloudAid1 had a very limited support for incomparability/incomplete information between alternatives. Given the results of the new *MCDM* methods, the need for a new aggregation algorithm was assured. While the previous algorithms are incompatible with the chosen data structure, they served as base of study and analysis for the development of the new aggregation algorithm.

The first step was to uniformize the *MCDM's* results: creating the preference graphs. Once we have the preference graph from each *Service Template* we can traverse them, starting at their roots and choosing the next alternative to visit (select) using the root's arcs (their preference relations, the alternatives that we can go to once we visit a certain alternative). When choosing an alternative, there're three questions that need to be taken into consideration:

1. Where do we start? How do we chose the root of the graph from the adjacency list?

2. Having selected the graph's root, how do we chose the next alternative to visit?

3. If we're traversing the graph using only the outgoing connections from the currently visited alternative, how do we deal with the inexistence of connections between alternatives (incomparability)?

These questions are related to the traversal of the graph, that is, the alternative that we want to visit next.

The sorting of the adjacency mentioned on section 5.3.3.3 addresses the first two questions. Fitting the graph into the context of the project, the root of the graph needs to be the *least dominated* alternative that is, the alternative that possesses the least incoming connections thus, ensuring that we always start by choosing/vising the best alternative(s) from the group of alternatives.

Once the root (first element of the list) has been identified, we start by visiting it and marking it as visited. From the root, we can move onto the next alternative by choosing an alternative from its *preferableTo* list which, is also sorted according to the number of *incoming* connections (keep in mind that each node is an instance of GNode). By sorting the *preferableTo* and *incomparableWith* lists from each *GNode*, we ensure that the next alternative we visit (which is always the first element of the list) will be the least dominated alternative from the remaining alternatives to visit. Marking a node as visited ensures that there's no node visited twice.

If we were dealing with a fully ordered list, every alternative on the preference graph would have a connection to every another alternative, be it an *incoming* or *outgoing* arc, thus, every node would be visited sooner or later. When dealing with incomparability/incomplete information, the preference graph contains unconnected alternatives which, requires extra measures before we choose the next alternative to visit.

For example, consider Figure.28 from section 5.3.3. Suppose that *Alt3* is the root of the graph and that we only use the outgoing connections from *Alt3* to other alternatives (alternatives to which *Alt3* is always preferred). This would be the path performed by a moving piece on the graph: *Alt3(0) → Alt2(3) -> Alt1(4)* where (*x*) represents the number of incoming connections of the alternative. *Alt4, Alt6* and *Alt5* wouldn't be visited, and therefore not tested, for there's no outgoing connection to any of them from any alternative that was visited.

As mentioned earlier, the inexistence of a connection between two alternatives means that they're incomparable and, we know that alternatives that are incomparable need to be treated equally. As such, if we visit *Alt3*, we need to visit *Alt4* and *Alt6* as well before we visit *Alt2*. It's only after visiting *Alt3, Al4* and *Alt6* that we can visit *Alt2*, and so on. The list of incomparable alternatives of every alternative is located on the *incomparableWith* list of its *GNode* instance.

Using this approach, we provide an answer to question 3 and we make sure there's no node left to visit on the graph. Now, the path performed by the moving piece would be: *Alt3(0) → Alt4(0) → Alt6(0) → Alt2(3) → Alt1(4) → Alt5(4)*.

Now that we can traverse and select the next alternative to be tested correctly from the preference graphs, we can move onto the actual aggregation algorithm. Combining the study and analysis of aggregation algorithms developed on [7] with this knowledge, it was possible to develop a new *Breadth-First Search*[51] based aggregation algorithm to traverse the aggregated solutions tree and, reduce its computational cost by applying a method similar to *Branch and bound*[52]. Algorithm.3 presents the *pseudo-code* of the new aggregation algorithm:

[51] http://en.wikipedia.org/wiki/Breadth-first_search
[52] http://en.wikipedia.org/wiki/Branch_and_bound

Algorithm.3 – Aggregation Algorithm

```
Create queue Q
ArrayList<GNode> solution

for(GNode root in preferenceGraphsRoots)
   solution.add(root)
end for

Q.add(solution)

for(k < solution.size())
   incomparables = getIncomparables(solution.get(k))
   ArrayList<GNode> newSolutions = replaceAlt(solution,k,incomparables)
end for

while(Q is not empty)
    node ← Q.poll()
    add node to tested
    if(node is admissable)
       if(node is not dominated or is incomparable with some admissable)
          admissables.add(node)
       end if
    else
       for(p < node.size())
          ArrayList<GNode> children = moveForward(node.get(p),node,p)
          for(child in children)
             if(Q not contains child and tested not contains child)
                if(child is not dominated or is incomparable with some admissable)
                   Q.add(child)
                end if
             end if
          end for
    end if
end while
```

The algorithm starts by placing into the *queue* the *best* aggregated solution possible, which is the solution composed of by the roots of every preference graph. Afterwards, *getIncomparables(k)* verifies if element *k* of the solution is incomparable with any alternative. If it is, it replaces *k* within the aggregated solution with its incomparable alternatives, creating new *best* aggregated solutions, and then adds them to the *queue*. For example, consider a *CSA* with two *Service Templates* and that each of these *Service Template* has a preference graph equal to the one depicted on Fig.28 of Section 5.3.3.3. If *Alt3* had no incomparable alternatives, the root of the *aggregated solution's tree* would be [*Alt3*, *Alt3*] however, *Alt3* is incomparable with *Alt4* and *Alt6* which means that the tree has 6 possible choices for its root: [*Alt3,Alt3*] – [*Alt3,Alt4*] – [*Alt3,Alt6*] – [*Alt4,Alt3*] – [*Alt6,Alt3*] and [*Alt3,Alt3*]. Every one of these solutions will be placed in the *queue* and tested by the algorithm. Only then, will it test any solution containing *Alt2*.

Once the root(s) of the tree has been placed on the *queue*, the algorithm will start traversing the tree. It fetches the first element of the *queue*, let's say that it's [*Alt3, Alt3*]. It marks this solution has tested and, afterwards verifies if the solution passes the *admissibility* test (section 5.3.4.1). If it does, it'll compare it with every admissible solution found so far, checking for dominance or incomparability. If the solution is not dominated by any solution or, if it's incomparable with all of them, it's placed onto the list of final solutions else, it's discarded. Afterwards, it terminates the search on that branch of the tree.

On the other hand, if the solution being tested fails the *admissibility* test, the algorithm needs to move one step forward in the tree to fetch the children of the solution. Supposing [*Alt3, Alt3*] failed the *admissibility test* and the algorithm requested its children using the *moveForward()* method, the new solutions that'd be placed into the *queue* would be: [*Alt2, Alt3*] and [*Alt3, Alt2*]. However, we know that there're some solutions missing since *Alt3* is incomparable with some alternatives (if we choose *Alt3*, we need to choose *Alt4* and *Alt6* as well). The full list of children for [*Alt3, Alt3*] would be: [*Alt2, Alt3*] – [*Alt2, Alt4*] – [*Alt2, Alt6*] – [*Alt3, Alt2*] – [*Alt4, Alt2*] and [*Alt6, Alt2*]. Once its children have been identified, it verifies if any of them is already on the *queue* or if it's already been tested. If they haven't, and, if they're not dominated by any solution on the final list of solutions or, are incomparable with all of them, they'll be placed into the *queue* for later testing.

5.3.4.1. Admissibility Test

The admissibility test has a critical role on the aggregation algorithm. Like in the previous version of the prototype, it's in charge of verifying if a certain aggregated solution fulfills the global requirements of the *CSA*. This *admissibility testing* of the solution is performed by the *AggChecker* class.

The previous version of the prototype only considered global requirements over the price of the solution. The decision maker could define a *pricing* requirement stating the *maximum* or *minimum* cost of the aggregated solution (the cost of an aggregated solution is equal to the sum of the cost of its alternatives) and, every alternative that failed to meet this requirement would fail the test.

On CloudAid2, this *admissibility test* has been expanded, now supporting three types of global requirements:

- Qualitative Requirement
- Quantitative Requirement
- Price Requirement

Regarding the price requirements, their behavior is the same as in the CloudAid1 prototype: should the aggregated solution's cost exceed a *maximum* or cost less than a *minimum*, it'll fails the test. Quantitative requirements follow the same ideology as the *price* requirements. E.g.: The decision maker creates a global *MemorySize* (RAM) requirement stating that the aggregated solution he's looking for should possess an overall *MemorySize* of 10 GB or higher. Every aggregated solution solution that possesses an overall *MemorySize* size lower than 10 GB will be discarded.

Qualitative requirements however, follow a different approach: in case the decision maker defines a global qualitative requirement (E.g.: *Location*), he'll be able to define the minimum number of alternatives (in the aggregated solution) that need to have/contain the value he's looking for. For example, consider that the decision maker create a *CSA* with 4 different *Service Templates* and he needs at least two of the alternatives in the aggregated solution to be located in *Tokyo*. To discard solutions with one or less alternatives located in Tokyo, he/she would need to define the minimum number of alternatives that need to be located in Tokyo (two), when creating the global *Requirement* in the *CSA*.

5.3.5. Graphical User Interface

CloudAid1 had no graphic interface available, the interaction between the decision maker and the prototype was carried out by a simple shell application which lowered its usability and concluded in an unpleasant experience. For the new prototype, a friendly JavaFX[53] Graphical User Interface was created, easing the interaction with the decision maker and strengthening the usability of the prototype.

Initially, the proposed objective was to develop a graphical interface using HTML5[54] however, due to time constrains, I decided to switch to another technology that had been considered but put aside in favor of HTML5: JavaFX. As the name implies, its Java based, posing a much softer learning curve for me and allowing a faster development.

Choosing JavaFX allowed me to focus right away on the functionalities and possibilities of the technology rather than spending extra time learning the basics and how its components interact with each other.

As mentioned on section 5.3.1.1, the GUI and the server communicate via JSON files. Every time they need to send or request information, they describe the data in JSON format and write it into a file on pre-defined directory. Since the GUI is also Java based, its JSON processing is equivalent to the JSON processing of the server side. It recurs to the same data models and to the *Gson* library to load the information described in the JSON files into their corresponding Java models and ease their manipulation.

Regarding the development of the GUI itself, I won't go into much detail for there's little value in its explanation however, it was an interesting challenge and a good experience from a technical perspective, providing me with a good *hands-on* on this technology.

Its development was greatly enhanced by the use of JavaFX Scene Builder[55]. Scene Builder is a visual layout tool that allows a quick design of JavaFX application user interfaces without the need to code. We can drag & drop UI components, edit their properties, and the FXML[56] code for the layout will be automatically generated. Afterwards, this FXML description of the layout is bound to the application's logic, more precisely, to their *Controller* class. This *Controller* class has access to the FXML (UI) components, being capable of editing their properties, fetch their content (*TextField*[57] is a common UI component) or even erase them programmatically.

[53] http://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm
[54] http://www.w3schools.com/html/html5_intro.asp
[55] http://www.oracle.com/technetwork/java/javase/downloads/javafxscenebuilder-info-2157684.html
[56] http://docs.oracle.com/javafx/2/api/javafx/fxml/doc-files/introduction_to_fxml.html
[57] http://docs.oracle.com/javafx/2/api/javafx/scene/control/TextField.html

5.3.5.1. Use Cases

This section presents the possible interactions between the user and the Graphic User Interface (GUI). Fig.31 presents every action the user is capable of doing using the prototype's GUI.

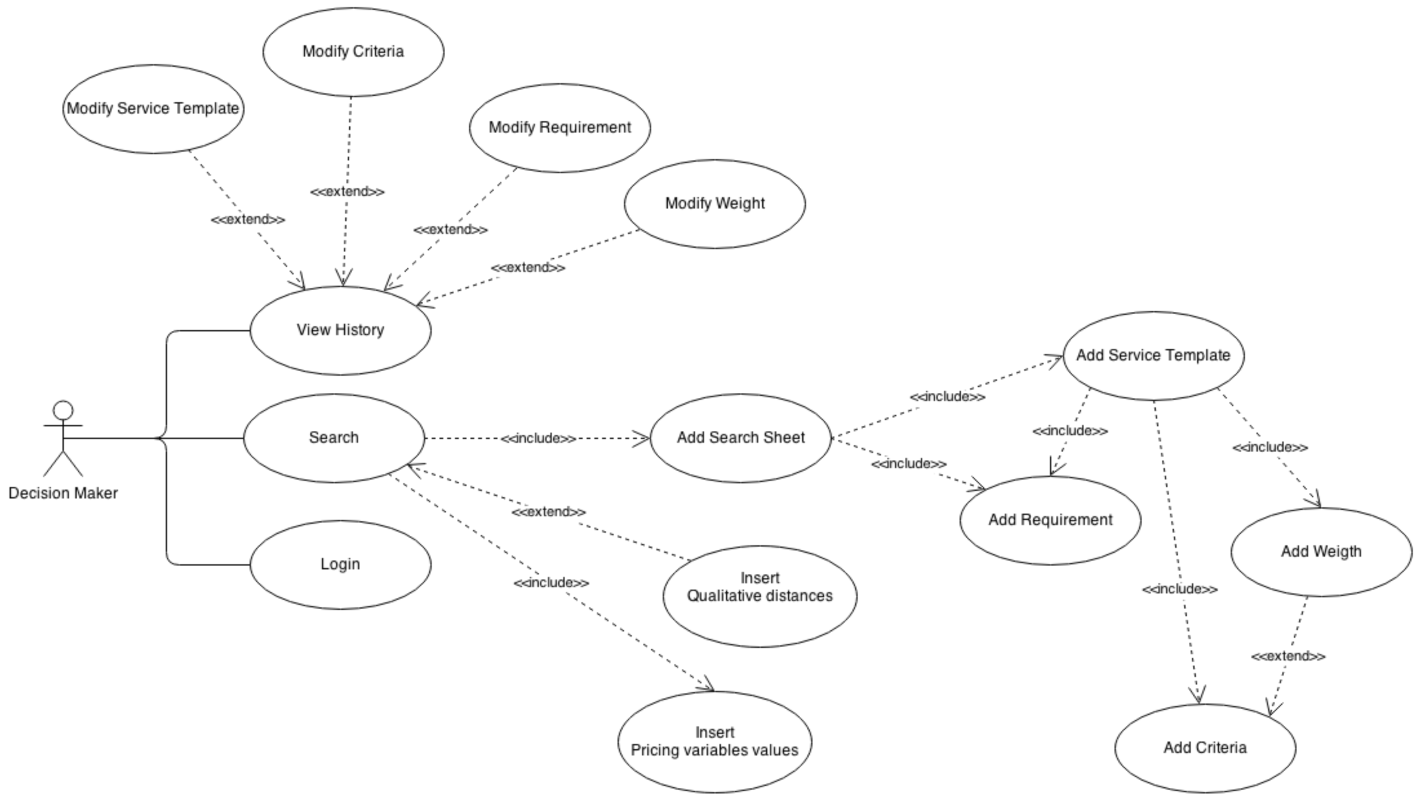


Fig. 31 - GUI's use cases

Next, each of these actions is described, explaining what their purpose is and what is expected from the decision maker:

- **View History** – The user is able to load 'Search Sheets' that he created in the past, modify and re-use them.

- **Search** – The user may send *Search Sheet* (*CSA*) requests the server:

  - **Add Search Sheet** – The user can create his Search Sheets and define the desired characteristics for the system he's looking for. As you can see, the action of inserting a new Search Sheet requires a certain number of interactions from the user. Some are mandatory, others are not.

  - **Add Service Template** – The user should create a new Service Template for each different component of his architecture. For example, let's say he wants to build a system composed of 3 different services: a server, a load balancer and a database. In this case the user needs 3 different Service Templates. Every Service Template has its own set of requirements and criteria that need to be defined by the user through the actions: **Add Requirement** and **Add Criteria**.

- **Add Weight** – The user can define the importance of each criterion he creates but, it's not mandatory since the prototype supports decision methods that don't require that kind of information.

- **Insert Pricing Variables values** – The user needs to insert the values of the pricing variables related to the Service Template. Once he/she sends his/hers *Search Sheet (CSA)* to the server, the latter will request some mandatory values for the pricing variables of the alternatives found in order to calculate their cost.

- **Insert Qualitative distance values** – After evaluating his Search Sheet, if he defined any qualitative criteria on it, a second interaction with the decision maker will be required. The GUI will receive a list of concepts from the server and will present the user a new *popup* where he needs to define the numerical distances between his preferred value and the values that the alternatives have. E.g.:

    - The user created a criterion for the *Performance* feature and set his ideal value to *Excellent.* Assuming that the prototype only found alternatives for the *Service Template* with a *Performance = High* and a *Performance = Low.* The GUI would request from the decision maker, the numerical distances between: *Excellent – High* and *Excellent – Low.*

5.3.5.2. Resulting GUI and Mockups comparison

In this section, I present some screen shots of the final GUI, comparing it with the initial specification of the interface. A video that exemplifies how to work with the GUI and its functions is available on [52]. While the final result is very similar to the initial specification, some changes occurred either due to some change on the logic of the prototype, simplicity or simply due to the shortage of time. The specification mockups were created with the help of the *balsamiq mockups* software. It's an easy to use and effective tool to build rich and understandable mockups [32].

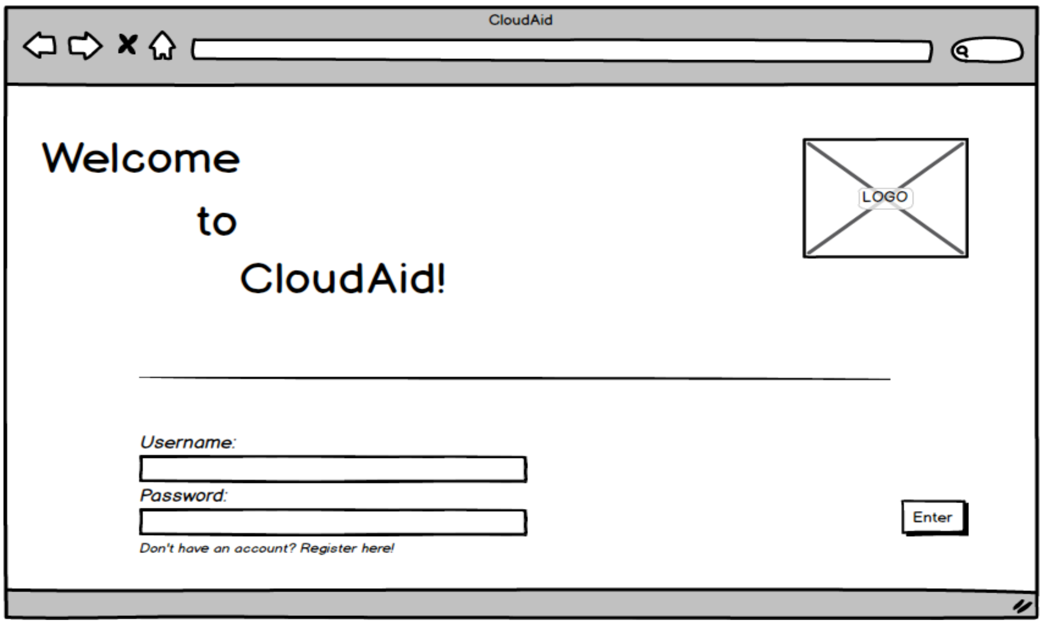


Fig. 32 - Login/Register panel mockup

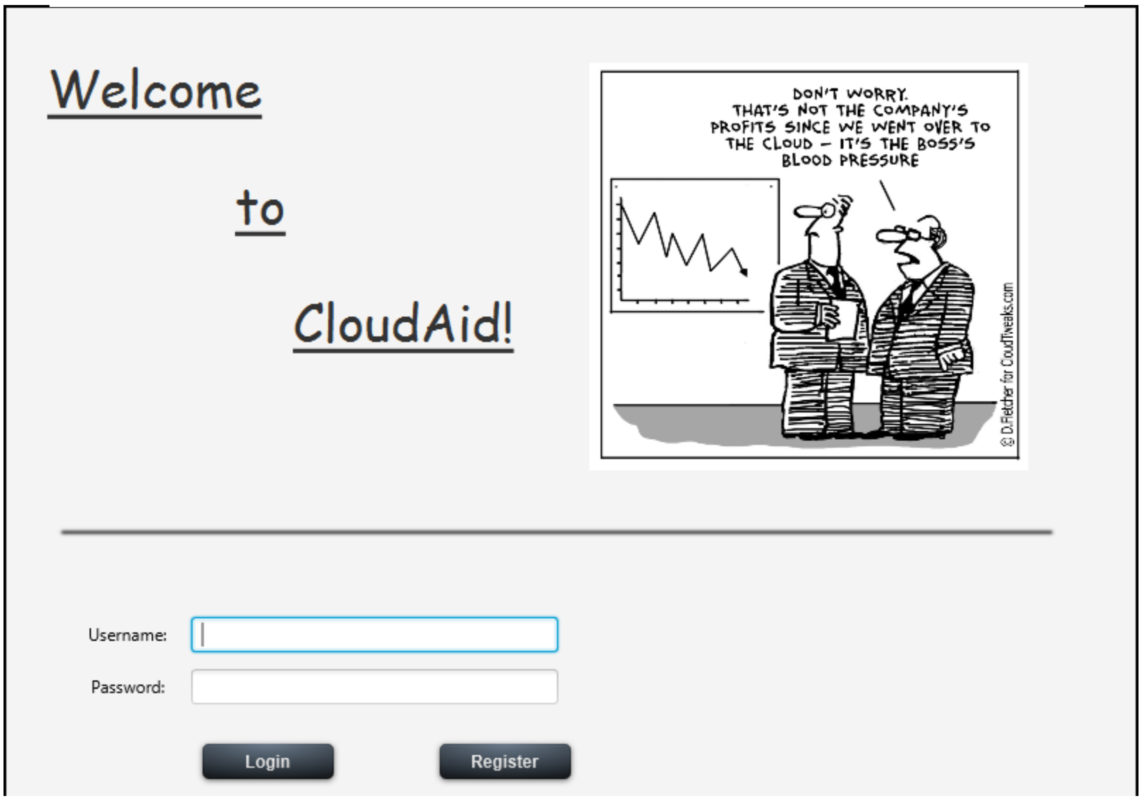

Welcome
to
CloudAid!
DON'T WORRY. THAT'S NOT THE COMPANY'S PROFITS SINCE WE WENT OVER TO THE CLOUD – IT'S THE BOSS'S BLOOD PRESSURE
© D.Fletcher for CloudTweaks.com
Username:
Password:
Login
Register

Fig. 33 - Login/Register panel GUI

Fig.32 presents the originally proposed login panel and Fig.33 presents the current log in panel of the interface. As you can see, they're very similar, differing only on minor aspects. The same applies for the Main Panel however, the GUI's main panel presents more information like the *checkbox* to create a global requirement or the *combobox* to select the *MCDM*.

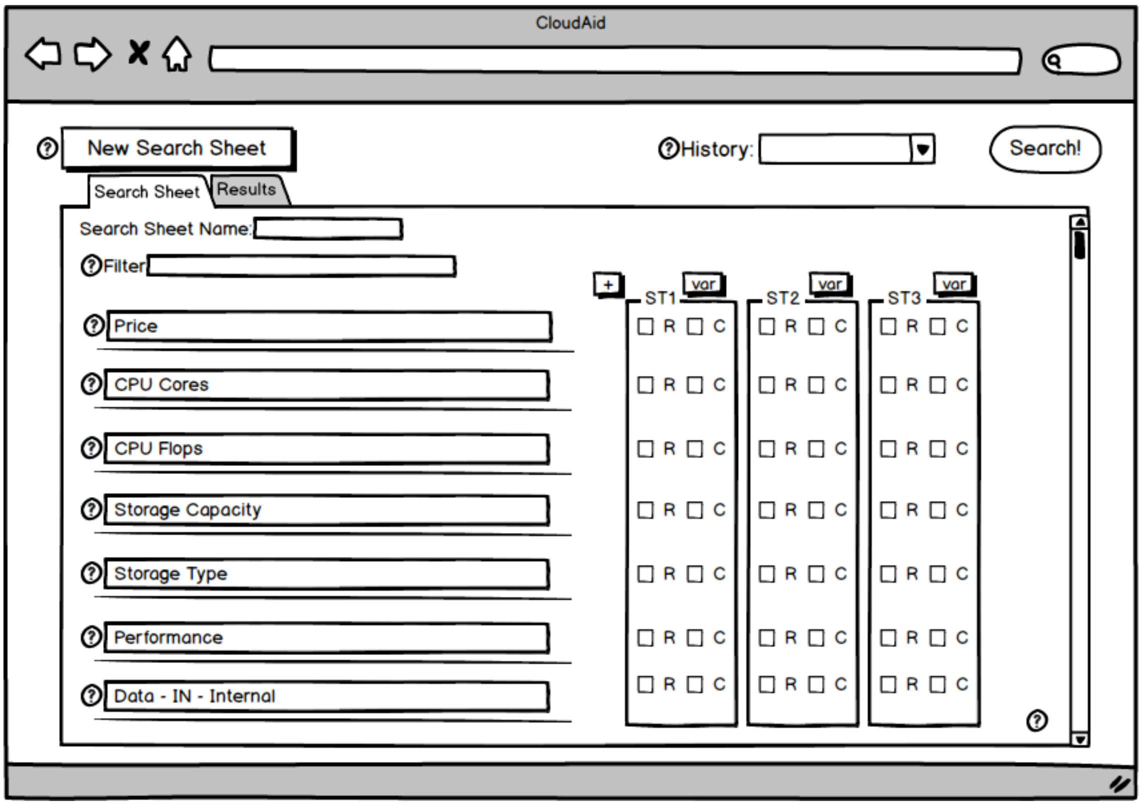


Fig. 34 - Main Panel mockup

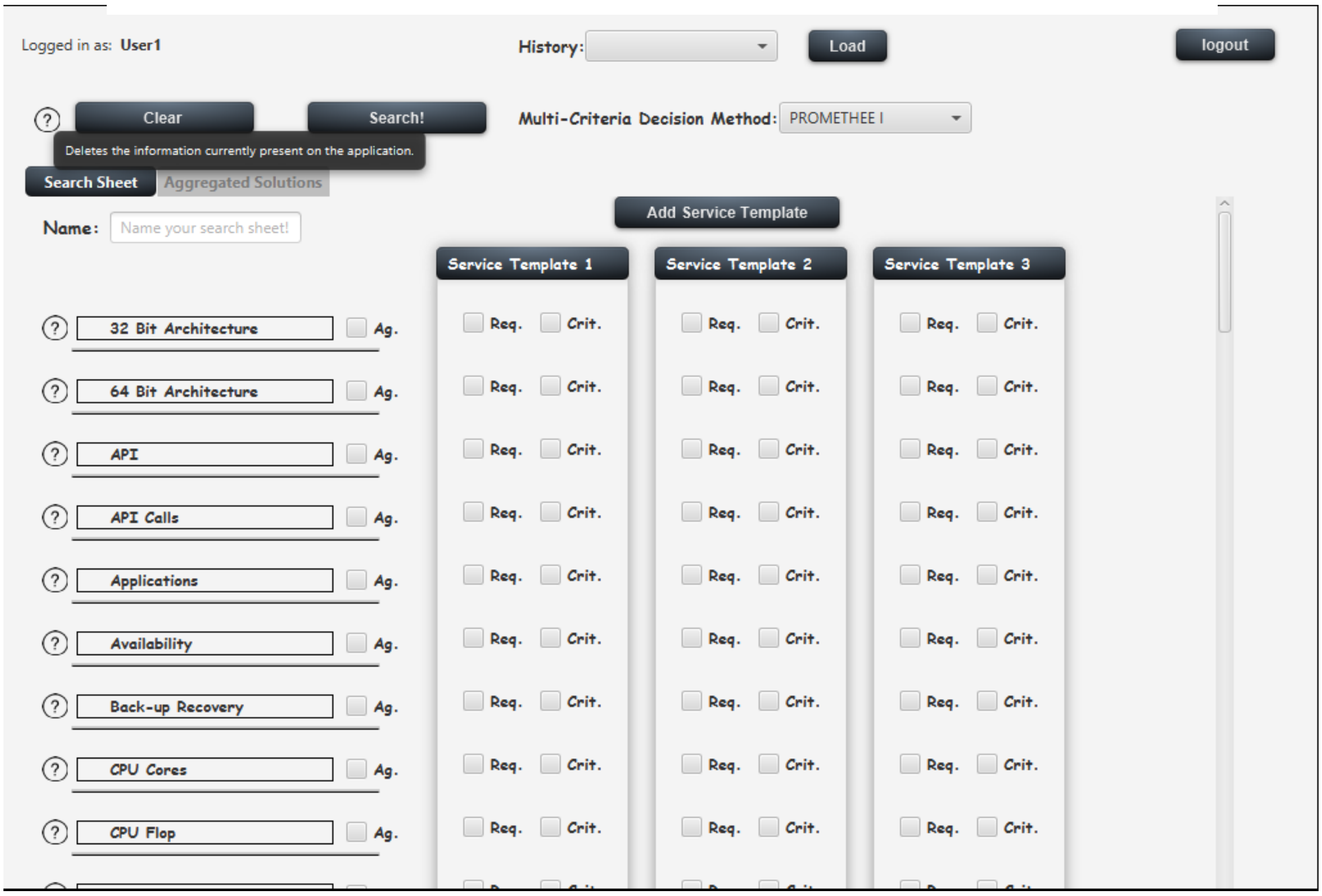
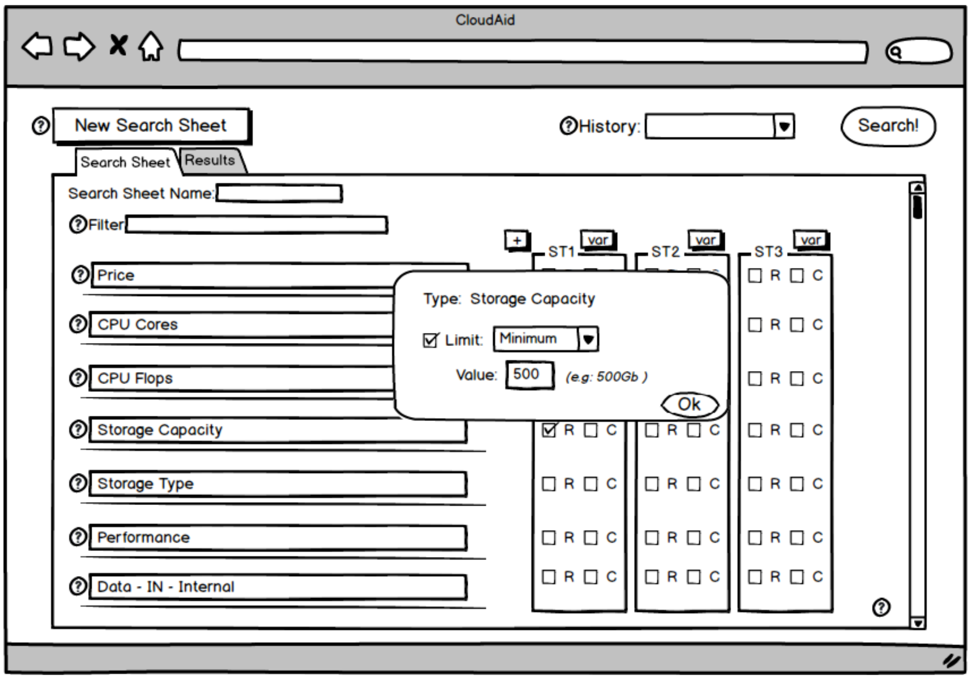


Fig. 35 - Main Panel GUI

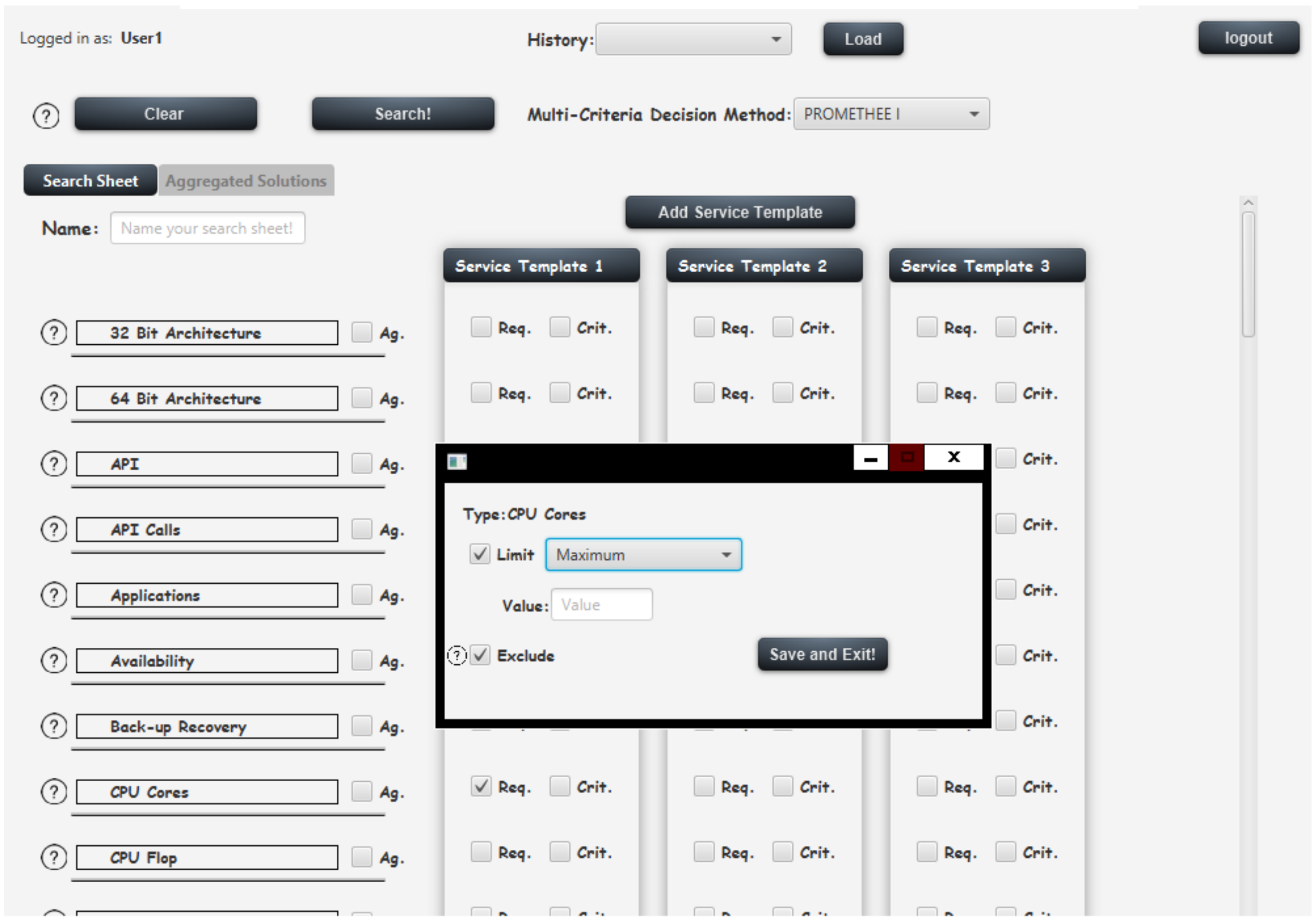


Type: Storage Capacity

Limit: Minimum

Value: 500 (e.g. 500Gb)

Fig. 36 - Adding a quantitative requirement mockup



Multi-Criteria Decision Method: PROMETHEE I

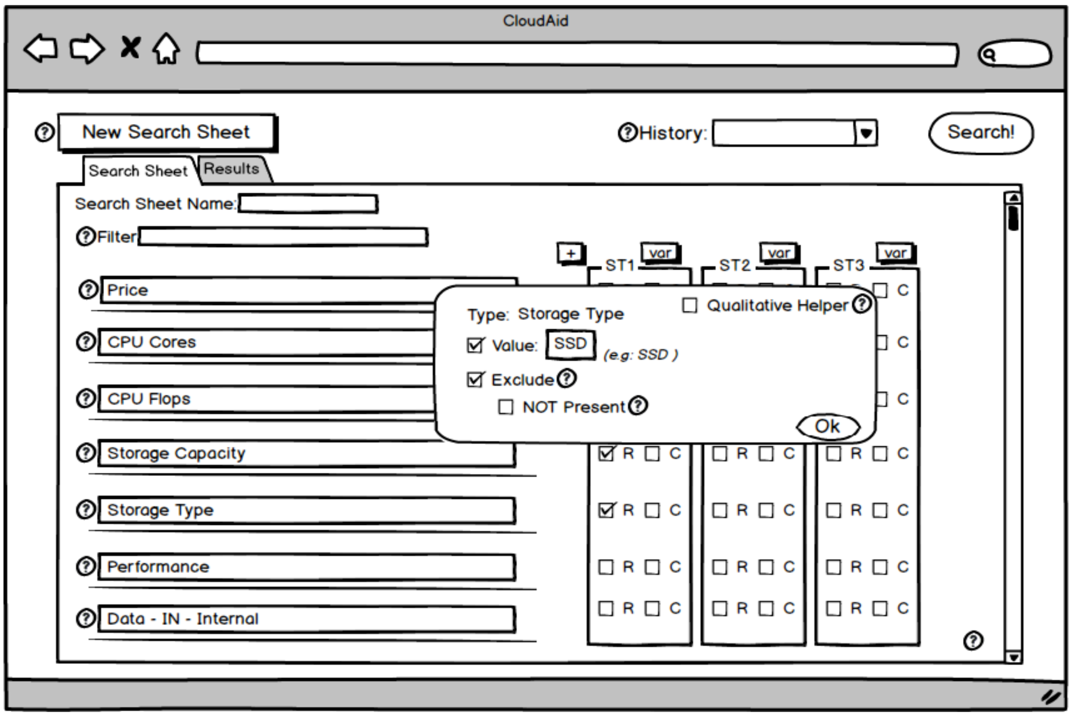
Type: CPU Cores

Limit: Maximum

Value: Value

Exclude

Fig. 37 - Adding a quantitative requirement GUI



Type: Storage Type

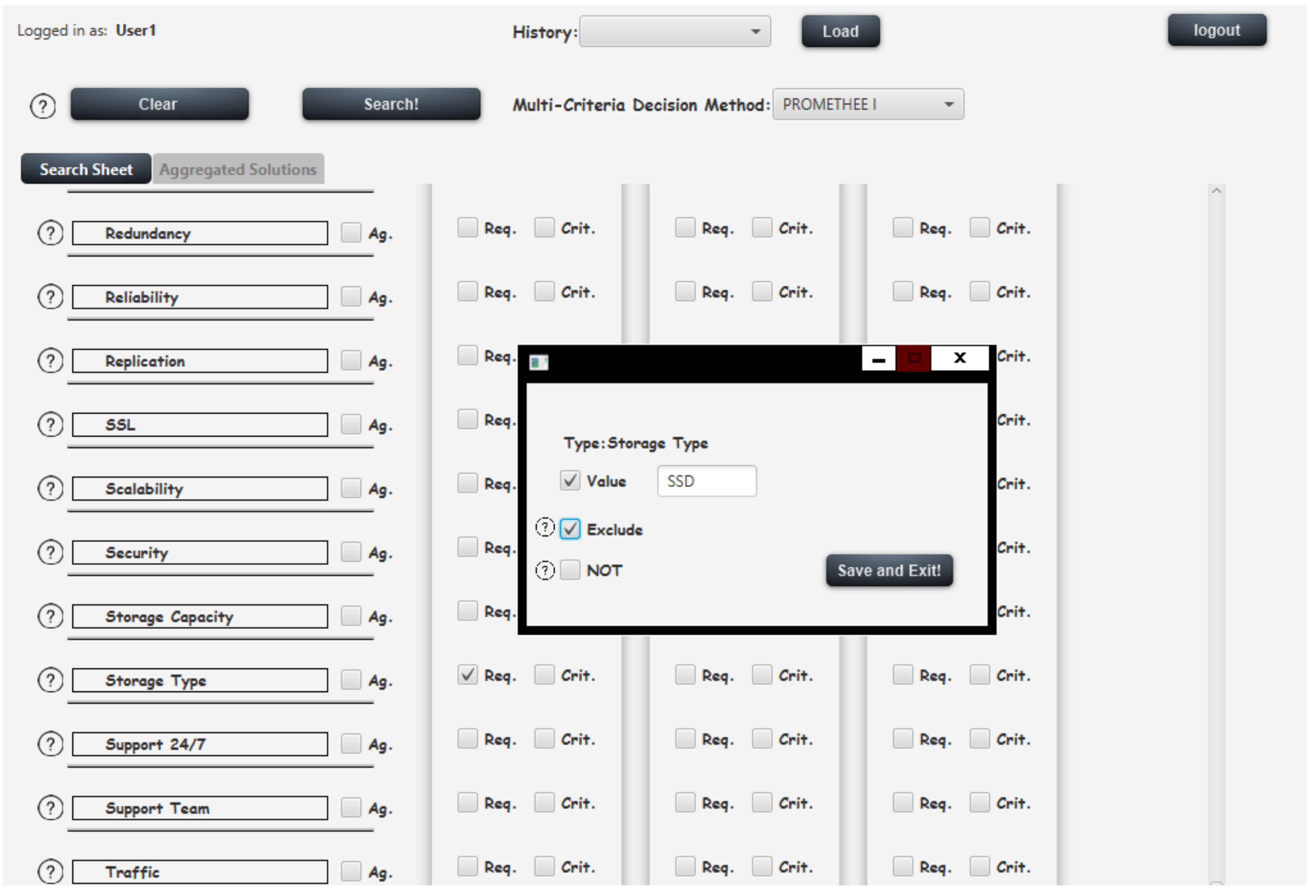
☐ Qualitative Helper

☑ Value: SSD (e.g. SSD)

☑ Exclude

☐ NOT Present

Fig. 38 - Adding a qualitative requirement mockup



Multi-Criteria Decision Method: PROMETHEE I

Type:Storage Type

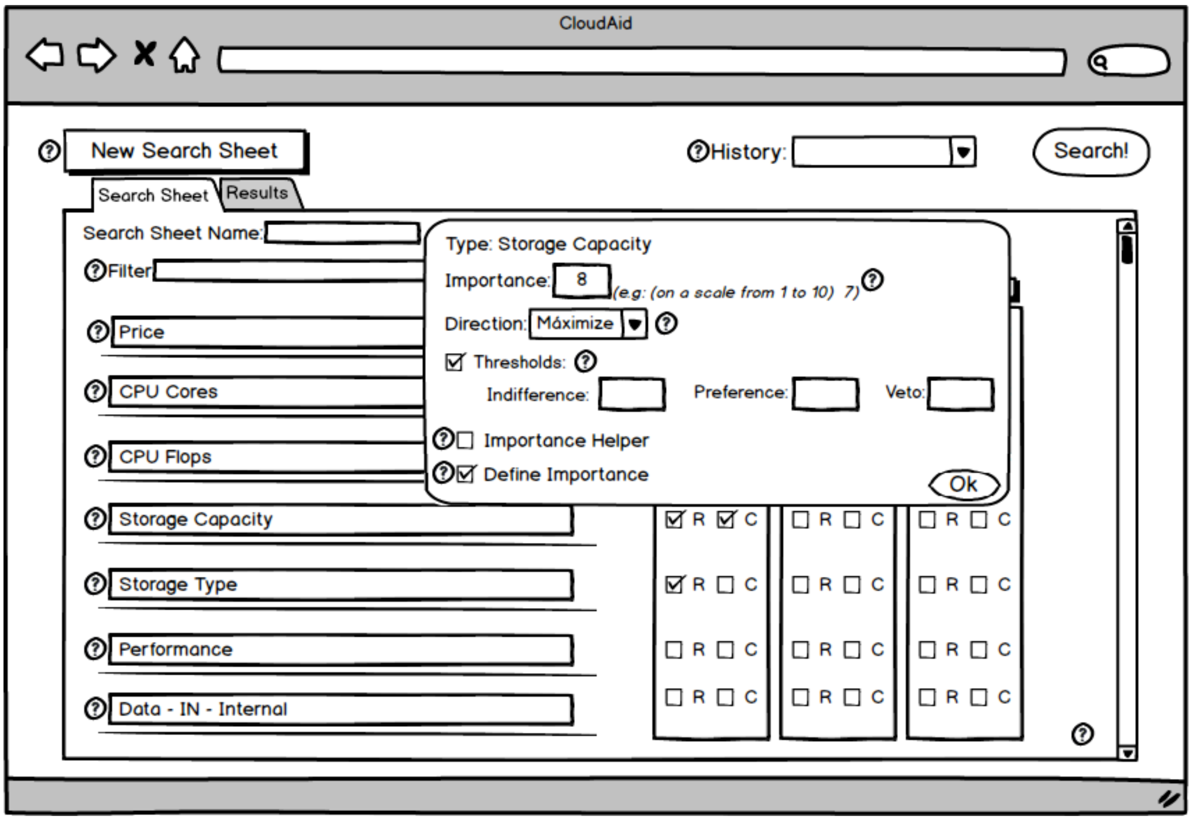
☑ Value: SSD

☑ Exclude

☐ NOT

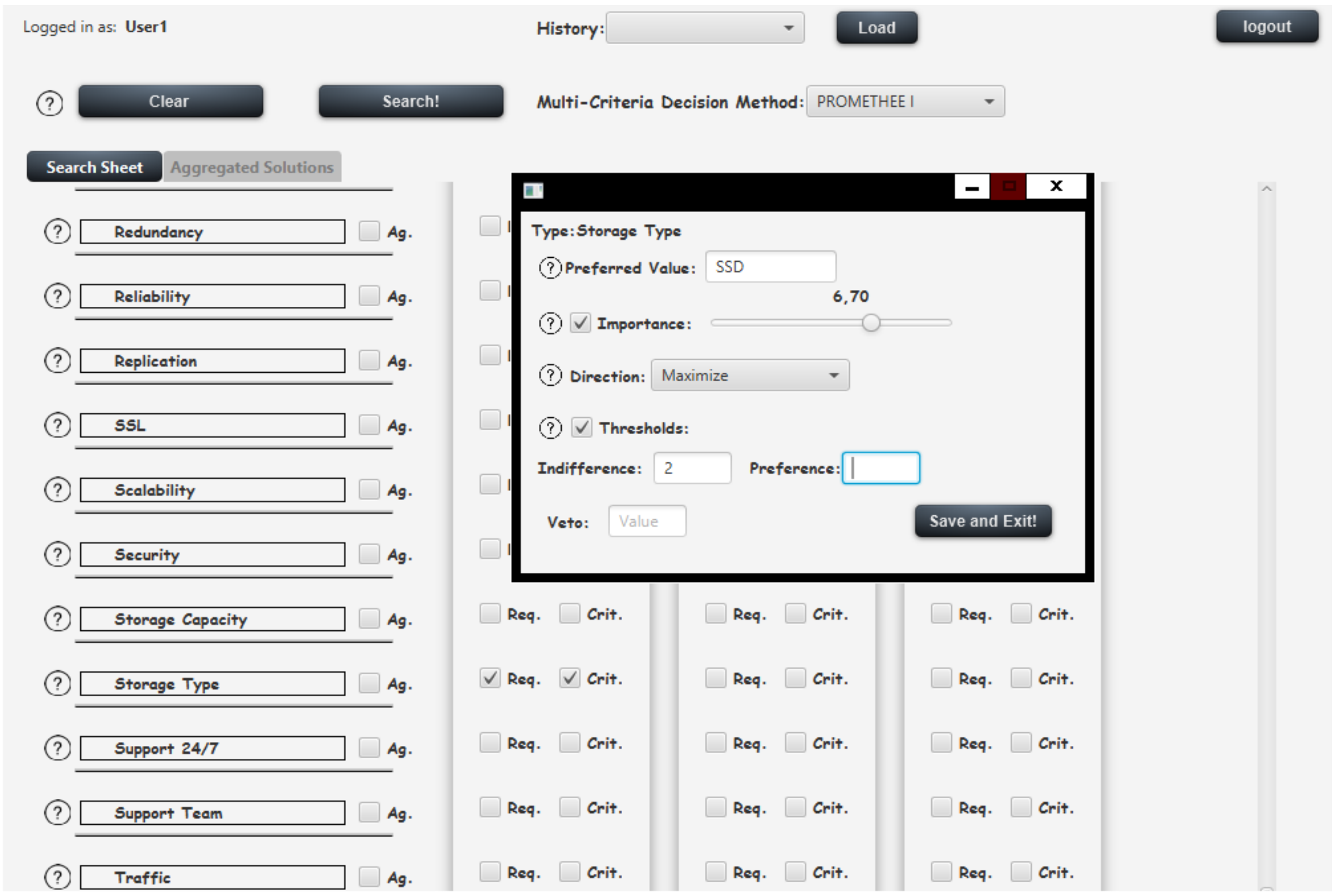
Fig. 39 - Adding a qualitative requirement GUI

The main difference between fig.38 and fig.39 is the removal of the *Qualitative Helper* functionality.



Type: Storage Capacity
Importance: 8 (e.g: (on a scale from 1 to 10) 7)
Direction: Máximize
☑ Thresholds:
Indifference:
Preference:
Veto:
☐ Importance Helper
☑ Define Importance

Fig. 40 - Adding a criterion mockup



Multi-Criteria Decision Method: PROMETHEE I
Type: Storage Type
Preferred Value: SSD
Importance: 6,70
Direction: Maximize
Thresholds:
Indifference: 2
Preference:
Veto: Value

Fig. 41 - Adding a criterion GUI

Adding a criterion has a few changes as well. The Importance Helper functionality was replaced by a *slider* (inspired by the *AHP* method) which ranges from 0 to 10. The slider provides visual support to the decision maker, making the need for a secondary method of support to define the importance unnecessary.

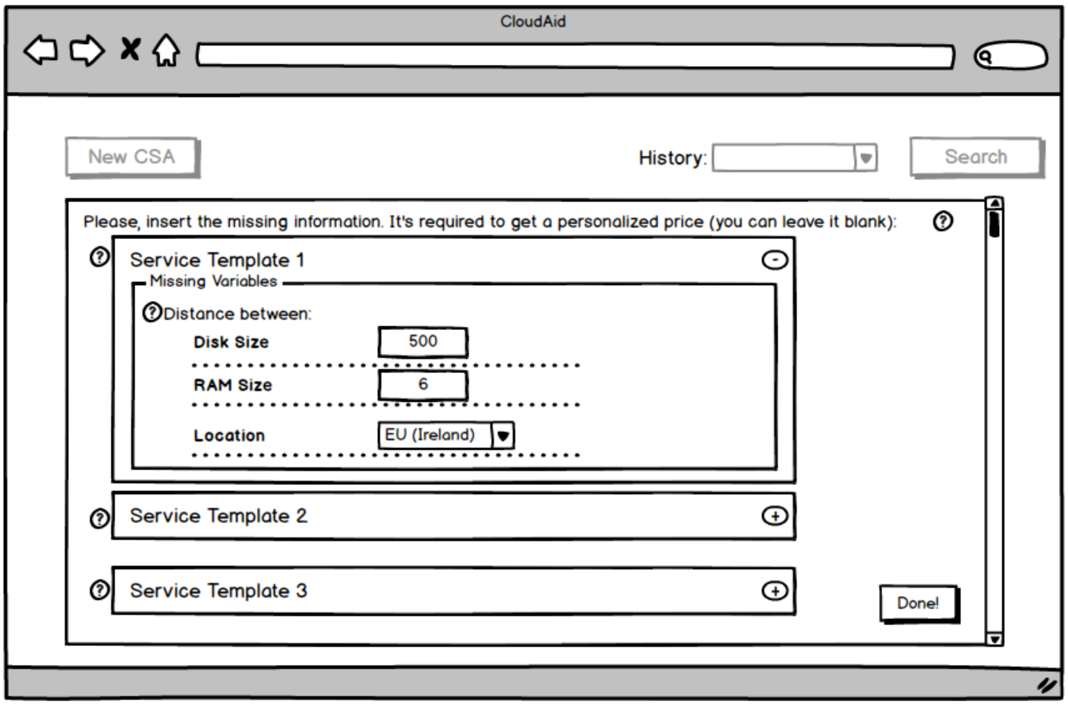
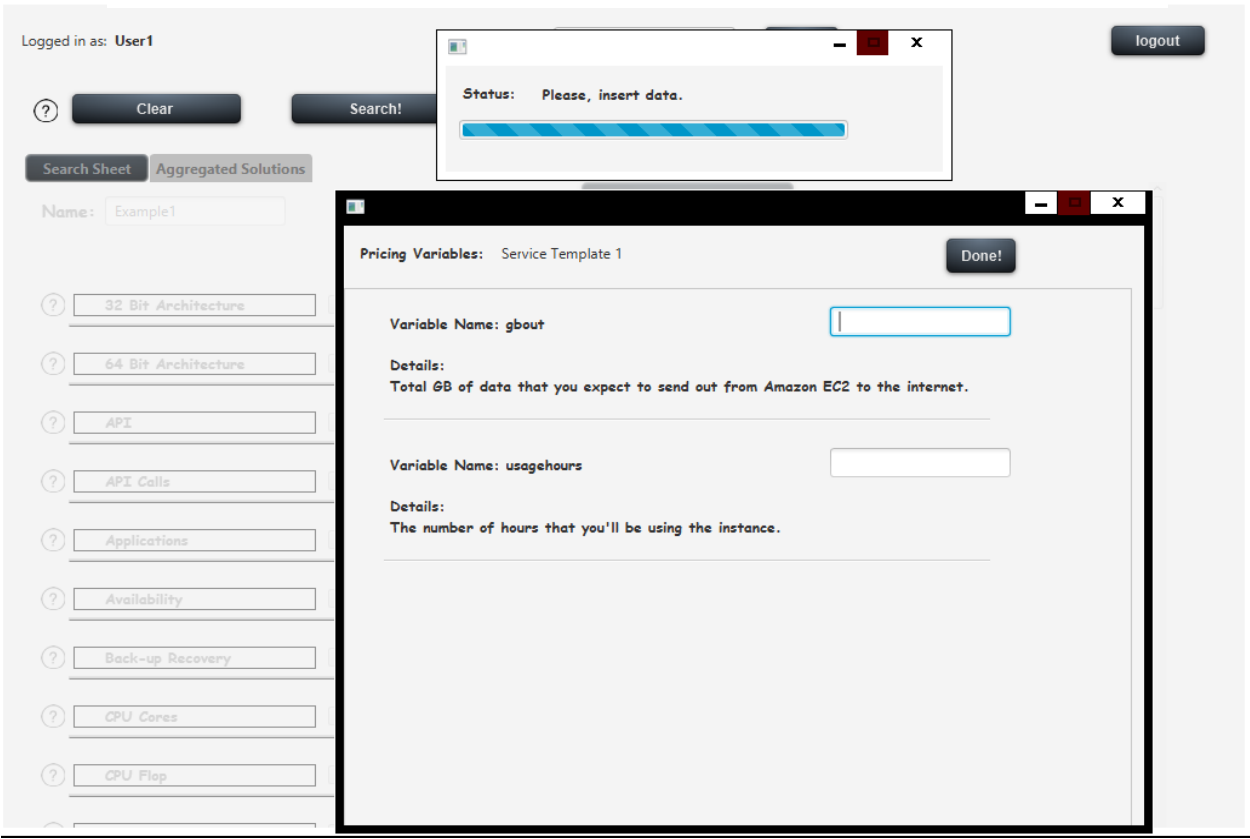


Fig. 42 - Defining pricing variables values mockup



Pricing Variables: Service Template 1

Variable Name: gbout

Details:
Total GB of data that you expect to send out from Amazon EC2 to the internet.

Variable Name: usagehours

Details:
The number of hours that you'll be using the instance.

Fig. 43 - Defining pricing variables values GUI

The main difference between the specification and the actual GUI (Fig.42 and Fig.43) is the moment in which the values for the pricing are requested. The initial proposal was to define some values prior to sending the request to the server however, for simplicity sake, I decided to remove that option and request the values for the pricing variables only once per *Service Template*, after the request is sent to the server.

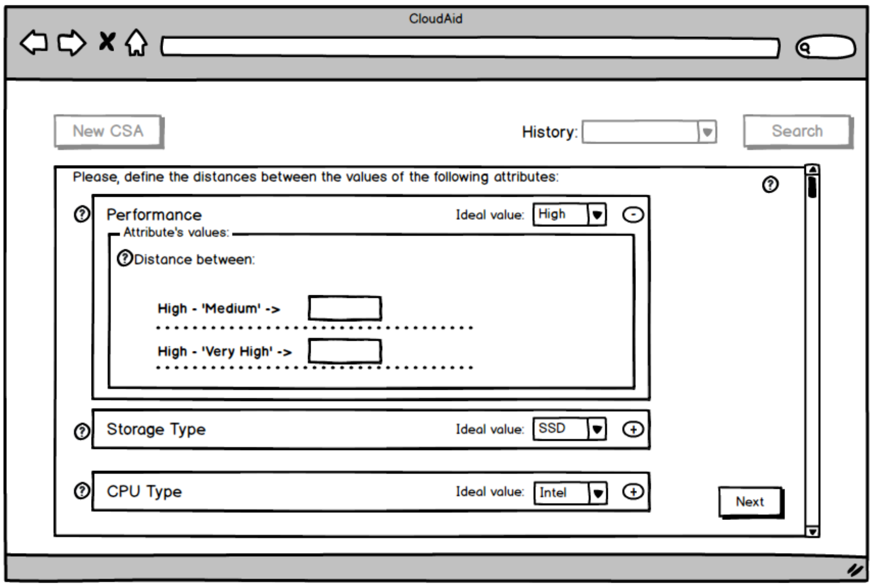


Fig. 44 - Define qualitative criterion's values distances mockup

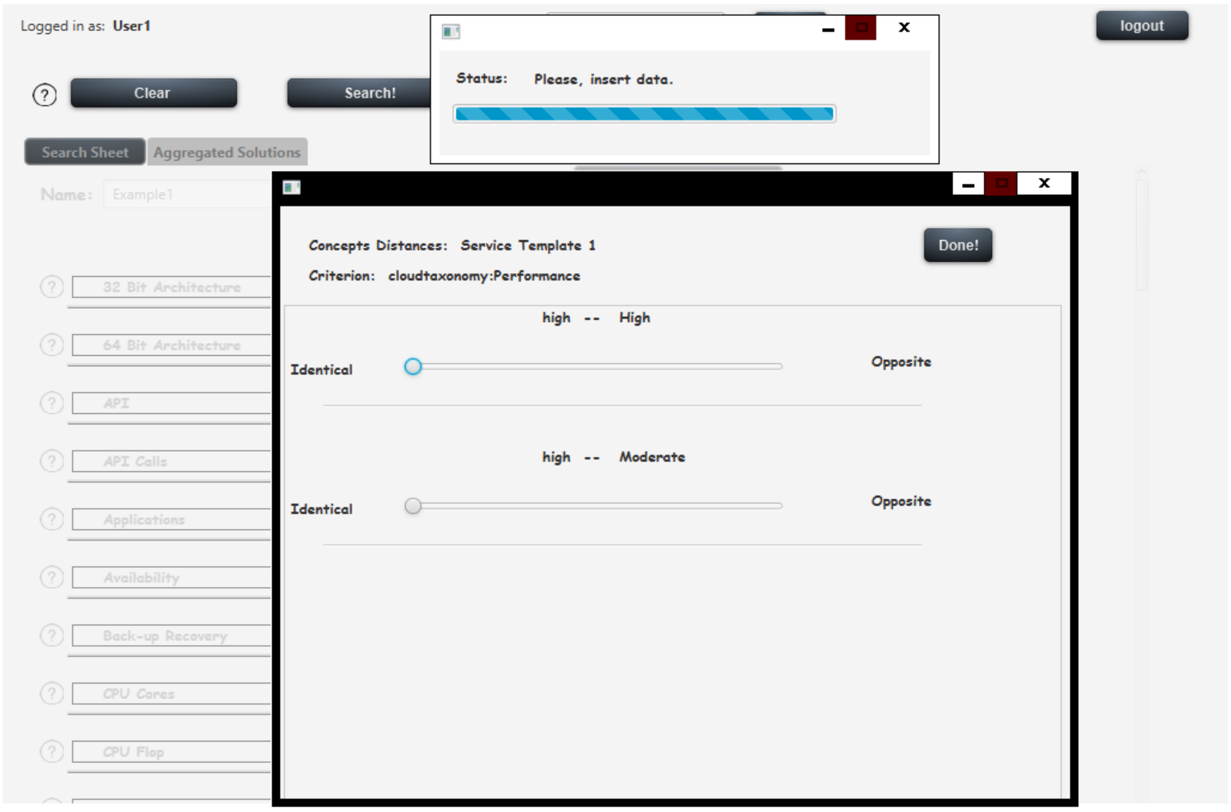


Fig. 45 - Define qualitative criterion's values distances GUI

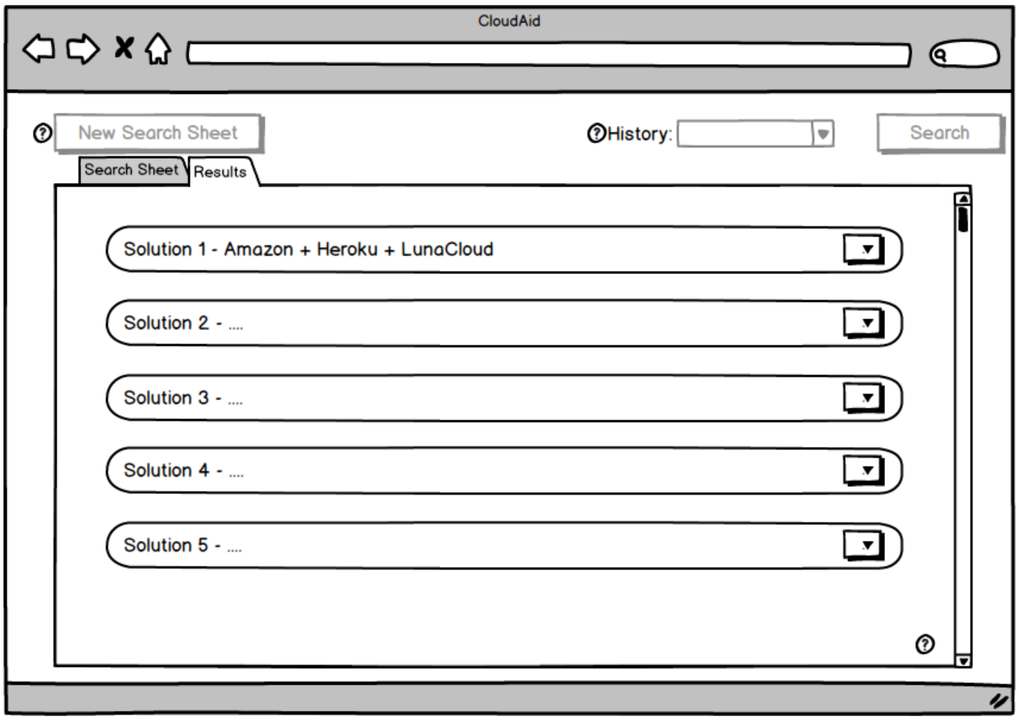
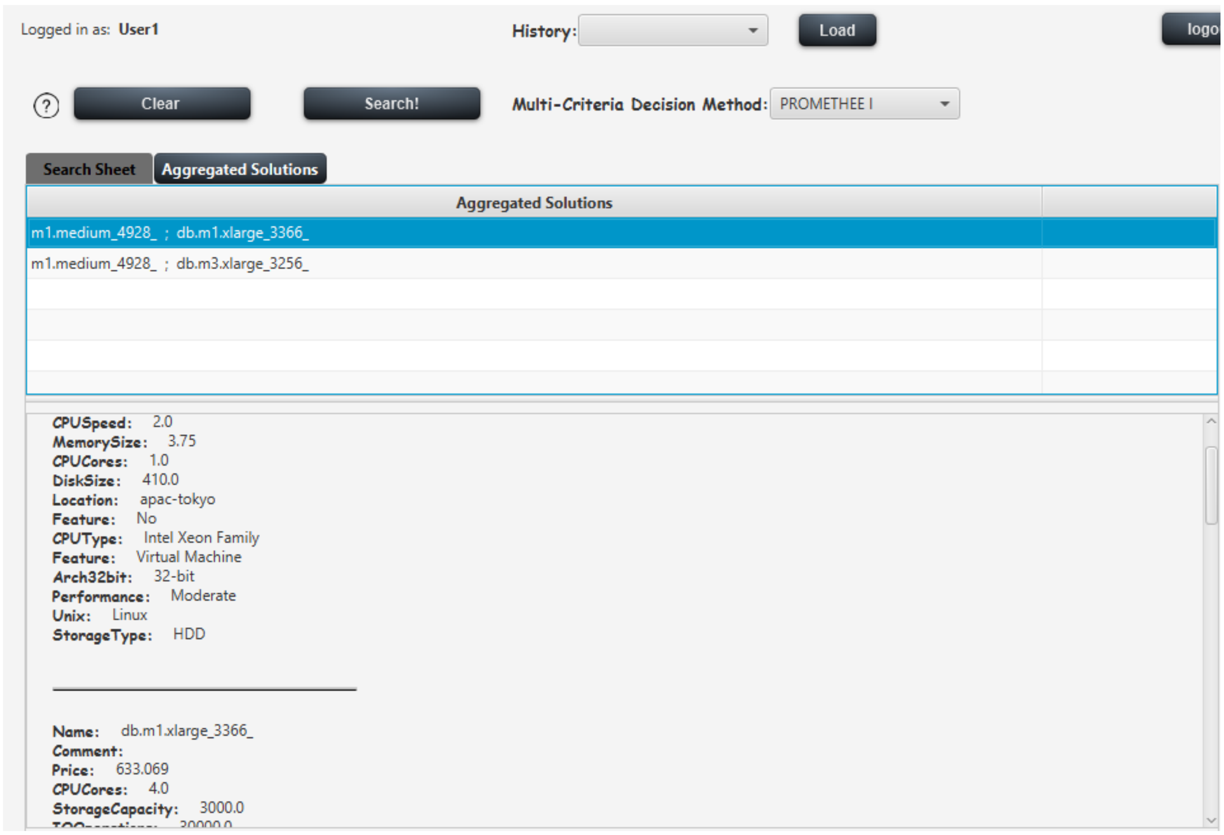


Fig. 46 - Results panel mockup



Multi-Criteria Decision Method: PROMETHEE I

Aggregated Solutions
m1.medium_4928_ ; db.m1.xlarge_3366_
m1.medium_4928_ ; db.m3.xlarge_3256_

CPUSpeed: 2.0
MemorySize: 3.75
CPUCores: 1.0
DiskSize: 410.0
Location: apac-tokyo
Feature: No
CPUType: Intel Xeon Family
Feature: Virtual Machine
Arch32bit: 32-bit
Performance: Moderate
Unix: Linux
StorageType: HDD

Name: db.m1.xlarge_3366_
Comment:
Price: 633.069
CPUCores: 4.0
StorageCapacity: 3000.0

Fig. 47 - Results panel GUI

The layout for the results tab is different from the one initially proposed. Now, it's very similar to the layout used by a typical e-mail platform. Each aggregated solution is presented on a clickable table which, when clicked, presents the detailed information about each alternative of the clicked aggregated solution, on the *panelview* below.

6. Testing

This chapter presents and validates the results obtained by the CloudAid2 prototype. Due to time constrains, not all of the tests presented on Table.13 were formally described however, most of them were merged with other tests instead of being individually tested allowing to still prove their correct implementation and functionality. Each functionality was target of informal individual preliminary tests (Unit testing) alongside the development phase.

It starts by presenting the list of functional requirements implementation status, and the functional tests performed, on section 6.1. Section 6.2 presents the reliability testing performed to the *Search Module* followed by section 6.3 which presents the *integration* testing with the external *web services* and their results validation. Finally, on section 6.4, I present the *reliability* and *performance* testing of the aggregation algorithm.

6.1. Functional testing

Functional testing aims at verifying that the prototype works as intended and that the proposed functionalities were correctly implemented [33, 34]. Table.13 presents the functional requirements extracted from **Appendix A** and their corresponding status. There's two possible status for a functional requirement:

1. **DONE** – The requirement was implemented and tested
2. **NOT DONE** – The requirement wasn't implemented

As you can see, every "MUST" requirement was implemented. This allowed to have a fully operational prototype and more importantly, allowed to meet every requirement proposed at the beginning of the project. Regarding the NOT DONE requirements, these derived from the self-proposal of implementing a *web* communication between the GUI and the Server however, due to some shortage of time, I wasn't able to comply with them. While they were not implemented, they do not affect the performance and objective of the prototype at all.

ID	**Name**	**Priority**	**Status**
	Functional		
Search Module			
RF1	Dynamic price calculation	MUST	**DONE**
RF2	Read and handle Linked USDL Cloud Services correctly	MUST	**DONE**
Service Set – Cloud Gen (Service Gatherer)			
RF1	Use Scrape techniques to fetch and create our Linked USDL service descriptions	MUST	**DONE**
RF2	Create files with the RDF service description	MUST	**DONE**

Decision Module			
RF1	Decision Method selection	MUST	**DONE**
RF2	New Multi-Criteria Decision Methods	MUST	**DONE**
RF3	Group similar alternatives	SHOULD	**NOT DONE**
RF4	Thresholds definition	MUST	**DONE**
RF5	Correct XMCDA description	MUST	**DONE**
RF6	SOAP Message encapsulation	MUST	**DONE**
RF7	Multi-Criteria Decision Methods execution	MUST	**DONE**
Aggregation Module			
RF1	Admissibility test expansion	MUST	**DONE**
RF2	Compatibility with the new results from the multi-decision criteria methods	MUST	**DONE**
Graphic User Interface (GUI) - Client			
RF1	Add a new Service Template	MUST	**DONE**
RF1.1	Add a requirement to a Service Template	MUST	**DONE**
RF1.2	Add a criterion to the Service Template	MUST	**DONE**
RF2	Add a global requirement	MUST	**DONE**
RF3	Send data to server using REST protocol	SHOULD	**NOT DONE**
RF4	Receive and present results from the server	MUST	**DONE**
RF5	Re-use *Search Sheets* from the History	MUST	**DONE**
RF6	Insertion of the pricing variables	MUST	**DONE**
RF7	Register	SHOULD	**NOT DONE**
RF8	Login	SHOULD	**NOT DONE**
Server			
RF1	Wait for data from the graphic interface	MUST	**DONE**
RF2	Send the results to the client interface	MUST	**DONE**
RF3	Send results to e-mail	SHOULD	**NOT DONE**
RF4	Keep past search sheets sent by users and their respective result	MUST	**DONE**
RF5	Access control	SHOULD	**NOT DONE**

Table 13 - Functional Requirements List

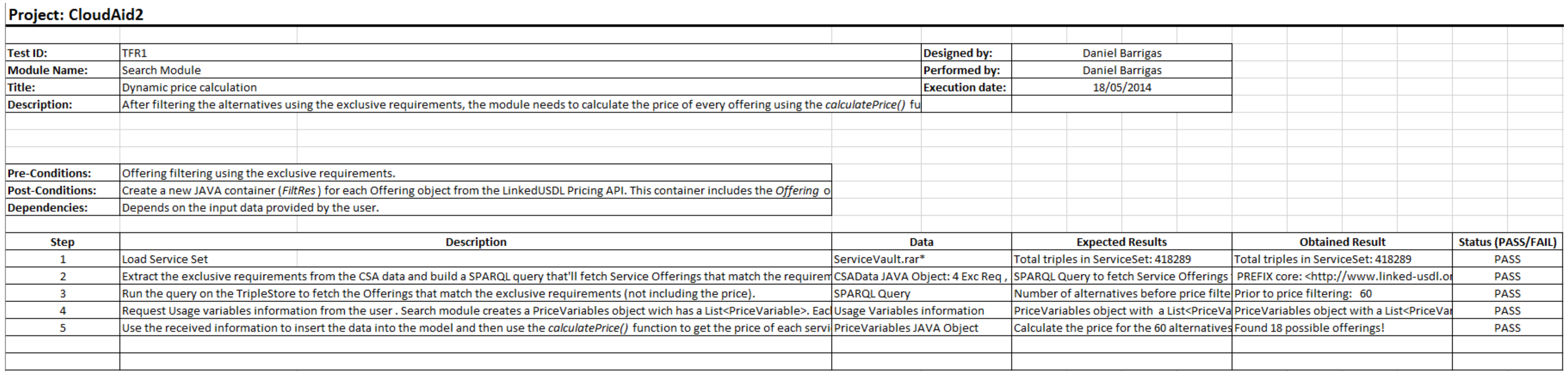
To assess the correct implementation of these requirements, a series of tests were performed. These tests also served as a mean to identify and correct several problems of the prototype, whether they were related to implementation problems, or even conceptual problems leading to some changes on the prototype's logic.

Fig.48 presents an example of a formal description for the RF1 requirement where the capability for dynamic price calculation is being tested. The test was successfully executed and the prototype successfully passed on every step.

RF1 is also a good example of the merge testing approach followed, where multiple functional requirements can be tested on a single test case. It tests not only RF1, but also RF6 (GUI), RF4 (GUI) and RF2 (Search Module).

In fact, the prototype (which can be downloaded from [52]) is accompanied with two *CSA* examples ready to use. The user just needs to "load" the *CSA* from the *History* and send the request to the server. Doing this will test every functional requirement depicted on table.13, from GUI's functionalities to the last step of the server: returning the aggregated solutions back to the GUI.

Note that a video presenting the usage of the prototype and its functionalities can be found on [52].



Project: CloudAid2

Test ID:	TFR1	**Designed by:**	Daniel Barrigas
Module Name:	Search Module	**Performed by:**	Daniel Barrigas
Title:	Dynamic price calculation	**Execution date:**	18/05/2014
Description:	After filtering the alternatives using the exclusive requirements, the module needs to calculate the price of every offering using the *calculatePrice()* fu		

Pre-Conditions:	Offering filtering using the exclusive requirements.
Post-Conditions:	Create a new JAVA container (*FiltRes*) for each Offering object from the LinkedUSDL Pricing API. This container includes the *Offering* o
Dependencies:	Depends on the input data provided by the user.

Step	Description	Data	Expected Results	Obtained Result	Status (PASS/FAIL)
1	Load Service Set	ServiceVault.rar*	Total triples in ServiceSet: 418289	Total triples in ServiceSet: 418289	PASS
2	Extract the exclusive requirements from the CSA data and build a SPARQL query that'll fetch Service Offerings that match the requirem	CSAData JAVA Object: 4 Exc Req ,	SPARQL Query to fetch Service Offerings	PREFIX core: <http://www.linked-usdl.o	PASS
3	Run the query on the TripleStore to fetch the Offerings that match the exclusive requirements (not including the price).	SPARQL Query	Number of alternatives before price filte	Prior to price filtering: 60	PASS
4	Request Usage variables information from the user . Search module creates a PriceVariables object wich has a List<PriceVariable>. Eac	Usage Variables information	PriceVariables object with a List<PriceVa	PriceVariables object with a List<PriceVa	PASS
5	Use the received information to insert the data into the model and then use the *calculatePrice()* function to get the price of each servi	PriceVariables JAVA Object	Calculate the price for the 60 alternatives	Found 18 possible offerings!	PASS

Fig. 48 - Test Case RF1

6.2. Search Module testing

The testing of the *Search Module* centers on *reliability* tests for its *Performance* testing has already been evaluated by Eng. Jorge Araújo on [7]. It's testing centered around two topics:

1. Does the *Search Module* find the alternatives that match the user's requirements correctly?
2. Are the alternatives prices calculated correctly?

To find an answer to these questions, I recurred to the *CSA* example that accompanies the prototype. It's a simple but effective test, composed of two *Service Templates*, that allows me to verify the results for these two questions. Table.14 presents its detailed description.

CSA Example

ServiceTemplate1: VM	**Requirements (Minimum/Maximum)**	**Criteria (Minimize/Maximize)**
Gbout: 50	**Location**:Tokyo	No
Usagehours: 732	**CPUCores**-Needs to have	Yes, W=3, Max
	CPUSpeed: 2.5 Ghz, Min	Yes, W=4, Max
	MemorySize (RAM): 4, Min	Yes, W=4.2, Max
	DiskSize:150GB, Min	Yes, W=3.5, Max
	Performance: Needs to have	Yes: High,W=4.5, Min
	Price: 200, Max	Yes, W=5. Ind:12, Pref:24, Veto:40, Min
	Feature: Virtual Machine	No
	UNIX	No
ServiceTemplate2: DB	**Requirements (Minimum/Maximum)**	**Criteria (Minimize/Maximize)**
Gbout: 50	**IOOperations**: 1500, Min	No
Usagehours: 732	**StorageCapacity**: 100GB, Min	No
IOPSWanted: 1500	**Backup_Recovery**: Needs to have	No
NumberOfMonths: 1	**MemorySize**:3, Min	Yes,W=4.5, Max
GBStorageWanted: 100	**Performance**: Needs to have	Yes:High,W=4.7, Min
	Platform: MySQL	No
	Price: 480, Max	Yes:W=5. Ind:9,Pref:15,Veto:36, Min
	Location:Tokyo	No
	CPUSpeed	Yes:W=4, Max
	Feature: Database	

Table 14 - CSA testing example

Using this *CSA*, we can verify if the module is capable of searching for alternatives that match a specific set of requirements and, at the same time, we can verify if the module is capable of handling every type of requirement. The type of requirements being tested by the *CSA* are:

- Qualitative Requirement with value – *Location*

- Qualitative Requirement without a value (but needed) – *Performance*
- Quantitative Requirement with a minimum value – *DiskSize*
- Quantitative Requirement with a maximum value – *Price*

The server prints every alternative found for each *Service Template* (and its *attributes*) so it's easy to verify that each alternative found obeys the requirements set by the decision maker. With this, we know for sure that the searching capabilities of the module work as intended.

Regarding the second question, whether the prices for the alternatives are correctly calculated or not, it's easily to verified using third party tools supplied by the providers, like Amazon's web calculator[58], to calculate the actual cost of the service and see if it matches the one calculated by the prototype. At the time that this testing was performed, every price calculated by the prototype was correct and matched the information on the provider's tools.

Another approach is to manually perform the necessary calculations by fetching the values provided by the decision maker (regarding the pricing variables of the service) and, using the mathematical expression defined when creating the description of the service with the Linked USDL Pricing API (Section 5.3.1).

Finally, **Appendix E** presents the SPARQL query generated, and used, by the *Search Module* to retrieve the alternatives that match *ServiceTemplate1*.

6.3. Decision Module testing

The testing of the *Decision Module* centers on two topics: XMCDA problem description and results processing.

To assess the XMCDA description of the problem, there're two approaches we can follow:

- Use *XMCDA* schema to validate the initial *XMCDA* problem description
- Create the *XCMDA* problem description and submit the problem to check the behavior of the *web service*.

For a stronger validation, both approaches were followed: first, I validated the creation of the *XMCDA* description with the *XMCDA* schema provided by the Decision Deck group, followed by an analysis of the responses from the *web services* regarding the submitted problems.

It's important to note that it's only the first messages (created by the prototype) from each workflow that need validation. Subsequent messages are provided by the *web services* themselves thus, validation would be redundant.

Appendix F presents the *SOAP* messages exchanged between the first *step* of the PROMETHEE I's workflow and its response, for the *CSA* presented on Table.14. Every message exchanged between the prototype and each *MCDM's* workflow can be consulted in the Integration Testing document [52].

[58] http://calculator.s3.amazonaws.com/index.html

This document verifies the successful integration of the external *Multi-Criteria Decision Methods* into the new prototype. (Currently, as the workflow is executed, the server also prints every response for each of its *steps*).

Regarding the construction of the preference graphs, we can validate their creation (only PROMETHEE I's and ELECTRE III's) using the *diviz* application. D*iviz* uses an extra *web service* to generate the image of the preference graph; the prototype on the other hand, creates the graphs programmatically. Fig.49 presents the preference graph for *ServiceTemplate1* generated by the prototype (image generated by *GraphvizFiddle*[59]). Fig.50 presents the preference graph for *ServiceTemplate1* as well, but created by the *diviz* application.

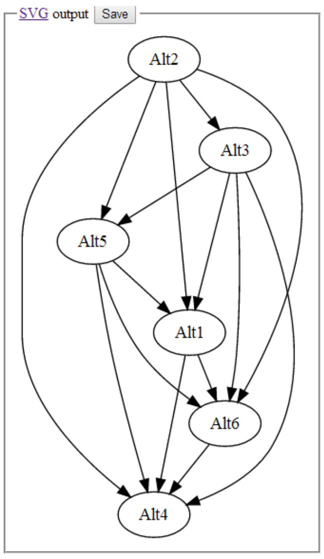


Fig. 49 - ServiceTemplate1's preference graph generated by the prototype (PROMETHEE I)

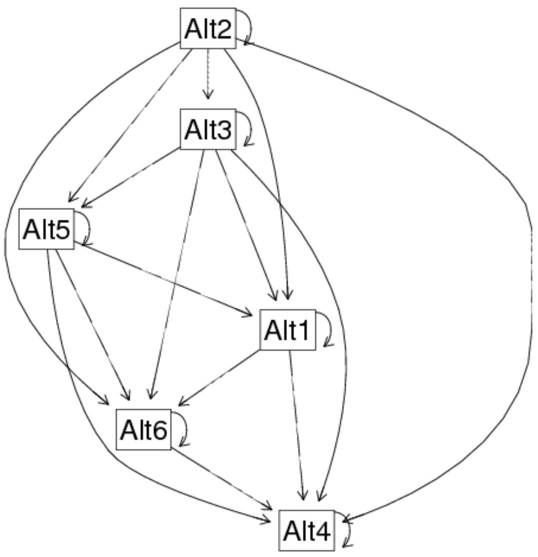


Fig. 50 - ServiceTemplate1's preference graph generated by diviz (PROMETHEE I)

[59] http://stamm-wilbrandt.de/GraphvizFiddle/

SMAA-2 and SAW results are much easier to validate for they're simply extracted by parsing the XMCDA result and fetching the corresponding values. However, the methods to transform their results into preference graphs were developed on this thesis. These methods are simple, relying mostly on finding a maximum value within a set of numbers and then placing an alternative based on that number. This approach is prone to errors and loss of information, it was adopted for simplicity sake, allowing for a faster development and to move onto the development of the aggregation algorithm. Section 7 presents other approaches that are most likely to enrich the results of the prototype and remove the shortcomings of the current approach.

6.4. Aggregation algorithm testing

The aggregation algorithm is a key component of the CloudAid2 prototype. As such, *performance* and *reliability* tests need to be performed in order to ensure that every objective concerning the *Aggregation Module* has been achieved.

Reliability testing centers around the actual results of the algorithm verifying its accuracy while, performance testing focuses on the behavior of the algorithm for differing quantities of information.

Reliability testing

As mentioned on section 5.3.4, least *dominated* solutions need to be visited and tested first. Going back to the example from section 5.3.4 – where we had two *Service Templates* and each had a preference graph like the one depicted on Fig.28 – we know that [*Alt3, Alt3*], along with its *incomparability* solutions, ([*Alt3,Alt4*], [*Alt3,Alt6*], [*Alt4,Alt3*], [*Alt6,Alt3*] and [*Alt3,Alt3*]. These solutions derive from the replacement of *Alt3* with its incomparable alternatives. Keep in mind that incomparable alternatives need to be treated equally which, basically means that if we chose *Alt3*, we need to choose its incomparable alternatives as well) are the first aggregated solutions that need to be placed into the *queue* and the ones to be tested first. Following, each of them needs is tested for its *admissibility* and, only in the case that they fail the test is that we move on to its children.

To verify if the algorithm traverses the *aggregated solution's tree* correctly, each solution needs to forcibly fail the *admissibility* test so the algorithm can move on to its children thus, preventing an early dismissal of the branch. This is easily done by creating an impossible global requirement like, stating that the overall cost of the aggregated solution should be equal to 0; which, of course, results into zero valid solutions. This will force the algorithm into visiting each node until there's no more nodes left to visit.

Let's take advantage of the preference graph from Fig. 49 and from the one depicted in Fig.51, which is the preference graph of *ServiceTemplate2*, to verify the behavior of the aggregation algorithm.

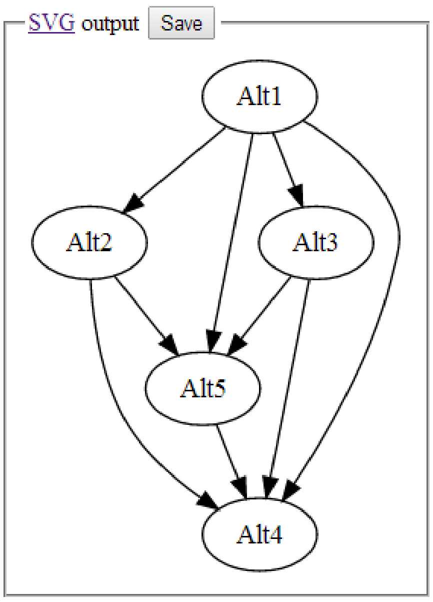


Fig. 51 - ServiceTemplate2's preference graph generated by the prototype (PROMETHEE I)

Considering this two preference graphs (Fig.49 and Fig.51), Fig.52 presents the first three levels of their aggregated solution's tree. These first three levels should prove sufficient to get a graphical idea the traversal of the aggregated solutions tree (Table.15). To get each node of the tree, you need to "move forward" on each preference graph, one at a time, choosing the least dominated alternative, and switch places with its father on the aggregated solution. Keep in mind that the algorithm won't test alternatives that have been already tested (visited) hence the "slash" on some nodes of the tree.

As you may notice, some of the solutions link to other solutions while some are linked to an empty circle. This means that the alternatives connected to the empty circle are incomparable with each other thus, needing to be treated (and considered) equally.

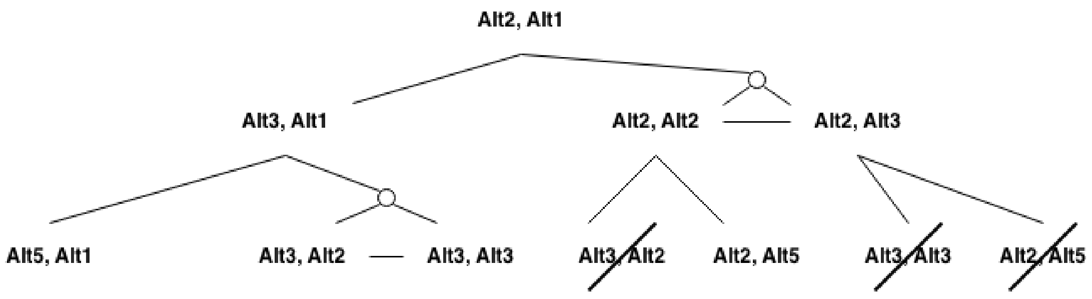


Fig. 52 - First three levels of the aggregated solution's tree from CSA Example

Now that we have both preference graphs and the aggregated solution's tree, we just need to create the impossible global requirement and see the order in which the ones are visited. The algorithm outputs the following order:

Order	*Visited solution*	
1	Alt2	Alt1
2	Alt3	Alt1
3	Alt2	Alt2
4	Alt2	Alt3
5	Alt5	Alt1
6	Alt3	Alt2
7	Alt3	Alt3
8	Alt2	Alt5
9	Alt1	Alt1
10	Alt5	Alt2
11	Alt5	Alt3
12	Alt3	Alt5
13	Alt2	Alt4
14	Alt6	Alt1
15	Alt1	Alt2
16	Alt1	Alt3
17	Alt5	Alt5
18	Alt3	Alt4
19	Alt4	Alt1
20	Alt6	Alt2
21	Alt6	Alt3
22	Alt1	Alt5
23	Alt5	Alt4
24	Alt4	Alt2
25	Alt4	Alt3
26	Alt6	Alt5
27	Alt1	Alt4
28	Alt4	Alt5
29	Alt6	Alt4
30	Alt4	Alt4

Table 15 - CSA Example tree traversal

Looking at Fig.52 and comparing it with the visiting order of the nodes presented on Table.15, we can conclude that the algorithm does traverse, and test, the aggregated solution's tree on the correct order.

Performance testing

To test the performance of the algorithm, an isolated testing class was created: *AggregationAlgorithm* class. This class creates results similar to those returned by SMAA-2 (section 5.3.3) to generate fictitious preference graphs. It creates a number of alternatives for a certain *ServiceTemplate*, attributes a random cost to each alternative (between 50 and 500) and afterwards, randomly attributes a (random) percentage to each possible rank of the alternative. Once it has generated the alternatives for each *ServiceTemplate*, it uses the *getSMAAGraphSolution()* method to create the adjacency list of each *ServiceTemplate*.

Once we have a set of preference graphs (adjacency lists), we can evaluate the performance of the aggregation algorithm.

The purpose of this test is to observe and evaluate the behavior of the algorithm with differing workloads of information. It focuses on two aspects: first we test the algorithm in the *worst* case scenario that is, when the algorithm needs to traverse the entire tree, visiting each node, to find a solution. Afterwards, we evaluate the behavior of the algorithm when it's not dealing with a *worst* case scenario.

Testing in the worst case scenario has an advantage: If we're not dealing with a *worst* case scenario (where no solution is found or the solution found is in the last node to be visited), the algorithm should solve the problem in less time than the one presented on Table.17. Table.16 presents the *CSA* generation parameters used for this test.

Parameter	**Value**
#Alternatives	[5,10,20]
#ServiceTemplates	[2,4,6,8]
SMAA indifference threshold	4%

Table 16 - Aggregation algorithm testing parameters

For example, looking at the first elements of #Alternatives and #ServiceTemplates from Table.16, the *AggregationAlgorithm* class would generate 2 preferences graphs, each with 5 alternatives.

For each test case, we evaluate the time (**T**, in seconds) it took to traverse the entire tree of aggregated solutions and the number of nodes **(*n*)** visited. Table.17 presents the results for each test case.

Number of Alternatives	***Number of Service Templates***							
	2		**4**		**6**		**8**	
	T (s)	***n***	***T (s)***	***n***	***T (s)***	***n***	***T (s)***	***n***
5	0,003	25	0.016	575	1.951	12800	1666.8	3.5*10^5
10	0.005	100	1.044	10000	10773.9	9.72*10^5	-	-
20	0.014	400	201.612	$16*10^4$	>25000	>15*10^5	-	-

Table 17 - Aggregation algorithm performance

Looking at table.17, we can see that the higher the number of *Service Templates* is, the higher should be the search parameter's specificity of the decision maker's problem. As you can see, in case there're two *Service Templates*, even if the decision maker isn't very specific about his requirements, the algorithm will still be capable of dealing with a considerably high amount of alternatives. However, if the decision maker maintains the same level of specificity for a higher number of *Service Templates*, the algorithm's performance deteriorates very rapidly due to the exponential increase in the number of possible solutions. As such, a balance between both would be the best approach.

As the number of *Service Template* increases, the decision maker should make a stronger effort in getting as much information about his target architecture as possible. Doing so, will strongly alleviate the working load of the algorithm, improving both accuracy of the results and the time required to compute them.

Augmenting the testing range, we can get a graphical perspective of the behavior of the algorithm for differing workloads. Recurring to the *mesh*[60] function from *matlab*, we get the surface presented on Fig.53 which, relates the number of *Service Templates* and the number of *Alternatives* with their corresponding number of visited nodes. Fig.54, in turn, presents the relation between the number of *Service Templates-Alternatives* and the amount of time it took the algorithm to visit every node of the tree.

Table.18 presents the full results in a table similar to Table.17. The parameter's ranges are.

- Service Templates (ST) – [2:8]
- Number of Alternatives(Alt) – [2:8]
- Indifference threshold – 4%
- Alternatives cost – [50,500]

Alts	***Number of Service Templates***													
	2		**3**		**4**		**5**		**6**		**7**		**8**	
	T(s)	***n***	***T(s)***	***n***	***T(s)***	***n***	***T(s)***	***n***	***T(s)***	***n***	***T(s)***	***n***	***T(s)***	***n***
2	0.0002	4	0.00005	7	0.0001	11	0.0001	21	0.0003	37	0.0007	53	0.0007	103
3	0.0001	9	0.0001	24	0.0003	73	0.001	195	0.0046	522	0.039	1491	0.236	3583
4	0.0001	16	0.00015	61	0.0011	220	0.009	893	0.13	3228	2.20	11074	28.3	43161
5	0.0001	25	0.00015	123	0.0033	600	0.082	2817	2.10	13344	48.9	61154	1415.5	308156
6	0.0005	36	0.0008	209	0.0131	1240	0.511	7262	16.9	40888	781.8	246240	8044	808564
7	0.0001	49	0.0012	333	0.048	2326	2.45	15720	105.4	105043	7606.2	749112	144328	3404288
8	0.0003	63	0.0023	502	0.138	4007	7.37	30992	611.47	249059	38234	1651888	183688	6851951

Table 18 - Aggregation Algorithm full performance results

Looking at the figures, the relation between the number of *ServiceTemplates* and *Alternatives* with the time required to compute the aggregated solutions becomes evident. As the decision maker increases the number of *Service Templates* on the *CSA*, he/she should take into consideration the degree of specificity of the requirements.

Now that we know how the algorithm behaves in its *worst* scenario, we can evaluate its behavior in a not *worst* situation, we just need to make sure that its global *price* requirement doesn't have its value set to zero. While it might not be set to zero, its value still has an impact on the behavior of the algorithm. For example, setting its value to 600 will find different admissible solutions than if it was set to 300 or 400. It might find (and probably will) admissible solutions on a higher level of the tree thus, terminating the search earlier than it would with 300 or 400 and as such, leading to a lower number of visited (tested) nodes.

Evaluating the algorithm on these situations it's important because the testing of the algorithm in a *worst* case scenario "skips" one of its steps: testing for dominance or incomparability of an admissible solution. As there're no admissible solutions in the worst

[60] http://www.mathworks.com/help/matlab/ref/mesh.html

case scenario, the time required to perform this operation is nearly zero as the list is always empty. With a "normal" global requirement, this (probably) won't be the case. The list will contain the aggregated solutions that match the decision maker's requirements (found so far) and, each time a new admissible solution is found, it needs to find out if it can be placed on the list as well. Of course, if the list no longer has a size equal to zero, the computational cost of this operation will be higher than on the *worst* case scenario.

To evaluate the algorithm's behavior on these situations, each *ServiceTemplate-Alternatives* pair was tested with differing values for its global price requirement. Table.19 presents the results of this *testing* for the pair *ServiceTemplates=6*, *Alternatives=8*.

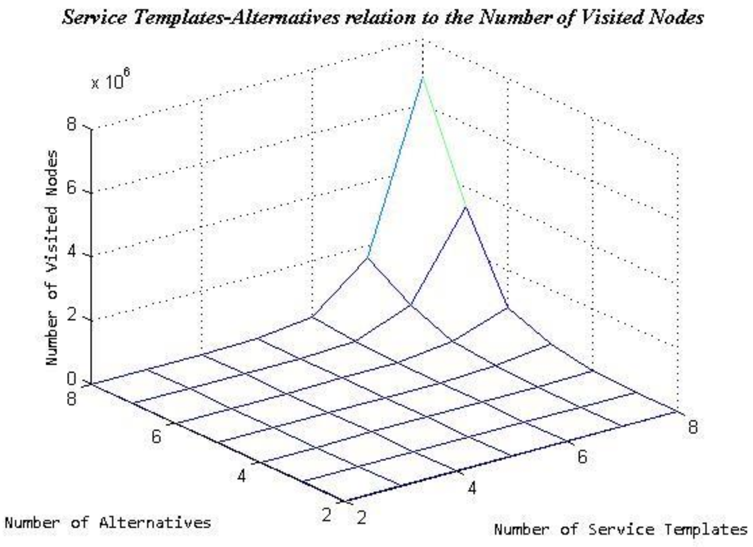


Fig. 53 - ST's-Alts relation to the number of visited nodes

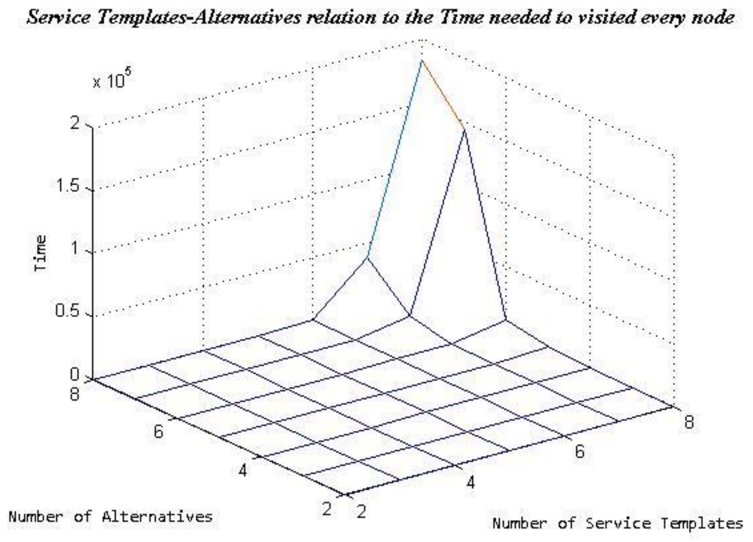


Fig. 54 - ST's-Alts relation to the time required to visit every node

Service Templates = 6 , Alternatives = 8			
Price	**Visited**	**Admissable**	**Time(s)**
300	247936	0	1170.2
600	247936	0	996.5
900	228378	300	693.6
1200	196622	6675	716.02
1500	165865	27706	709.93
1800	78908	29175	345.02
2100	8030	3852	13.78
2400	4	4	0
2700	4	4	0
3000	4	4	0

Table 19 - Global Requirement's Value variation for the pair ST=6,Alt=8

Looking at Table.19, we can see that the higher the value of the global requirement, the lower is the time required to compute the aggregated solutions. As the condition imposed by the global requirement *softens*, the time required to compute the aggregated solution highly diminishes due to the decrease in the number of nodes we need to visit. The *softness* of a global requirement is directly related to its value; for example, looking at the example presented on Table.19, assume that the maximum cost of any existing aggregated solution is 3000 (each alternative costs at max 500, and each aggregated solution is composed of one alternative from each *Service Template*). If the global requirement states that no aggregated solution should cost more than 300, it'll be extremely hard to find a solution with an overall cost lower or equal to that value thus, forcing the algorithm to visit almost every node of the tree, if not all, looking for an admissible solution. On the other hand, if the global requirement states that an aggregated solution may cost up to 2700, the probability of finding one is much higher (the constrain of the *admissibility* test has been *softened*) which means that the search on a branch of the tree will cease as soon as it finds an admissible solution. Terminating the search on higher levels leads to a decrease on the number of nodes that the algorithm will have to visit later on.

We can conclude that, while checking for incomparability/dominance has a certain computational cost, it's much less than the one required to traverse and visit every node of the tree. While the number of *Service Templates* and *Alternatives* play an extremely important role on the algorithm, its global requirements are also a critical point in the computation of the aggregated solutions. Fig.55 presents a graphical relation between Table.19's *price*, *visited* and *time* attributes.

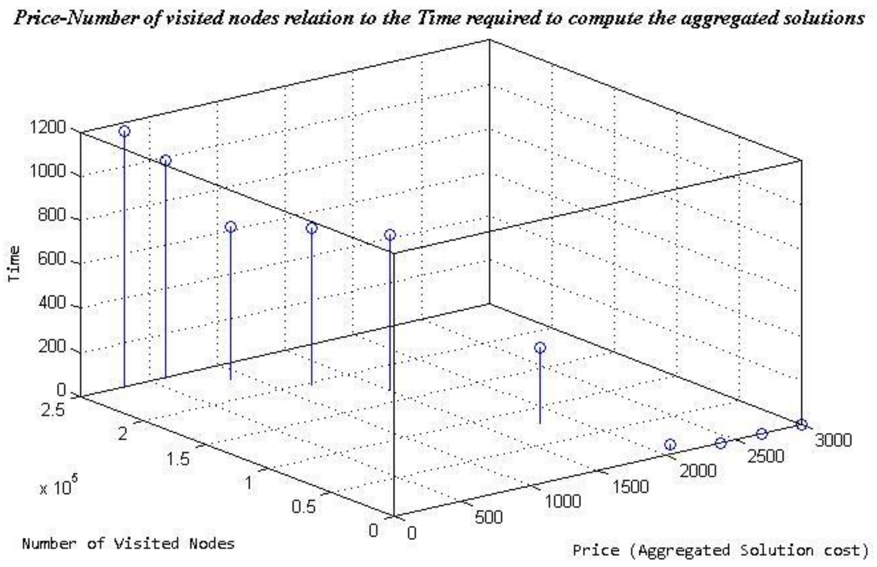


Fig. 55 – ST's=6, Alts=8 behavior for varying values of the global price requirement

Unfortunately, we cannot compare the results of the algorithm developed on the previous prototype (the algorithm with incomparability support), with the results of the new algorithm due to their diverging approaches and the way they deal with incomparability.

Briefly, the previous algorithms relied on a fully ordered list to create the aggregated solutions which led to some loss of information (solutions that were never visited or even considered), not only because the data structure itself is unfit for the purpose at hand and the *MCDM* used didn't even consider the existence of such concept, but also, because the algorithm too had its shortcomings regarding the concept of incomparability. To cope with this situation, the new algorithm relies on *graphs* which provides a much more suitable and reliable data structure, preventing the loss of information and at the same time, enhancing the performance of the algorithm.

Finally, every testing result presented on this section can be found on [52], under the *Testing* folder of the prototype.

7. Conclusions

This chapter presents the final conclusions of this thesis as well as contributions during its research and development phases. Section 7.1 begins by summarizing the work done, focusing on the key elements explained throughout this document. Contributions and some final conclusions are presented on section 7.2. Finally, section 7.3 presents future work that would most certainly improve the current state of the prototype.

7.1. Summary

The purpose of this thesis was to augment and improve the CloudAid1 prototype, an application that aims at supporting decision makers choose an appropriate aggregation of cloud services. To do so, it combines knowledge from several scientific areas as described on chapter 1.

Despite its promising results, there was a lot of room for improvement from a usability perspective and more importantly, from a technical perspective. In the earlier version of the prototype, services were fictional with a mere static value representing its cost. Not only that, it had very low support for the most common situation when one is faced with a *Multi-Criteria Decision Problem*: incomparable alternatives, where we can't say that alternative *x* is better than *y* or *y* better than *x* thus, needing to consider them both as viable choices. Another issue was the lack of a friendly graphical interface that'd ease the interaction of the decision maker with the prototype. The previous prototype's interaction was in charge of a simple shell application that had a rather limited usability and delivered an unpleasant experience to its users.

Given these shortcomings, this thesis started by presenting the Linked USDL Pricing API, a Java API created to ease and promote the adoption of Linked USDL core and pricing models for cloud service's descriptions. This API enables a programmatic interaction with the models, providing a mean to calculate the cost of a service dynamically.

Combining the API with *scrapping/parsing* techniques, the creation of service descriptions from real services data becomes much simpler. This created the *ServiceGatherer* project, a simple application that generates semantic descriptions using data extracted from the cloud provider's *web pages*. Including the API and these dynamic descriptions of the services highly improves the accuracy and realism of the prototype for it's now capable of calculating a personalized price for the user.

This thesis also presented a new aggregation algorithm that is capable of dealing with incomparability/incomplete information between alternatives, proposing a graph-theory based (more precisely, preference graphs) solution for the problem at hand. As well as, an augmented version of the *admissibility* test which is now capable of dealing with quantitative and qualitative requirements, aside the pricing requirement.

Finally, these concepts can be put in practice using the new graphical interface which provides a much smoother and comfortable experience to the decision maker. The prototype (the server side and its GUI) can be downloaded from [52], where it's also possible to find a video that demonstrates their usage.

7.2. Findings

Throughout the development of the project, a considerable amount of conclusions/findings were reported and discussed however, there're a few that we need to highlight.

The pricing validation performed on section 5.1 allowed to not only confirm that the Linked USDL pricing model (and API) is capable of tackling today's cloud pricing methodology, but also to prove its flexibility. The modeling of a service's pricing method can be done in multiple ways and each could be just as correct as the other, as long as the calculated cost matches the real one. For example, the user might decide to model a certain feature's discount directly on the service's *PriceComponent* responsible for calculating the cost of said feature or, create a new *PriceComponent* and marking it as a *deduction*.

In fact, this validation also proved useful to "point out" some of the limitations of the Linked USDL Pricing API allowing its continuous adjustment to the problems as they emerged.

The testing presented on chapter 6 allowed us to evaluate the influence of the *Composite Service Architecture (CSA)* on the prototype's behavior. A balance between the number of *Service Templates* and the specificity of its requirements (global and local) is something to strive for. The time it takes to visit a node depends on the number of *Service Templates*, the number of *Alternatives* that each possesses and, the *CSA's* global requirements. For example, a lower number of *Service Templates* will have higher tolerance for a low specificity on the *Service Template's* requirements (low specificity relaxes the filtering of alternatives) however, as the number of *Service Templates* increases, the *Service Template's* requirements should possess a higher specificity than the ones used with a lower number of *Service Templates*. Independently of the number of *Service Templates*, it's still necessary to be cautions when specifying its requirements for a (very) high number of alternatives will have serious repercussions on the time required by the prototype to compute the aggregated solutions.

On the other hand, constrains imposed by global requirements (GR) have the opposite effect. Independently of the number of *Service Templates* and *Alternatives*, if the constrains imposed by the GR have low specificity (are *soft*) the algorithm will rapidly compute the aggregated solutions. Low specificity on GR is especially useful for a high number of *Service Templates* and *Alternatives* for it broadens the admissibility test thus, preventing the search on lower levels of the tree where there's a higher number of possible aggregated solutions (especially, if there's a high degree of incomparability between alternatives).

Overall, we can say that the new version prototype (CloudAid2) is an improvement of its earlier version (CloudAid1) and that the proposed objectives were accomplished. Of course, despite its improvement, there's still room for improvement regarding several aspects of the prototype. The following section presents some improvements that could be considered for future work.

7.3. Future Work

Given the highly broad scope of the project, covering every scientific field to an extreme detail is impossible thus, leaving room for future development. Not only that, but as the project moves forward, new solutions and possible approaches can be proposed/discussed however, due to shortage of time, putting them to practice is near to impossible as well. Therefore, we hope that by presenting these possible solutions and the current state of the prototype, further research and development will build upon this concept. A list of possible improvements is presented below. This list encompasses two perspectives: improvements to the prototype from a technical perspective and, further researching topics;

- **Technical perspective:**

  - **[CloudAid2]** *GUI improvement* – The current GUI is still "raw", having a lot of room for improvement. Especially since we're talking about an extremely recent technology (JavaFX) and the functionalities inherent to it.

- **[Linked USDL Pricing API]** *Dynamic population of the Enum ontology interfaces* **–** The current version of the Linked USDL Pricing API uses static *Enums* to provide an interface between the semantic ontologies and the API. This approach should be replaced by a more dynamic approach (using The Owl API [61]for example), enabling the modification of the ontologies without the need of adapting the source code to said modifications.

- **[Linked USDL Pricing API]** *Mathematical Expression Converter expansion* **–** The current Mathematical Expression to SPARQL converter of the Linked USDL Pricing API is still very raw, supporting only basic mathematical operators (multiplication, division, addition, etc.) and a very limited group of functions (CEIL for example). SPARQL, however, has a much wider range of functionalities and as such, including those functionalities into the converter would greatly enhance its usability.

- **[CloudAid2] Web communication –** While the prototype doesn't recur to a standard web protocol to establish a communication between the server and it's GUI, its architecture enables its integration with ease. Both entities follow a *request-response* architecture which aims at emulating a typical *web* interaction. This however, does not dismiss future architectural changes for the chosen protocol.

- **[ServiceGatherer] Exceptions handling –** The current *ServiceGatherer* application has a fragile exception handling mechanism and a low degree of information regarding its exceptions. Further effort on this aspect would most certainly improve the usability of the application.

➢ **Research perspective:**

- **Bayesian Aggregation[62] –** The current approach to transform the results from SMAA-2 to a preference graph is prone to errors and loss of information. To prevent this, a Bayesian Aggregation approach could be considered.

- **Aggregation Algorithm optimization –** There's two approaches that can be considered to optimize the current aggregation algorithm: add an extra step to the algorithm that removes from the queue solutions that are dominated by the solution that is going to be placed into the queue or, create a new aggregation algorithm that follows a distributed approach (similar to *Divide-and-Conquer*[63]), dividing the problems into smaller problems, attributing each sub-problem to a *worker* instance and finally combining every result to get the final result.

[61] http://owlapi.sourceforge.net/
[62] http://www.stat.duke.edu/~mw/MWextrapubs/West1984.pdf
[63] http://www.cs.berkeley.edu/~vazirani/algorithms/chap2.pdf

- **Second layer of decision –** Once every *admissible* aggregated solution has been found, it's possible to request extra information from the decision maker to create a second *Multi-Criteria Decision Problem* but this time, evaluating the aggregated solutions as a whole thus, getting yet another preference graph.

- **Higher semantic service descriptions diversity** – The current *ServiceGatherer* project generates the description of approximately 9000 services however, they're only from two service providers. Having a higher diversity of service descriptions could lead to new findings and further improvement of the prototype or, at the very least, allows us to observe the behavior of the prototype in a much more interesting scenario.

- **Further aggregation algorithm testing** – From the results obtained on section 6.4, it was possible to observe and draw some conclusions about the algorithms behavior. However, further and a more thorough testing of the algorithm should be performed to assess the nature of the algorithm.

Appendices

Appendix A

CloudAid2 – Requirements

This document presents the requirements for the CloudAid2 prototype. These requirements were defined following a FURPS structuring and prioritized with the MoSCoW method [35]. They're divided into two major categories: Functional Requirements and Non-Functional Requirements. The last one (Non-Functional) is sub-divided into 4 different categories: Usability, Reliability, Performance and Support [36].

Functional Requirements

In this first section we present every functional requirement of the CloudAid2 prototype. Functional requirements are heavily linked to the general behavior and functionalities of the prototype [36]. For an easier understanding of the requirements, we decided to group them according to the module they belong to.

CloudAid2 – Functional Requirements

ID	Name	Priority
Search Module		
RF1	**Dynamic Price Calculation**	
	Description: The price of each *Service Offering* needs to be determined dynamically through the use of SPIN functions. These functions will possess two types of variables (*usage* and *constant*) and will be linked to the *Service Offerings* through the property *price:hasPriceFunction* (chapter 4 for further detail).	**MUST**
RF2	**Read and handle Linked USDL Cloud Services correctly**	
	Description: The Search Module needs to retrieve the relevant information of the services, along with their new Social property, from the TripleStore.	**MUST**
Service Set - ServiceGatherer		
RF1	**Use Scrape techniques to fetch and create our Linked USDL service descriptions**	
	Description: Fetch service description data from the web using scrape techniques. Once the data has been collected it'll be modeled using the JAVA data model described on section 4.4. Using this JAVA object representation we'll be able to create an RDF based description of the service.	**MUST**
RF2	**Create files with the RDF service description**	
	Description: The RDF service description needs to be exported to a file on the disk. This file will be later on imported into the TripleStore.	**MUST**

Decision Module		
RF1	**Decision Method selection**	
	Description: The Decision Method to use should be able to select the *MCDM* to use based on information provided by the user.	**MUST**
RF2	**New Multi-Criteria Decision Methods**	
	Description: New decision methods need to be added into the prototype. After a discussion with the project managers, the chosen methods were: • ELECTRE3 • PROMETHEE1 • SMAA-2	**MUST**
RF3	**Group similar alternatives**	
	Description: Alternatives with similar values on the defined criteria should be grouped into a singular entity and be treated as such. Let's imagine two alternatives, A_1 and A_2, being evaluated on criteria C_1 e C_2. Suppose $g_{C1}(A_1)$ returns the performance of alternative A_1 on the criteria C_1. In case that $g_{C1}(A_1) \approx g_{C1}(A_2)$ and $g_{C2}(A1) \approx gC_2(A_2)$, they should be grouped and from then on be considered as a single alternative.	**SHOULD**
RF4	**Thresholds definition**	
	Description: The user should be able to define several types of thresholds that can supplied to the decision methods and can improve the results. There are three different types of thresholds: **1.** Indifference **2.** Preference **3.** Veto	**MUST**
RF5	**Correct XMCDA description**	
	Description: Every relevant piece of information related to the problem, submitted by the decision maker and required by the chosen decision method, need to be mapped correctly into XMCDA.	**MUST**
RF6	**SOAP Message encapsulation**	
	Description: The XMCDA problem description needs to be encapsulated within a SOAP message that is to be sent to an external web-service made available by the decision deck. It's these web-services that perform the actual calculations of the methods.	**MUST**
RF7	**Multi-Criteria Decision Methods execution**	
	Description: The decision module is responsible for managing the execution of the *MCDM* chosen by the prototype. Depending on the method, its execution might be divided into several steps where each step is performed by a different web-service. Since the method needs to be executed on a sequential order, the decision module should keep track of the evolution of the calculations at all time.	**MUST**
Aggregation Module		
RF1	**Admissibility test expansion**	
	Description: In the current prototype, the only restriction that is possible to apply on an aggregation of services is a pricing restriction. Whenever the decision maker creates a global requirement of	**MUST**

	the type *price,* he's telling us that his aggregated solution cannot cost more than the amount specified. To obtain a more flexible and realistic aggregation of services, new type of restrictions should be added. Examples can be the requirement of the same OS on every machine or a guaranteed overall availability of 95% or higher.	
RF2	**Compatibility with the new results from the multi-decision criteria methods**	
	Description: The aggregation algorithms of the current prototype were developed assuming a specific structure for the ranked lists returned by the decision module. Since the structure of the new results has changed, we either need to adapt the current algorithms or create a new one based on the existing ones, in order to support the new structure of the preference relations.	**MUST**
Graphic Interface – Client		
RF1	**Add a new Service Template**	
	Description: The user needs to be able of adding new *Service Template's* to his search sheet. A *Service Template* represents a set of features, desired by the decision maker, for a component of his final aggregation of services.	**MUST**
RF1.1	**Add a requirement to a Service Template**	
	Description: The user needs to be able of inserting new requirements inside his *Service Template*. Defining a requirement is the way the decision maker has of informing us that he/she desires a certain feature on the corresponding alternatives. An example of a requirement can be *Memory*(RAM)=8192.	**MUST**
RF1.2	**Add a criteria to the Service Template**	
	Description: The user needs to be able of inserting criteria onto the Service Template. Criteria can be of the same type as the requirements but, contrary to the requirements which represent desires, criteria represent the features that will be used to evaluate the corresponding alternatives of the Service Template. An example of criteria might be the *DiskSize* , which can be used to evaluate alternatives of a service template whose main targets are databases.	**MUST**
RF2	**Add a global requirement**	
	Description: The user needs to be able of adding global requirements to his search sheet. These requirements will be shared by every alternative on the final aggregated solution.	**MUST**
RF3	**Send data to server using REST protocol**	
	Description: The client's graphic interface will gather every relevant information for the prototype, describe it using JSON format and send it to the server using the REST protocol.	**SHOULD**

RF4	**Receive and present results from the server**	
	Description: The graphic interface needs to be able of receiving the results sent by the server. These results will be on JSON format and should be handled accordingly.	**MUST**
RF5	**Re-use *Search Sheets* from the History**	
	Description: Past search sheets created by the user will be available for re-use on the History. This way the user won't be obliged to insert every piece of information again thus easing the interaction with the application.	**MUST**
RF6	**Insertion of the pricing variables**	
	Description: It'll be possible to define some of the most common pricing variables prior to the search of the alternatives. After searching for the alternatives, a list of variables (send by the server) that are missing can be presented and should be inserted in order to calculate the personalized price.	**MUST**
RF7	**Register**	
	Description: Given the nature of the new prototype, registration becomes imperative in order to avoid negative abuse of the application (e.g: *spam*). The user should provide the following information when registering: **1.** Username **2.** Password **3.** E-mail	**SHOULD**
RF8	**Login**	
	Description: Completed the registration, the user will be able of logging into the application and will be given access to the full functionalities of the prototype.	**SHOULD**
Server		
RF1	**Wait for data from the graphic interface**	
	Description: The server needs to listen for information sent by the graphic interface and handle it accordingly.	**MUST**
RF2	**Send the results to the client interface**	
	Description: Once the results have been obtained, these should be transformed into JSON and sent to the user that posted the corresponding problem.	**MUST**
RF3	**Send results to e-mail**	
	Description: In the case that some results take longer than expected, these should be sent to the e-mail provided by the user upon registration. This enables the user to keep using the application instead of waiting for the results of the problem.	**SHOULD**

RF4	**Keep past search sheets sent by users and their respective result**	
	Description: Past search sheets submitted by the user and their corresponding results should be saved. Due to the complexity of the information required by the system, the usage of a system to keep a history of search sheets becomes imperative. Even with the addition of a graphic interface to aid in the process of the problem description, this remains a bothersome tasks. In case a specific search sheet doesn't return the expected results or none at all, instead of introducing the same information again the user will be able to modify and re-use the respective search sheet.	**MUST**
RF5	**Access control**	
	Description: The server is responsible for managing the access of users to the application. Only registered users will be able to use the prototype.	**SHOULD**

Non-Functional Requirements

On the previous section we presented the requirements that fell under the *Functional* ('F') category of the FURPS structuring. This section will tackle the non-functional requirements which are divide into the following categories: Usability – 'U', Reliability – 'R', Performance – 'P' and Support – 'S'. These requirements are related to the properties and needs of the system rather than its functionalities [36].

- Usability Requirements

These requirements are related to the graphic design of the user interface and how the information is presented. Basically, they're focused on the client's needs [36].

Usability Requirements – CloudAid2

ID	Name	Priority
RU1	**Present the results on a clear and objective way**	
	Description: The presentation of the results on the graphic interface should be performed in simple, yet clear and objective way. When looking for the first time to the results, the user should be able to identify immediately which is best aggregation for his needs.	**MUST**
RU2	**Dynamically retrieve relevant data**	
	Description: Every relevant data should be retrieved by the graphic interface, prior or in the middle of the process.	**MUST**
RU3	**Simple and objective *History***	
	Description: The *history* provided to the user should be an easy and rather intuitive tool to use.	**MUST**

RU4	**A friendly and ergonomic graphic interface**	
	Description: The interaction between the user and the graphic interface needs to be as smooth as possible. The functionalities should be presented in a simple and objective way, easing the tasks required by the user.	**MUST**
RU5	**MCDM choice**	
	Description: The multi-criteria decision method to use should be inferred from a series of questions or through other indirect methods, completely transparent from the user.	**SHOULD**

- Reliability Requirements

Reliability requirements are related to the availability of the system, fail-over mechanisms and system accuracy [36]. The user is waiting for results that may play a critical role on his/hers business/project therefore, we need to ensure their correctness and validity.

Reliability Requirements – CloudAid2

ID	**Name**	**Priority**
RR1	**Correct price calculation**	
	Description: The price component usually plays a critical role in the decision making process of the user. In order to promote a valid decision and avoid inducing the user into wrong conclusions, the price calculation needs to be correct at all time.	**MUST**
RR2	**Correct aggregated solutions**	
	Description: The recommended service compositions need to respect the user's requirements (global and local) as much as possible.	**MUST**
RR3	**Results delivery**	
	Description: The results need to reach the user, be it via direct delivery through the user interface or via e-mail, which he provided upon registration.	**MUST**

- Support Requirements

These requirements are related with the structure of the system and the degree of extensibility/changeability it possesses [36]. This is a rather important characteristic since this is a prototype that will probably be changed in the future.

Support Requirements – CloudAid2

ID	Name	Priority
SR1	**Modular development**	
	Description: Independently of the changes made to the system, its modular characteristic needs to be maintained. This means that every change/new functionality needs to be implemented inside it's corresponding module ensuring the ease of extension and modifiability of the prototype.	**MUST**

- Performance Requirements

These type of requirements are related to the general performance of the system [36]. Given the nature of the new architecture for this version of the prototype, we need to take into consideration some metrics like response time or, the number of concurrent client's our server can handle in order to ensure a minimum quality of service. While important, these won't be considered as mandatory but they shall be taken into account in the development stage.

Performance requirements – CloudAid2

ID	Name	Priority
PR1	**Acceptable response time**	
	Description: The time it takes for the server to answer a request from the user should be around 200ms [8].	**COULD**
PR2	**Acceptable number of concurrent clients**	
	Description: Optimize the server in order to support a reasonable amount of concurrent users. A minimum of 5 to 10 concurrent users should be able of using the application without, or with minimum, loss of quality.	**COULD**

Appendix B

CloudAid1

In this section I present an overview of the current CloudAid1 prototype. I'll present each element of the system, their objective, how they work and interact with each other.

Section 4.1 provides an overall view of the system and its components. From section 4.2 to section 4.6 I'll describe each of these component with further detail, presenting the reason why they were created and their objective. This analysis of the CloudAid1 was supported by [7] and a personal study of the prototype.

4.1 Overall Architecture

The CloudAid1 application was built using a Model-View-Controller (MVC) approach. With the help of Fig.1, which was created using the *Fundamental Modeling Concepts* (FMC) notation, we can visualize a high level representation of internal components of the system.

FMC complements the software-description achieved by UML, providing a set of tools to describe the system's structures, communication channels and internal flow [21]. It enables the description of architectural components leaving the software specification to UML. It's important to note that several diagrams presented on this document don't follow a strict FMC notation in order to provide a higher degree of detail on the system's architecture (while it's not strictly followed they're heavily based on it).

As we can see on Fig.1, the CloudAid1 prototype is composed by five modules that are coordinated by a sixth model (the *Controller*) whose main objective is to monitor and control the flow of execution, making sure that each of the components receives the data they need to do their job. I'll make a brief introduction for each of the components to provide a global idea of how the system works from the beginning. A more detailed description of the modules will be presented on the following sections.

- UI – This component is the 'View' on the MVC model. It's the interface between the user and the application, it's responsible for capturing/presenting information from/to the user.

- Controller – 'Controller' in the MVC model. It's responsible for mediating information transactions between the UI and the rest of the components. It's also responsible for initializing and controlling the system and its execution.

- Composite Service Architecture (CSA) [7] Evaluator – Its part of the 'Model' in the MVC model. Responsible for evaluating and preparing the CSA data. The data is inserted by the user thus, some measures must be taken in order to guarantee that there are no problems with it.

- Search Module – It's part of the 'Model' in the MVC model. Responsible for retrieving alternatives that match the user's requirements from the *TripleStore*.

- Decision Module – Component from the 'Model' in the MVC model. Responsible for ranking the alternatives of certain *Service Template*.

- Aggregation Module – It's part of the 'Model' in the MVC model. Responsible for computing the aggregated solutions. Aggregated solutions are composed by one alternative from each *Service Template*.

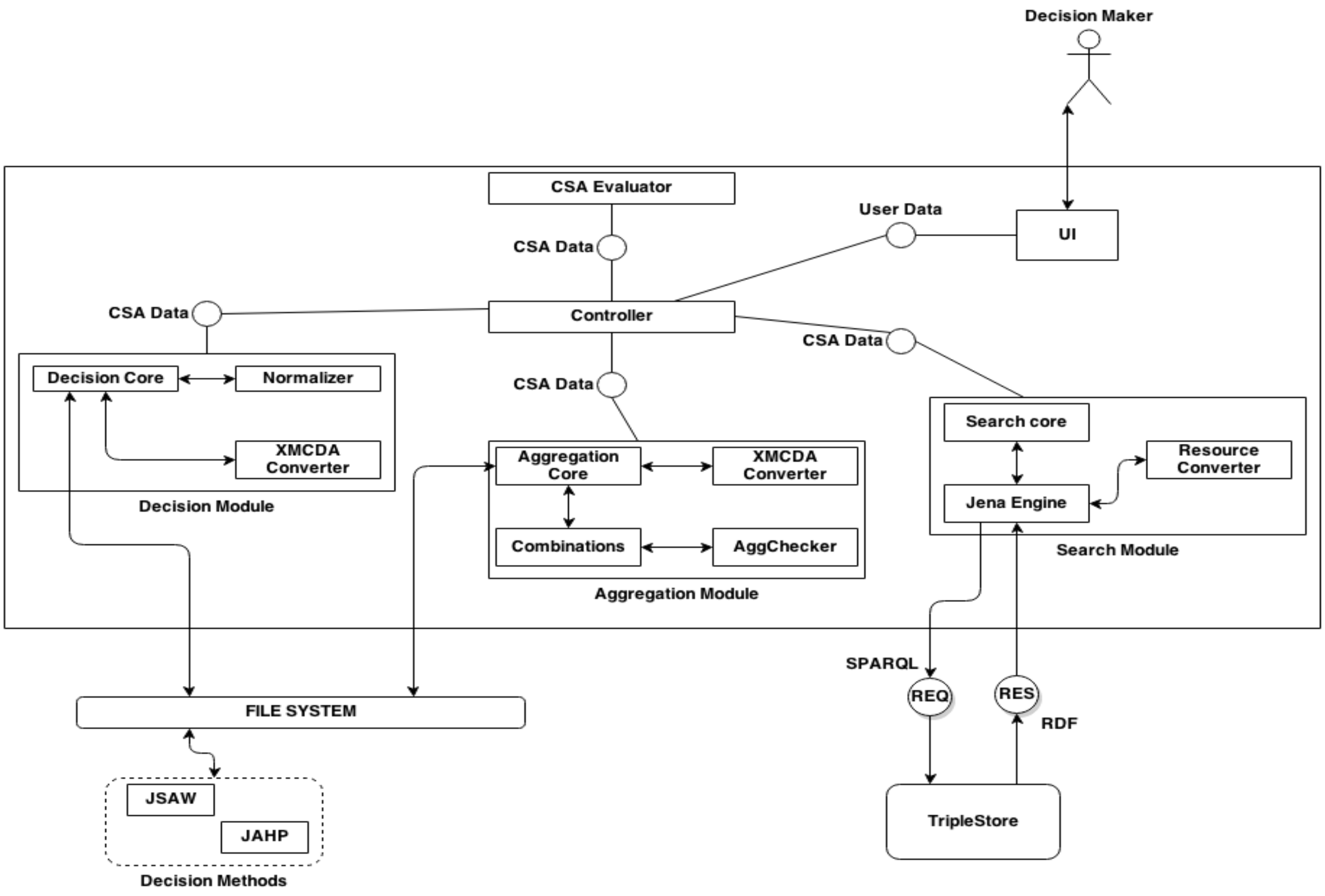


Fig.1 – CloudAid 1 – High Level Architecture

4.2 The Controller and the User Interface (UI)

4.2.1 User Interface

UI stands for *User Interface* and as the name suggests, it's the point of interaction between the user and the application. It's where the user inserts the data required by the application and defines the characteristics he desires for his composite service aggregation (*CSA*). We can categorize the data required by the application on four topics:

- *Service Template*
- *Requirements*
- *Criteria*
- *Preferences*

The first two are related to the service characteristics desired by the user while the other two are metrics and information that's be used by the prototype to find alternatives that match the user's preferences.

To capture and represent this information, a Java data model was created. Following, I present its class diagram and describe of each of its components.

As mentioned earlier, the CSA is the container for the aggregation's features/characteristics; we can look at it as a sort of template into which the alternatives need to fit. If we look at Fig.2 we can identify five entities:

- **CSAData** – It's the container for every entity listed below. Every time the user wishes to request a new service aggregation, a new *CSAData* is created to hold its description.
- **ServiceTemplate** – It's the representation of a service. This class holds a set of characteristics that a certain alternative on the final aggregated solution needs to possess. An example:
    - *ServiceTemplate1:*
        - *Disk Size: 500Gb*
        - *RAM: 5Gb*
        - *CPUCores: 2*
        - *...*
    - *ServiceTemplate2:*
        - *Disk Size: 100Gb:*
        - *RAM: 8Gb*
        - *CPUCores: 4*
        - *CPUArchitecture: 64bits*

Each alternative on the final solution needs to respect the rules specified on their corresponding *ServiceTemplates*.

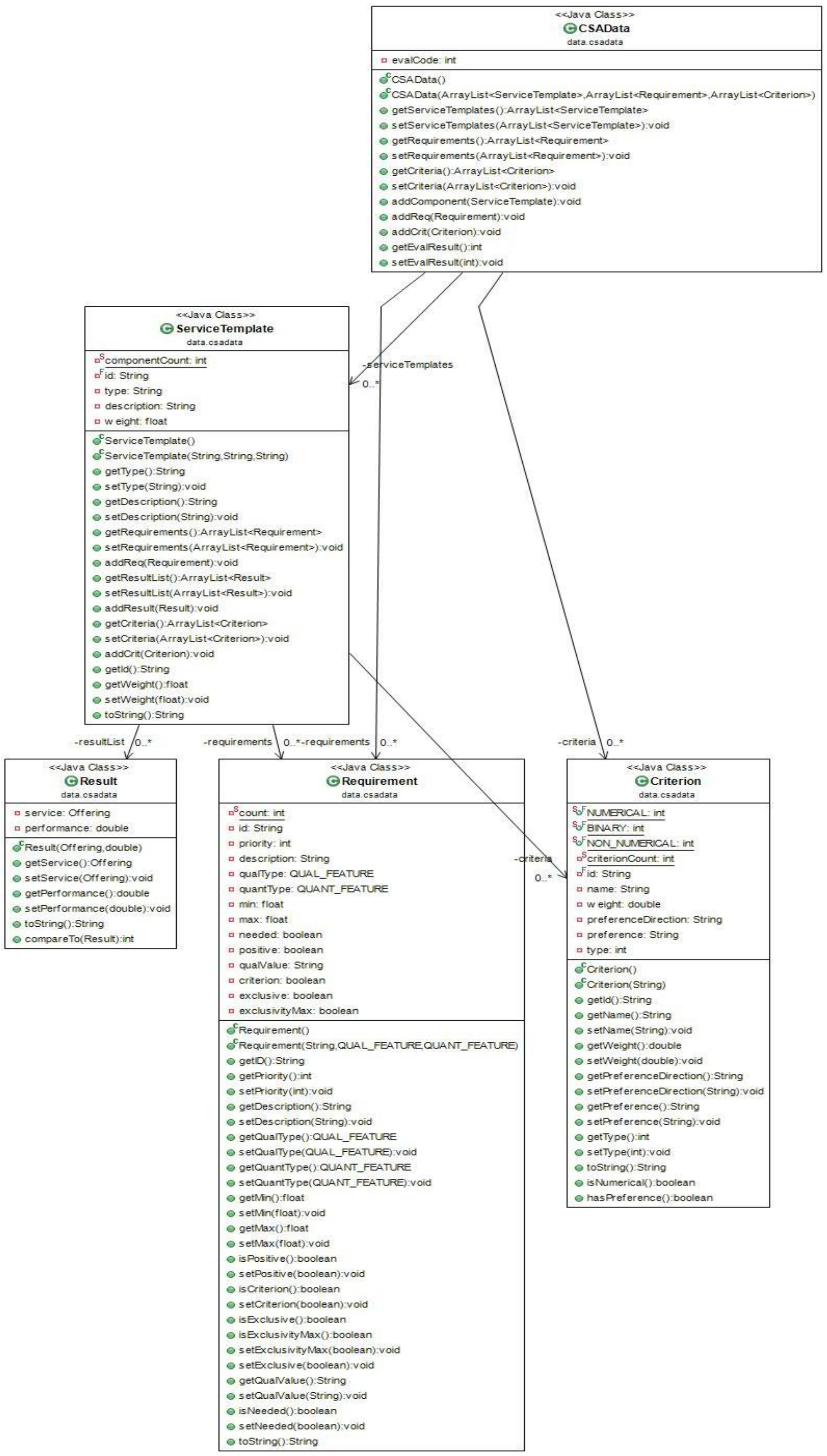


Fig.2 – CSA Java Data Model

- **Requirement** – Is a resource or a capability the cloud service needs to or should possess. If we look at the examples above, a Requirement would be for example *Disk Size: 500 Gb*. A requirement can either be *exclusive* or *non-exclusive*. An *exclusive* requirement means that every alternative that does not comply with it will be discard. With *non-exclusive,* even if the alternative fails to comply, it will still be considered a valid alternative. A requirement can also be classified has a *Qualitative* or a *Quantitative* requirement but I'll address this again with further detail on Section 4.5. Finally, a requirement has two possible scopes: *ServiceTemplate* (local level) or *CSA* (global level). If a requirement is defined inside a *ServiceTemplate*, only the alternatives of that template will be under its influence else, we're dealing with a requirement that'll influence the aggregation of the cloud services.
- **Criteria** – A criterion is created in the same way as the requirements and it too respects the same rules however, its purpose is different. While requirements represent rules that allows the prototype to discard alternatives with no interest to the user, a criterion is a variable that will be used by the decision methods to evaluate the alternatives. When the user creates a criterion, he's saying that each alternative of that *ServiceTemplate* will be evaluated using that attribute's value.

  There's another important aspect about criteria that needs to be addressed: *preference direction*. This is the way the user tells us if he wants to *maximize* or *minimize* the attribute's value. An example of maximization might be *StorageCapacity* where the user might say that the higher the disk size the better; a good *minimization* example is *Price* where one usually wants to spend as minimum as possible but at the same expect a good/decent product. Of course, it might happen that both *maximization* and *minimization* are not good enough to find the optimum value; when this happens the user has the possibility to define the value he considers best.

- **Result** – Entity responsible for holding the results obtained by the decision module. It holds the alternative's data and its corresponding performance value.

Preferences are pieces of information that are needed by the modules to perform their job and are requested throughout the execution process. It's usually related to a preference value or a criterion weight.

Now, let's see some screenshots of the current UI and how the data is captured by the CloudAid1 prototype.

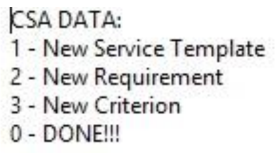
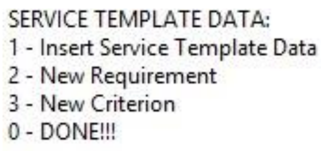

```
CSA DATA:
1 - New Service Template
2 - New Requirement
3 - New Criterion
0 - DONE!!!
```

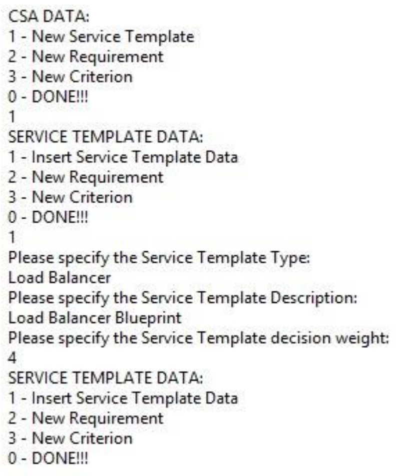
Fig.3 – CloudAid1 – CSA Menu



```
SERVICE TEMPLATE DATA:
1 - Insert Service Template Data
2 - New Requirement
3 - New Criterion
0 - DONE!!!
```

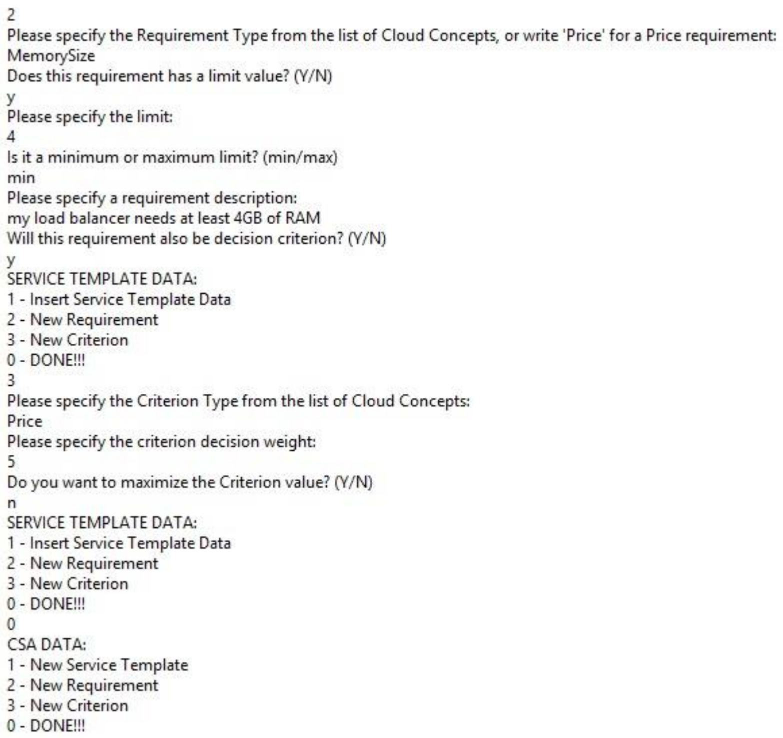
Fig.4 – CloudAid1 – Service Template Menu

Fig.3 shows the menu that is first presented to the user. From here, the user is capable of creating new *Service Templates* (option 1), global Requirements (option 2) and global Criteria (option 3). Fig.4 shows the prompted menu when the user chooses to create a new *Service Template*; from here, the user can insert create its respective Requirements and Criteria.



```
CSA DATA:
1 - New Service Template
2 - New Requirement
3 - New Criterion
0 - DONE!!!
1
SERVICE TEMPLATE DATA:
1 - Insert Service Template Data
2 - New Requirement
3 - New Criterion
0 - DONE!!!
1
Please specify the Service Template Type:
Load Balancer
Please specify the Service Template Description:
Load Balancer Blueprint
Please specify the Service Template decision weight:
4
SERVICE TEMPLATE DATA:
1 - Insert Service Template Data
2 - New Requirement
3 - New Criterion
0 - DONE!!!
```

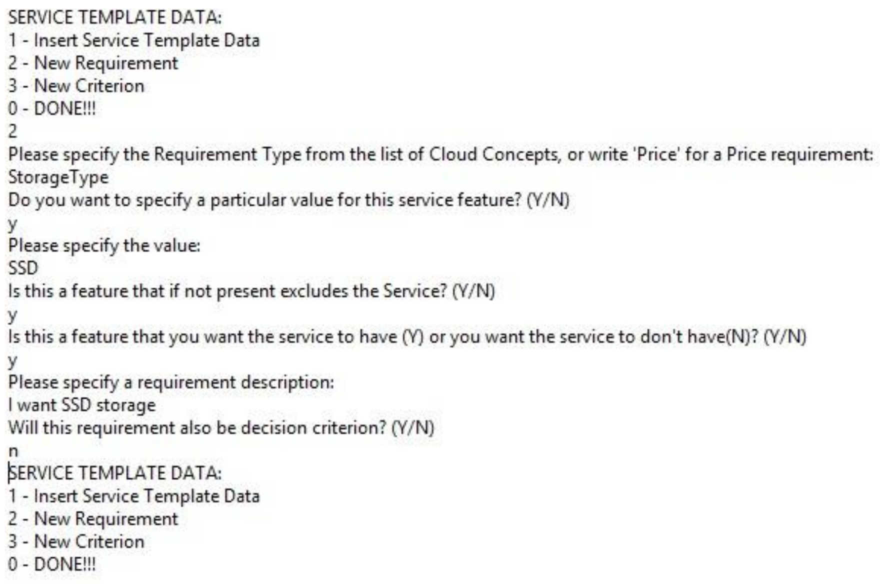
Fig.5 – CloudAid1 – New Service Template



```
2
Please specify the Requirement Type from the list of Cloud Concepts, or write 'Price' for a Price requirement:
MemorySize
Does this requirement has a limit value? (Y/N)
y
Please specify the limit:
4
Is it a minimum or maximum limit? (min/max)
min
Please specify a requirement description:
my load balancer needs at least 4GB of RAM
Will this requirement also be decision criterion? (Y/N)
y
SERVICE TEMPLATE DATA:
1 - Insert Service Template Data
2 - New Requirement
3 - New Criterion
0 - DONE!!!
3
Please specify the Criterion Type from the list of Cloud Concepts:
Price
Please specify the criterion decision weight:
5
Do you want to maximize the Criterion value? (Y/N)
n
SERVICE TEMPLATE DATA:
1 - Insert Service Template Data
2 - New Requirement
3 - New Criterion
0 - DONE!!!
0
CSA DATA:
1 - New Service Template
2 - New Requirement
3 - New Criterion
0 - DONE!!!
```

Fig.6 – CloudAid1 – New Requirement and Criterion

Fig.5 presents an example of the creation of a *Service Template* while Fig.6 presents an example of the insertion of a new *Requirement* and a new *Criterion*.



```
SERVICE TEMPLATE DATA:
1 - Insert Service Template Data
2 - New Requirement
3 - New Criterion
0 - DONE!!!
2
Please specify the Requirement Type from the list of Cloud Concepts, or write 'Price' for a Price requirement:
StorageType
Do you want to specify a particular value for this service feature? (Y/N)
y
Please specify the value:
SSD
Is this a feature that if not present excludes the Service? (Y/N)
y
Is this a feature that you want the service to have (Y) or you want the service to don't have(N)? (Y/N)
y
Please specify a requirement description:
I want SSD storage
Will this requirement also be decision criterion? (Y/N)
n
SERVICE TEMPLATE DATA:
1 - Insert Service Template Data
2 - New Requirement
3 - New Criterion
0 - DONE!!!
```

Fig.7 – CloudAid1 – New Qualitative Requirement

Finally, Fig.7 presents an example of the insertion of a *Qualitative* requirement and its respective value.

4.3.2 Controller

The *Controller* is the main component of the CloudAid 1 prototype. It's responsible for managing the whole system, making sure that every other component receives the data they need and, at the same time, it controls the flow of the execution process. Fig.8 presents a diagram, created using an informal notation, that'll help us get a clearer image of the Controller's objective and the flow of the execution of the CloudAid1 application.

As we can see in the picture, this module acts a data dispatcher for the other modules in a sequential order. After one module has done its job, the flow of execution goes back to the controller who will initiate the next step in the process. Knowing this, and after a brief analysis of the Fig.8, we can identify the core steps of the execution process:

- **a.** Fetch CSA from the user
- **b.** Evaluate CSA – CSA Evaluator
- **c.** Search for alternatives that match the user's needs – Search Module
- **d.** Rank the alternatives of each Service Template – Decision Module
- **e.** Create aggregated solutions – Aggregation Module

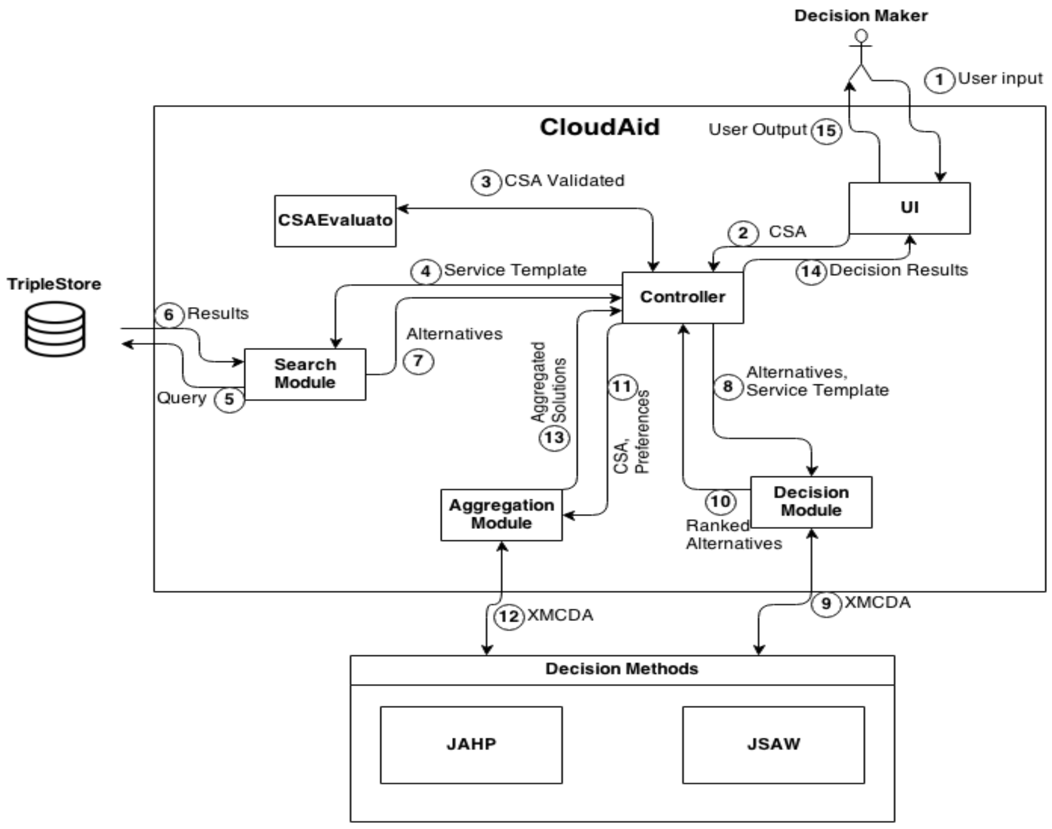


Fig.8 – CloudAid1 – Controller's flow of execution.

It's important to note that every module has its own execution process that will be explained on the following sections but for now, the main objective of this section is to provide an overall perspective of the system and how it works.

Controlling the flow of execution and acting as a data dispatcher for the other modules is but one of the four "jobs" the controller is in charge of:

1. Environment setup
2. Choose Decision Method
3. Manage the flow of execution
4. Establish a link of communication between the UI and the other modules.

Task 2 and 3 are sequential while task 1 is executed only once and task 4 is executed at the same time as tasks 2 and 3.

- **Environment setup** – This task is responsible for instantiating the rest of the system's components that is, it "creates" the UI followed by the Search Module (Section 4.3), then the Decision Module (Section 4.4) and finally the Aggregation Module (Section 4.5).
- **Choose Decision Method** – The CloudAid1 prototype supports two different Multi-Criteria Decision Methods: Simple-Additive-Weighting (SAW) and Analytic Hierarchic Process (AHP). To decide which of the methods should be used a simple question is asked to the user: "Are you comfortable giving weight to the criteria?" In case he answers 'y' SAW will be used otherwise, AHP will be used.

- **Manage flow of execution** – Once each of the modules has been instantiated (task 1) and the decision method has been chosen (task 2), the application can start its main objective: find aggregated solutions of cloud services that go towards the user's needs. This process starts by evaluating the CSA created by the user (performed by the CSAEvaluator module). If it succeeds, the Controller can start the next stage: Search for alternatives. This step is performed by the Search Module and it searches alternatives for each *ServiceTemplate* on the CSA. Once the search module finishes retrieving the alternatives, it's necessary to rank them according to the defined criteria. This step is performed by the Decision Module. When every Service Templates has its corresponding alternatives ranked, the final step of the application begins: compute the aggregated solutions. This final step is performed by the Aggregation Module.
- **Link of communication** – This is the final responsibility of the controller. The Controller is the communication link between the different modules of the system and, between the UI and the application. Every time a module needs to interact with the user, it calls the Controller who in turn invokes the proper functionality on the UI to communicate with the user. When the user finishes his interaction, the controller is then in charge of transferring the inserted information back to the module. Fig.9 shows how this request is processed at an architectural level.

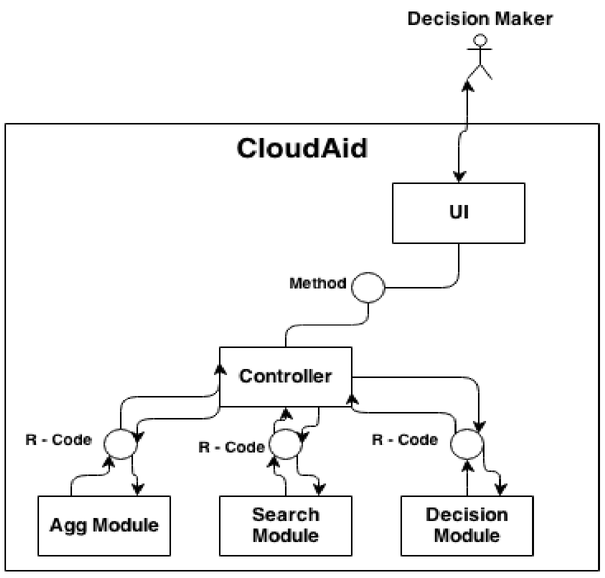


Fig.9 – CloudAid1 – Model-View Communication

4.4 Search Module

This module is responsible for finding alternatives that match the decision maker preferences. It performs three major tasks:

1. Divide the exclusive requirements from the non-exclusive requirements
2. SPARQL Query construction.
3. Fetch matching alternatives based on the exclusive requirements
4. Enrich the found alternatives with the service offering attributes

The reason a distinction is made between exclusive and non-exclusive requirements is because of the impact they have on the results. Exclusive requirements are those that actually work as a "filter" on the search mechanism since these represent conditions imposed by the decision maker, alternatives that don't obey them will be left out. Non-exclusive requirements are features that are desired by the Decision Maker however, they're not relevant enough to discard alternatives that don't match up to them.

To be considered exclusive, a requirement should meet at least one of the following conditions:

- Linked to a criterion – By specifying a certain feature as a criterion the user is saying that every alternative will be evaluated by their performance on it. In case the feature isn't linked to a criterion the user is simply saying "I want this feature but it's not important enough to discard alternatives that don't match my target value for this feature".
- Has a maximum value – If the Decision Maker defines a maximum value on a feature, every alternative that surpasses said value needs to be discarded thus, every feature with a maximum value is considered to be exclusive.
- Has a minimum value – Follows the same principle as the one described above.

Once this distinction between requirements has been established, we can move to the following step: the SPARQL Query construction.

There are three types of requirements:

- Quantitative Requirements
- Qualitative Requirements
- Price Requirement

The SPARQL query that retrieves every service offering from the TripleStore is composed of multiple nested SPARQL *selects*, also known as *sub-queries*. For example, assume that the consumer creates a *Service Template* with three requirements:

- Maximum cost of 300 euros (price requirement)
- Running MySQL (qualitative requirement)
- With at least 150Gb of disk (quantitative requirement)

Ignoring SPARQL syntax, the final query would be something like:

SELECT offering
WHERE ((every offering with price <= 300)
Ω (every offering running MySQL)
Ω (every offering with at least 150Gb disk))

Once the SPARQL query is build, we move to the next step: fetching the actual services from the TripleStore. To retrieve the service offerings we just need to run the SPARQL query over the TripleStore. When finished, it returns every *ServiceOffering* resource[64] that respects every exclusive requirement defined by the user.

The exclusive requirements identification and the SPARQL query building are responsibilities of the *JenaEngine* class which, as the name implies, uses the Apache Jena Framework for RDF objects manipulation and storage [22].

Once we have every *ServiceOffering* resource, we can move to the last step of the module: resource conversion. To ease the following steps of the CloudAid, the information from the semantic model is "loaded" into Java classes which, are later on passed onto the following modules. The class responsible for the TTL/RDF to Java conversion is the *ResourceConverter* class. Using the returned resources from the earlier search on the TripleStore, it is able to extract the remaining information (its features and their corresponding values) related to the *ServiceOffering* from the semantic model.

Once these steps have been completed the module's job is done and, all that is left to do is to return the results to the *Controller* in order to proceed with the flow of execution.

Fig.10 presents the module's execution flow from an architectural point of view, showing in an ordered way the steps described on this chapter.

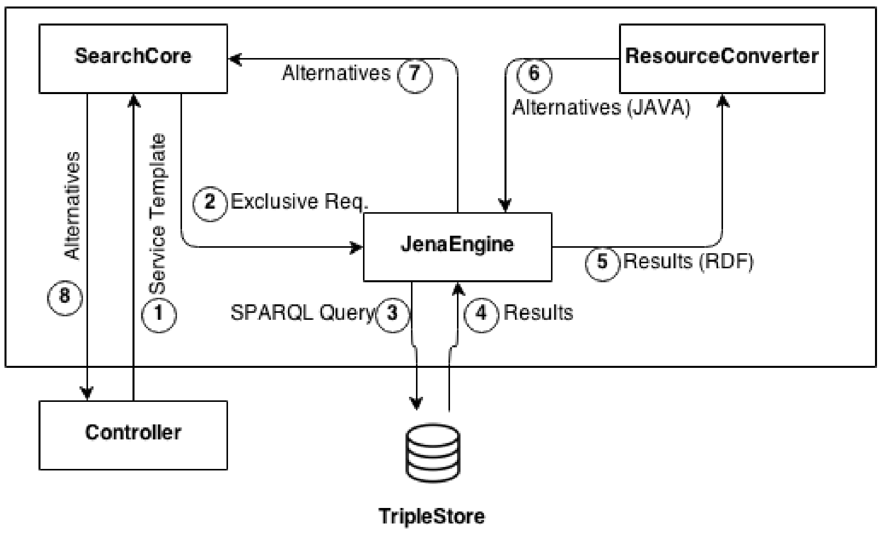


Fig.10 – CloudAid1 Architecture - Search Module

[64] http://jena.apache.org/documentation/javadoc/jena/com/hp/hpl/jena/rdf/model/Resource.html

4.5 Decision Module

After retrieving the Service Template's possible alternatives, it's time to rank them according to the criteria defined by the Decision Maker. Here is where the Decision Module comes in. It retrieves the criteria, the user's preferences and asks other relevant information to produce a valid ranking list of the alternatives for a particular service template. This list has the form of a typical ranked list where the head of the list is the best alternative and the last one the worst. This ranking is accomplished using *Multi-Criteria Decision Methods* (MCDM) that receive data inserted by the Decision Maker, process it, and return the desired ranked list.

In order to get a correct result from the *MCDM*, it's necessary to normalize the data (section 4.4.1) and then, transform it into XMCDA[65] format (section 4.4.2). Once the problem is in XMCDA format, it's published into a pre-defined directory. External applications that are monitoring the directory will fetch the files that are newly written, calculate the ranking of the alternatives and publish the results on another pre-defined directory. This directory is, in turn, monitored by the Decision Module. It will fetch them as soon as they're created and map the results to Java Objects (section 4.4.3).

We can split the Decision Module in 5 steps:

1. Data normalization – Normalization is needed since data from the Service Template can belong to different intervals, adding undesirable noise and behaviors to the methods calculations.
2. Express the problem in XMCDA format – After normalizing the data, the problem should be described in XMCDA format (section 4.4.2).
3. Publish XMCDA file – Once it's described in XMCDA, it's written into into a file on a pre-defined directory for the external *MCDM* application.
4. Read and Transform the Results – After publishing the problem, the module will monitor a pre-defined directory waiting for the results to be published. Once they're published, they'll be read and transformed into Java objects.
5. Obtain and Sort the ranked List – The results from the *MCDM* may not be ordered correctly thus, before returning the results onto the Controller, a descending ordering of the list is performed.

Fig.11 describes the complete flow of execution of the Decision Module.

[65] http://www.decision-deck.org/xmcda/

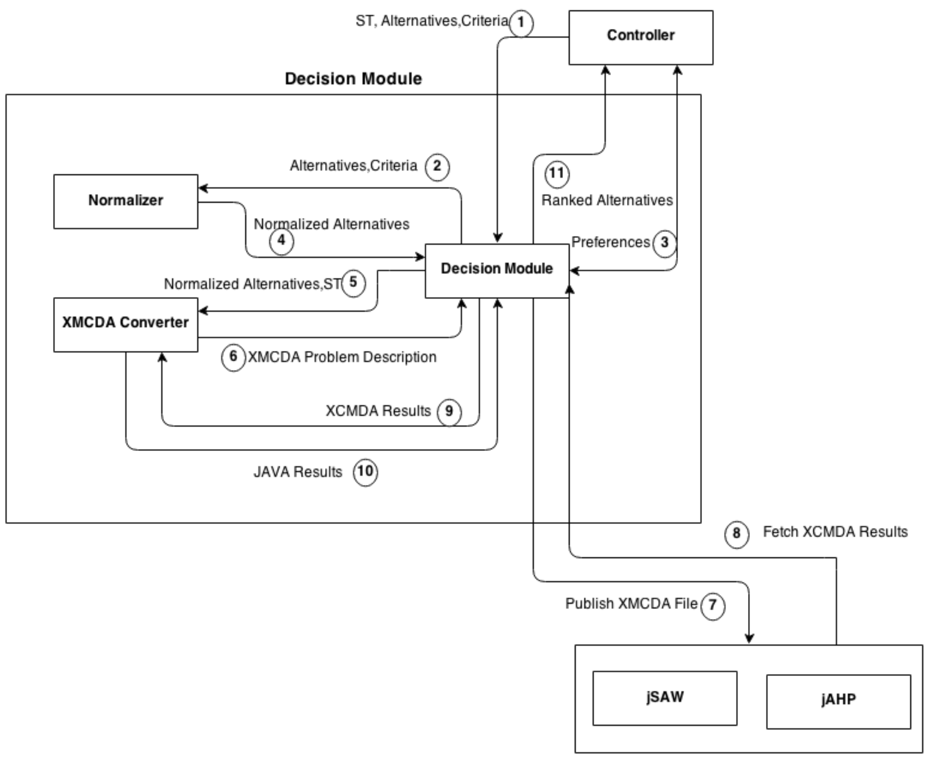


Fig.11 – CloudAid1 – Decision Module Architectural Flow

4.5.1 Data Normalization

The data normalization process consists in a series of mathematical calculations done by the Java class *Normalizer* to map the data (related to the defined criteria) on the current Service Template into the [0, 1] interval. Let's see an example, imagine we have a Service Template *S* with two alternatives A_1 and A_2, A_1 with attributes [*Price* = 40, *MemorySize = 1024*] and A_2 with [*Price* = 130, *MemorySize* = 4096]. Now suppose the criterions have the following weights: *Price* = 5 and *MemorySize* = 2. With this example we can see that the criteria *MemorySize*, despite having a lower weight than *Price*, will overwhelm the influence of *Price* on the calculations [7]. With the help of the *Normalizer* Java class we can avoid this type of situations.

In the previous example, the criteria fell inside the *numerical* (*quantitative*) type that is, they possess a specific value and are therefore quantifiable. However, this is not always the case; sometimes, the attributes to be normalized are non-numerical, also known as *qualitative* attributes. In this case, the Decision Module needs to stop the execution and ask some questions to the decision maker in order to assess preferred values and differences between them and the other possible values.

There are three types of attributes:

- *Non-numerical*
  - *Multi-Valued* – Necessary to choose the best value and define distances with other values.

    - *Binary* – Only two possible values, one is the best and the other is the worst.
- *Numerical*

The *Normalizer* class also asks the decision maker if there's a preferable value to the quantitative criteria being evaluated or if he/she prefers the simple maximization/minimization (also known as preference direction). In case he/she answers yes, the new value will be used instead of the maximum/minimum default value present in the alternatives.

We can identify 4 major steps in the normalization:

1. Check the attribute's type – Check if we're dealing with a *non-numerical binary*, *non-numerical* or *numerical* attribute.
2. Ask for preferred value – The Decision Maker can define a specific value to the criteria being evaluated.
3. Normalize the attribute – Once everything has been set the data can be mapped into the interval [0, 1] (0 is the worst and 1 the best).
4. Return the results – When the process is over, the results are stored on a new HashMap. This new version has the normalized values of the attribute and not the original ones.

This process is executed for every criteria defined by the decision maker.

4.5.2 XML Encoding of Multi-Criteria Decision Aid Data

XML Multi-Criteria Decision Analysis (XMCDA) is an XML based data standard developed by Decision Deck to describe Multi-Criteria Decision problems [23]. A Java library called J-XMCDA is also provided by the Decision Deck to help us create and manipulate information described in XMCDA. Every XMCDA related operation is performed by the Java class *XMCDAConverter* including operations related to the reading and writing of files.

4.5.3 Decision Methods

CloudAid1 uses external applications to solve the multi-criteria decision problem. To establish a communication link between the applications and the prototype, a file based approach was proposed. When in need of the external applications, a file is written on a directory monitored by them. The results obtained by the external applications are transmitted back to the prototype using the same approach but through a different directory. Fig.12 shows a visual representation of these transactions. To assist in this process, the *FileChecker* class was created. This class listens to events on the specified directories and calls the corresponding methods to handle them.

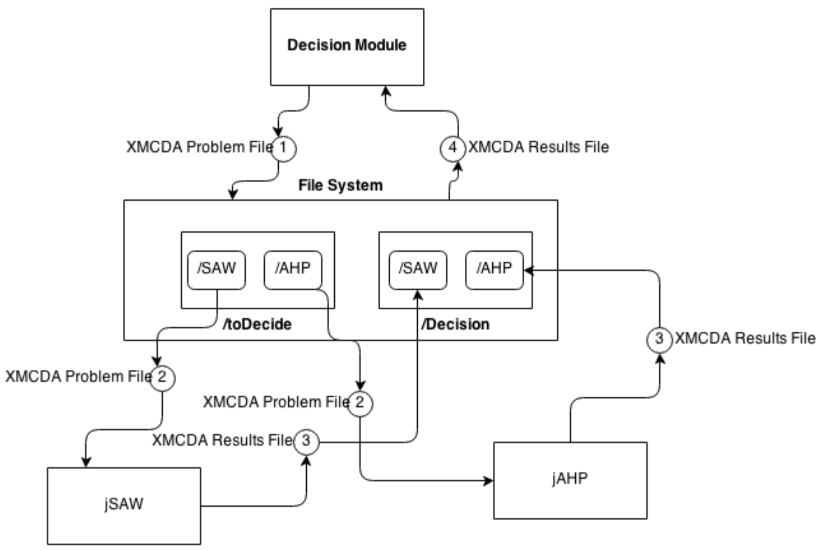


Fig.12 – CloudAid1 – File Communication

The CloudAid1 prototype supports two different *Multi-Criteria Decision Methods*: Simple Additive Weight (SAW) and the Analytic Hierarchic Process (AHP).

4.4.3.1 Simple Additive Weight

SAW is a simple multi-criteria decision method that relies on criterion weighting to determine the ranking of the alternatives [24]. Criterion weighting consists in the definition of an importance degree for every criterion in the Service Template. These values are at the core of the method and therefore are mandatory.

4.4.3.2 Analytic Hierarchic Process

Contrary to the SAW method, the AHP method is based on comparisons between data in order to extract the importance weights and perform the necessary calculations to derive the desired ranked list [25]. When I refer to comparisons, I speak of something like "Alternative A_1 is better than Alternative A_2" or "Criterion C_1 is more important than Criterion C_2".

4.6 Aggregation Module

Once the decision module finishes ranking the alternatives of each *Service Template*, we move to the Aggregation Module. This is the final step performed by the CloudAid1 prototype; after this step, a ranked list of admissible aggregated solutions that match the Decision Maker's requirements will be passed onto the Controller.

The purpose of the aggregation module is to exploit the ranked list of alternatives of each *Service Template* to create possible aggregations of services. It's necessary to point out that each aggregated solution is composed by one alternative from each *Service Template*. Let's consider an example: assume we have a *CSA* composed by three *Service Template*'s A, B, C and their corresponding ranked lists of alternatives RL_A, RL_B, RL_C. If we denote A_i has an alternative from RL_A with rank *i*, a possible aggregated solution would be: $A_1B_1C_1$. Fig.13 presents a visual representation of this example.

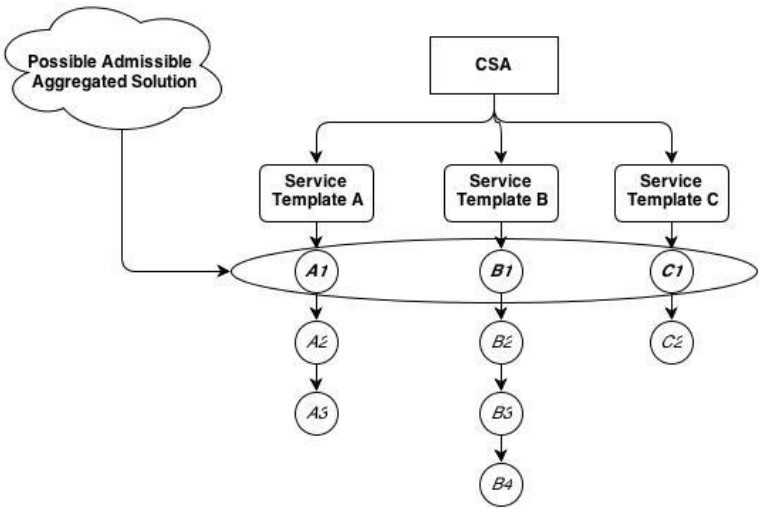


Fig.13 – Aggregation Module – Aggregated Solution Example (Z*i* are alternatives with rank *i* on list Z = {A,B,C})

The Java class responsible to compute these aggregated solutions is the class *Combinations*. It implements two algorithms that compute all the possible Admissible Aggregated Solutions based on the ranked lists of each *Service Template*. One version of the algorithm is able to deal with incomparability between alternatives (on a very limited scale) while the other doesn't consider this particular yet important aspect of multi-criteria decision problems. I'll cover this topic with further detail on section 4.5.2. Admissible Aggregated Solutions are solutions that match the Decision Maker global constraints or, in other words, requirements that were defined at the *CSA* level. Section 4.5.1 will provide a better description on this subject.

Once the possible admissible aggregated solutions have been found, the module performs one last step to optimize the results: *ranking* of the aggregated solutions. To get this ranked list of admissible solutions, each *ServiceTemplate* needs to either have their importance weight previously defined or, use the AHP method to indirectly "ask" them to the user. After, it uses once again the SAW method to rank the aggregated solutions.

4.5.1 Admissibility test

When considering every possible combination of alternatives from each *Service Template*, we must keep in mind that not every combination might be a valid one. For an aggregated solution to be considered valid, and therefore to be considered as possibility for the final solution list, needs to pass a series of tests first; these series of tests are performed by the Java class *AggChecker*.

These tests depend on the global constraints (requirements) set by the Decision Maker when creating his CSA. Consider an example where we have two alternatives for two different *Service Template*'s; a possible global requirement might be that every alternative needs to be compatible with Linux OS. If any alternative fails to support this type of operating system, the solutions fails the test and will be considered inadmissible. On CloudAid1, the only restriction supported is the global price; when the Decision

Maker defines a global Price requirement he's telling us that every aggregated solution's cost needs to be inferior or equal to some value set by him.

4.5.2 Aggregated Solutions Algorithms

As mentioned earlier, an aggregated solution is composed of by one alternative from each *Service Template* on the CSA. Following this line of thought, we can see that the number of possible aggregated solutions will grow exponentially depending on the number of *Service Templates* and corresponding alternatives.

To avoid this issue, the algorithms take advantage of the sorted ranked lists provided by the Decision Module. Let's assume we have three *Service Templates* A, B, C, with their corresponding sorted ranked lists of alternatives. Let's denote A_i as an alternative from *Service Template* A with rank *i*. With the knowledge that $A_i > A_k$, $i < k$ (alternative *a* is better than alternative *b* if *a* is ranked above *b*) we can say that any aggregated solution $A_iB_kC_l > A_oB_pC_q$ if $i,k,l < o,p,q$. In this case, $A_jB_kC_l$ dominates $A_oB_pC_q$ hence, there's no need test $A_oB_pC_q$ since its overall performance value will be lower.

Combining this knowledge with combinatory tree theory, the algorithms generate all possible combinations. They're based on the *Breadth-First Search algorithm* together with the notion of the *Branch and Bound* technique in order to transverse the tree of solutions and at the same minimize the computational weight of the problem. As mentioned earlier, there's two versions of the aggregation algorithm: *algorithm1* and *algorithm2*. In *algorithm1*, every time a new possible solution is discovered, it simply compare it with the already found admissible solutions to check for dominance. If there's dominance found between the node being tested and an already found solution, this node can be discarded along with its children. However, if no dominance is found, the node is added to the list of admissible solutions and its children discarded. Algorithm 1 shows the complete algorithm in a simplified way.

```
queue Q
add root to Q
while Q is not empty do
    node <- Q.pop()
    if node is Admissible then
        if node is not dominated by some admissibleSolution then
            add node to admissibleSolutions
        endif
    else
        for all possible children c of node do
            if((c ∉ Q) and (c ∉ tested)) then
          if c is not dominated by some admissibleSolution then
               add c to Q
           endif
             endif
        endfor
    endif
endwhile
```

Algorithm 1 – No incomparability

Algorithm2 is a bit different; by considering the possibility of incomparability between the alternatives (also known as a partial ordered list) it no longer assumes that when an admissible aggregated solution is found its children can be discarded. It checks for incomparability between the node and its children; if any children is incomparable with its father (which is a solution that passed the admissibility test), it'll be place into the *queue* for later testing. Algorithm 2 shows the full *algorithm2* in a simplified way.

```
queue Q
add root to Q
while Q is not empty do
   node <- Q.pop()
   add node to tested
   if node is admissible then
      if node is incomparable with all admissibleSolutions then
         add node to admissibleSolutions
         for all possible children c of node do
            if((c ∉ Q) and (c ∉ tested)) then
               if c is incomparable with node then
                  add c to Q
               endif
            endif
         endfor
      endif
   else
      for all possible children c of node do
         if((c ∉ Q) and (c ∉ tested)) then
            If c is not dominated by some admissibleSolution then
               add c to Q
            endif
         endif
      endfor
   endif
endwhile
```

Algorithm 2 – Incomparability support

Fig.14 shows the flow of execution of the Aggregation Module on an architectural level.

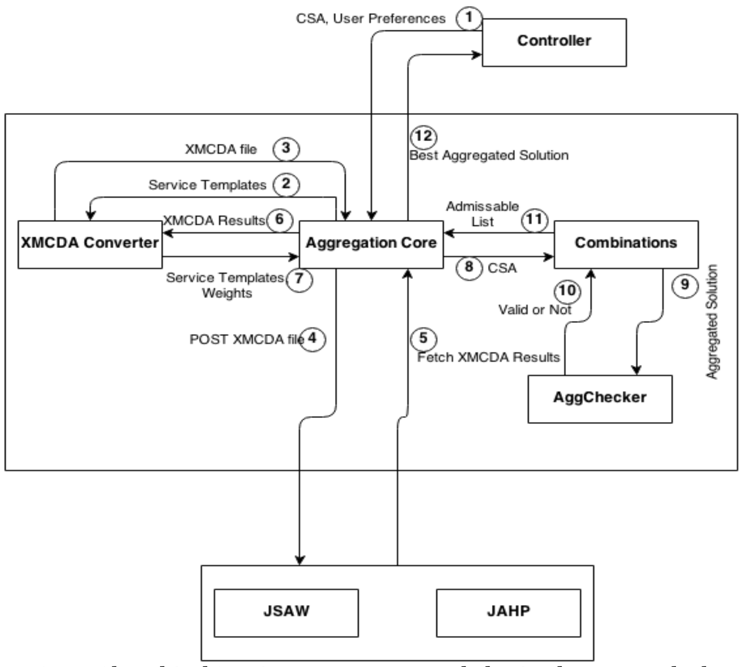


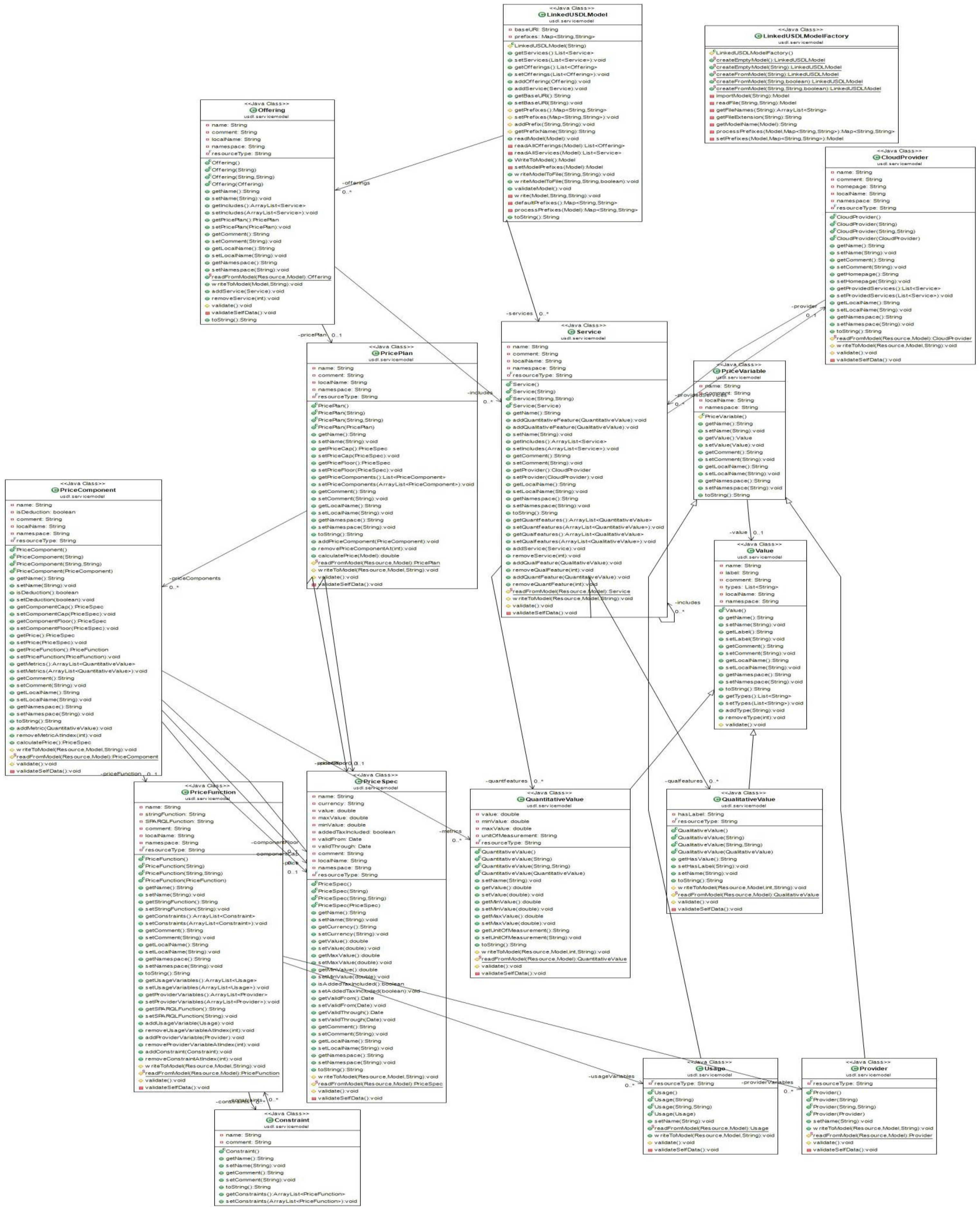
Fig.14 – CloudAid1 – Aggregation Module Architectural Flow

In a summarized way, these are the main steps of the Aggregation Module:

1. Get *Service Template*s weights – After receiving the CSA from the *Controller*, it's necessary to retrieve the importance weights from the *Service Templates* in order to compare and rank the aggregated solutions. There are two ways of getting these weights: the Decision Maker explicitly defines them when creating his CSA or, use JAHP and the Decision Make's preferences in order to calculate the *Service Template* weights.
2. Find admissible aggregated solutions – This is the main objective of the module: compute the possible aggregated solutions. Every time a new solution is found, the admissibility test is applied to verify if the solution meets the requirements necessary to enter the list of admissible solutions.
3. Find the best Aggregated Solution – When every solution has been found it ranks the solutions using the importance weights and the SAW method.

Appendix C

Linked USDL Pricing API – Class Diagram



Appendix D

Service modeling Use Cases assessment

Recurring Resource Pooling	
Query (Provider + Service Name)	**Results**
colt' 'IaaS/Flexible vCloud'	541000
VMWare' 'vCloud Hybrid'	**367000**
Cloudsigma' 'Cloud'	167000
iLand' 'Cloud'	154000
Orange Business Services' 'Flexible Computing'	106000
IBM' 'SmartCloud Enterprise'	81600
NTT Communications' 'Enterprise Cloud'	44600
Windstream' 'Public Cloud'	34900
GoGrid' 'Cloud Servers'	18800
Bluelock' 'Virtual Data Center'	16600
Singtel' 'PowerON Compute'	13200
Carrenza' 'cloud'	12400
CSC' 'BizCloud'	10900
Peak 10' 'Enterprise Cloud'	7130
LeaseWeb' 'Virtual Servers'	6200
ElasticHosts' 'Cloud Servers'	5650
GDS Services' 'HiA Cloud'	1310
Claranet' 'virtual datacenter'	1190

Table.18 CloudAid2 – Recurring Resource Pooling use cases

PrePaid Credit	
Query (Provider + Service Name)	**Results**
Microsoft' 'Azure Virtual Machines'	**4890000**
Internap' 'Public Cloud'	22000
LunaCloud' 'Cloud Servers'	8690
ElasticHosts' 'Cloud Servers'	5650
Gandi' 'Cloud Servers'	3730

Table.19 CloudAid2 – PrePaid use cases

PrePaid Subscription Credit	
Query (Provider + Service Name)	**Results**
Microsoft' 'Azure Virual Machines'	4890000
CloudSigma' 'Cloud'	**167000**
LunaCloud' 'Cloud Server'	42700
Internap' 'Public Cloud'	22000
GoGrid' 'Cloud Servers'	18800
ElasticHosts' 'Cloud Servers'	5650

Table.20 CloudAid2 – PrePaid Subscription Credit use cases

PrePaid VM	
Query (Provider + Service Name)	**Results**
IDC' 'Frontier CLoud'	**41000**
GoGrid' 'Cloud Servers'	18800
Peak 10' 'Enterprise Cloud'	7130

Table.21 CloudAid2 – PrePaid VM use cases

Spot Pricing	
Query (Provider + Service Name)	**Results**
Amazon' 'EC2'	**62300000**
CloudSigma' 'Cloud'	167000

Table.22 CloudAid2 – Spot pricing use cases

Reserved Instance	
Query (Provider + Service Name)	**Results**
Amazon' 'EC2'	**62300000**
Colt' 'IaaS/Flexible vCloud'	541000
Joyent' 'Compute'	107000
CSC' 'BizCloud'	10900
Fujitsu' 'IaaS Public'	1590

Table.23 CloudAid2 – Reserved Instance use cases

On-Demand	
Query (Provider + Service Name)	**Results**
Amazon' 'EC2'	**62300000**
Arsys' 'CloudBuilder'	2310000
AT T' 'Synaptic'	1800000
Google' 'Compute Engine'	562000
Colt' 'IaaS/Flexible vCloud'	541000
IIJ' 'GIO'	519000
Tier 3' 'Virtual Servers'	214000
Verizon Terremark' 'vCloud Hybrid'	187000
iLand' 'Cloud'	154000
Joyent' 'Compute'	107000
Orange Business Services' 'Flexible Computing'	106000
Hewlett Packard' 'Cloud Compute'	101000
IBM' 'SmartCloud Enterprise'	81600
ProfitBricks' 'Cloud'	72200
Microsft' 'Azure Virtual Machines'	62000
CenturyLink' 'Savvis' 'Cloud Servers'	48600
NTT Communications' 'Enterprise Cloud'	44600
LunaCloud' 'Cloud Server'	42700
Markley Group' 'Public Cloud'	37500
Singtel' 'PowerON'	36100
Windstream' 'Public Cloud'	34900
Internap' 'Public Cloud'	22000
Rackspace' 'CloudServers'	20100
GoGrid' 'Cloud Servers'	18800
Atlantic.net' 'Cloud Servers'	15800
Greenclouds' 'IaaS'	15100
Swisscom' 'Dynamic Server/DC'	14400
XO Communications' 'Compute'	13300
Carrenza' 'Cloud'	12400
CSC' 'BizCloud'	10900
Bluelock' 'Virtual DataCenter'	9340
IDC Frontier' 'Cloud'	8480
SoftLayer' 'CloudLayer'	7960
LeaseWeb' 'Virtual Servers'	6200
Fujitsu' 'IaaS Public'	1590
Dimension Data' 'Public CaaS'	1560
GDS Services' 'HiA Cloud'	1310
Claranet' 'Virtual DataCenter'	1190
Calligo' 'CloudCore'	194
Digital Ribbon' 'Hybrid Cloud'	29

Table.24 CloudAid2 – On Demand use cases

Recurring PrePaid VM	
Query (Provider + Service Name)	**Results**
Arsys' 'CloudBuilder'	**2310000**
Colt' 'IaaS/Flexible vCloud'	541000
IIJ' 'GIO'	519000
UK2 Group' 'Web Hosting'	210000
Hosting.com' 'Cloud Hosting'	94000
Numergy' 'Cloud'	57100
CenturyLink' 'Savvis' 'Cloud Servers'	48600
Connectria' 'Cloud Servers'	39200
Markley Group' 'Public Cloud'	37500
Windstream' 'Public Cloud'	34900
Bit Refinery' 'Cloud Hosting'	31800
XO Communications' 'Compute'	13300
IDC Frontier' 'Cloud'	8480
SoftLayer' 'CloudLayer'	7960
Peak 10' 'Enterprise Cloud'	7130
CloudCentral' 'Compute'	5360
Gandi' 'Cloud Servers'	3730
Cyberindo' 'IaaS'	891
KDDI' 'Virtual DC'	51

Table.25 CloudAid2 – Recurring PrePaid VM use cases

Appendix E

ServiceTemplate1's exclusive Requirements SPARQL Query

(**NOTE:** Every requirement on *ServiceTemplate1* is exclusive)

```
PREFIX  rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX  pf:   <http://jena.hpl.hp.com/ARQ/property#>
PREFIX  price: <http://www.linked-usdl.org/ns/usdl-price#>
PREFIX  xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX  rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX  spin: <http://spinrdf.org/spin#>
PREFIX  core: <http://www.linked-usdl.org/ns/usdl-core#>
PREFIX  cloudtaxonomy: <http://rdfs.genssiz.org/CloudTaxonomy#>
PREFIX  gr:   <http://purl.org/goodrelations/v1#>

SELECT REDUCED  ?offering
WHERE
  { { SELECT REDUCED  ?offering
      WHERE
        { ?offering rdf:type core:ServiceOffering .
          ?offering core:includes ?serv
            { ?serv gr:qualitativeProductOrServiceProperty ?f .
              ?f rdf:type cloudtaxonomy:Location .
              ?f rdfs:label ?value
              FILTER regex(?value, "tokyo", "i")
            }
          UNION
            { ?serv core:hasServiceModel ?model .
              ?model gr:qualitativeProductOrServiceProperty ?f .
              ?f rdf:type cloudtaxonomy:Location .
              ?f rdfs:label ?value
              FILTER regex(?value, "tokyo", "i")
            }
        }
    }
    { SELECT REDUCED  ?offering
      WHERE
        { ?offering rdf:type core:ServiceOffering .
          ?offering core:includes ?serv
            { ?serv gr:quantitativeProductOrServiceProperty cloudtaxonomy:CPUCores .
              ?f gr:hasValue ?value
            }
          UNION
            { ?serv gr:quantitativeProductOrServiceProperty ?f .
              ?f rdf:type cloudtaxonomy:CPUCores .
              ?f gr:hasValue ?value
            }
          UNION
            { ?serv core:hasServiceModel ?model .
              ?model gr:quantitativeProductOrServiceProperty cloudtaxonomy:CPUCores .
              ?f gr:hasValue ?value
            }
          UNION
            { ?serv core:hasServiceModel ?model .
              ?model gr:quantitativeProductOrServiceProperty ?f .
              ?f rdf:type cloudtaxonomy:CPUCores .
              ?f gr:hasValue ?value
            }
          UNION
            { ?serv core:hasServiceModel ?model .
              ?model gr:quantitativeProductOrServiceProperty ?f .
              ?f rdf:type cloudtaxonomy:CPUCores .
              ?f gr:hasMinValue ?value
            }
        }
    }
    { SELECT REDUCED  ?offering
      WHERE
        { ?offering rdf:type core:ServiceOffering .
          ?offering core:includes ?serv
            { ?serv gr:quantitativeProductOrServiceProperty cloudtaxonomy:DiskSize .
              ?f gr:hasValue ?value
              FILTER ( ?value >= 150.0 )
            }
          UNION
            { ?serv gr:quantitativeProductOrServiceProperty ?f .
              ?f rdf:type cloudtaxonomy:DiskSize .
              ?f gr:hasValue ?value
              FILTER ( ?value >= 150.0 )
            }
          UNION
            { ?serv core:hasServiceModel ?model .
              ?model gr:quantitativeProductOrServiceProperty cloudtaxonomy:DiskSize .
              ?f gr:hasValue ?value
              FILTER ( ?value >= 150.0 )
            }
          UNION
            { ?serv core:hasServiceModel ?model .
              ?model gr:quantitativeProductOrServiceProperty ?f .
              ?f rdf:type cloudtaxonomy:DiskSize .
              ?f gr:hasValue ?value
              FILTER ( ?value >= 150.0 )
            }
          UNION
            { ?serv core:hasServiceModel ?model .
              ?model gr:quantitativeProductOrServiceProperty ?f .
              ?f rdf:type cloudtaxonomy:DiskSize .
              ?f gr:hasMinValue ?value
              FILTER ( ?value >= 150.0 )
            }
        }
    }
    { SELECT REDUCED  ?offering
      WHERE
        { ?offering rdf:type core:ServiceOffering .
          ?offering core:includes ?serv
            { ?serv gr:qualitativeProductOrServiceProperty cloudtaxonomy:Performance
}
          UNION
            { ?serv gr:qualitativeProductOrServiceProperty ?f .
              ?f rdf:type cloudtaxonomy:Performance
            }
          UNION
            { ?serv core:hasServiceModel ?model .
              ?model gr:qualitativeProductOrServiceProperty cloudtaxonomy:Performance
            }
          UNION
            { ?serv core:hasServiceModel ?model .
              ?model gr:qualitativeProductOrServiceProperty ?f .
              ?f rdf:type cloudtaxonomy:Performance
            }
        }
    }
    { SELECT REDUCED  ?offering
      WHERE
        { ?offering rdf:type core:ServiceOffering .
          ?offering core:includes ?serv
            { ?serv gr:qualitativeProductOrServiceProperty ?f .
            ?f rdf:type cloudtaxonomy:Feature .
            ?f rdfs:label ?value
            FILTER regex(?value, "Virtual Machine", "i")
          }
        UNION
          { ?serv core:hasServiceModel ?model .
            ?model gr:qualitativeProductOrServiceProperty ?f .
            ?f rdf:type cloudtaxonomy:Feature .
            ?f rdfs:label ?value
            FILTER regex(?value, "Virtual Machine", "i")
          }
      }
  }
  { SELECT REDUCED  ?offering
    WHERE
      { ?offering rdf:type core:ServiceOffering .
        ?offering core:includes ?serv
          { ?serv gr:qualitativeProductOrServiceProperty cloudtaxonomy:Unix }
        UNION
          { ?serv gr:qualitativeProductOrServiceProperty ?f .
            ?f rdf:type cloudtaxonomy:Unix
          }
        UNION
          { ?serv core:hasServiceModel ?model .
            ?model gr:qualitativeProductOrServiceProperty cloudtaxonomy:Unix
          }
        UNION
          { ?serv core:hasServiceModel ?model .
            ?model gr:qualitativeProductOrServiceProperty ?f .
            ?f rdf:type cloudtaxonomy:Unix
          }
      }
  }
  { SELECT REDUCED  ?offering
    WHERE
      { ?offering rdf:type core:ServiceOffering .
        ?offering core:includes ?serv
          { ?serv gr:quantitativeProductOrServiceProperty cloudtaxonomy:MemorySize
.
            ?f gr:hasValue ?value
            FILTER ( ?value >= 4.0 )
          }
        UNION
          { ?serv gr:quantitativeProductOrServiceProperty ?f .
            ?f rdf:type cloudtaxonomy:MemorySize .
            ?f gr:hasValue ?value
            FILTER ( ?value >= 4.0 )
          }
        UNION
          { ?serv core:hasServiceModel ?model .
            ?model gr:quantitativeProductOrServiceProperty cloudtaxonomy:MemorySize
.
            ?f gr:hasValue ?value
            FILTER ( ?value >= 4.0 )
          }
        UNION
          { ?serv core:hasServiceModel ?model .
            ?model gr:quantitativeProductOrServiceProperty ?f .
            ?f rdf:type cloudtaxonomy:MemorySize .
            ?f gr:hasValue ?value
            FILTER ( ?value >= 4.0 )
          }
        UNION
          { ?serv core:hasServiceModel ?model .
            ?model gr:quantitativeProductOrServiceProperty ?f .
            ?f rdf:type cloudtaxonomy:MemorySize .
            ?f gr:hasMinValue ?value
            FILTER ( ?value >= 4.0 )
          }
      }
  }
  { SELECT REDUCED  ?offering
    WHERE
      { ?offering rdf:type core:ServiceOffering .
        ?offering core:includes ?serv
          { ?serv gr:quantitativeProductOrServiceProperty cloudtaxonomy:CPUSpeed .
            ?f gr:hasValue ?value
            FILTER ( ?value >= 2.5 )
          }
        UNION
          { ?serv gr:quantitativeProductOrServiceProperty ?f .
            ?f rdf:type cloudtaxonomy:CPUSpeed .
            ?f gr:hasValue ?value
            FILTER ( ?value >= 2.5 )
          }
        UNION
          { ?serv core:hasServiceModel ?model .
            ?model gr:quantitativeProductOrServiceProperty cloudtaxonomy:CPUSpeed .
            ?f gr:hasValue ?value
            FILTER ( ?value >= 2.5 )
          }
        UNION
          { ?serv core:hasServiceModel ?model .
            ?model gr:quantitativeProductOrServiceProperty ?f .
            ?f rdf:type cloudtaxonomy:CPUSpeed .
            ?f gr:hasValue ?value
            FILTER ( ?value >= 2.5 )
          }
        UNION
          { ?serv core:hasServiceModel ?model .
            ?model gr:quantitativeProductOrServiceProperty ?f .
            ?f rdf:type cloudtaxonomy:CPUSpeed .
            ?f gr:hasMinValue ?value
            FILTER ( ?value >= 2.5 )
          }
      }
  }
}
```

Appendix F

PROMETHEE I's *step1* message exchange

Web Service: *PrometheePreference-J-MCDA.py*	**Message Number:** 1
Message sent (decision problem XMCDA description)	

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <submitProblem>
      <alternatives xsi:type="xsd:string"><XMCDA xmlns="http://www.decision-deck.org/2009/XMCDA-
2.1.0">
  <alternatives xmlns="">
    <alternative id="Alt1" name="m1.xlarge_5020_TIME248205626989940linux-
AmazonSpotInstance_TIME248205626996782"/>
    <alternative id="Alt2" name="c3.2xlarge_5158_TIME248205635160313linux-
AmazonSpotInstance_TIME248205635168866"/>
    <alternative id="Alt3" name="c1.xlarge_5342_TIME248205645931555linux-
AmazonSpotInstance_TIME248205645941818"/>
    <alternative id="Alt4" name="m1.large_4974_TIME248205624872297linux-
AmazonSpotInstance_TIME248205624879139"/>
    <alternative id="Alt5" name="m2.2xlarge_5503_TIME248205656212301linux-
AmazonSpotInstance_TIME248205656219143"/>
    <alternative id="Alt6" name="m2.xlarge_5457_TIME248205654175481linux-
AmazonSpotInstance_TIME248205654182751"/>
  </alternatives>
</XMCDA></alternatives>
      <criteria xsi:type="xsd:string"><XMCDA xmlns="http://www.decision-deck.org/2009/XMCDA-2.1.0">
  <criteria xmlns="">
    <criterion id="Crit0" name="cloudtaxonomy:CPUCores">
      <scale mcdaConcept="PreferenceDirection">
        <quantitative>
          <preferenceDirection>max</preferenceDirection>
        </quantitative>
      </scale>
    </criterion>
    <criterion id="Crit1" name="cloudtaxonomy:CPUSpeed">
      <scale mcdaConcept="PreferenceDirection">
        <quantitative>
          <preferenceDirection>max</preferenceDirection>
        </quantitative>
      </scale>
    </criterion>
    <criterion id="Crit2" name="cloudtaxonomy:MemorySize">
      <scale mcdaConcept="PreferenceDirection">
        <quantitative>
          <preferenceDirection>max</preferenceDirection>
        </quantitative>
      </scale>
    </criterion>
    <criterion id="Crit3" name="cloudtaxonomy:DiskSize">
      <scale mcdaConcept="PreferenceDirection">
        <quantitative>
```

```
          <preferenceDirection>max</preferenceDirection>
        </quantitative>
      </scale>
    </criterion>
    <criterion id="Crit4" name="cloudtaxonomy:Performance">
      <scale mcdaConcept="PreferenceDirection">
        <quantitative>
          <preferenceDirection>max</preferenceDirection>
        </quantitative>
      </scale>
    </criterion>
    <criterion id="Crit5" name="price">
      <scale mcdaConcept="PreferenceDirection">
        <quantitative>
          <preferenceDirection>min</preferenceDirection>
        </quantitative>
      </scale>
      <thresholds>
        <threshold mcdaConcept="ind">
          <constant>
            <real>0.09204628</real>
          </constant>
        </threshold>
      </thresholds>
      <thresholds>
        <threshold mcdaConcept="pref">
          <constant>
            <real>0.18409257</real>
          </constant>
        </threshold>
      </thresholds>
      <thresholds>
        <threshold mcdaConcept="veto">
          <constant>
            <real>0.30682093</real>
          </constant>
        </threshold>
      </thresholds>
    </criterion>
  </criteria>
</XMCDA></criteria>
      <weights xsi:type="xsd:string"><XMCDA xmlns="http://www.decision-deck.org/2009/XMCDA-2.1.0">
  <criteriaValues mcdaConcept="Importance" name="significance" xmlns="">
    <criterionValue>
      <criterionID>Crit0</criterionID>
      <value>
        <real>0.12470588</real>
      </value>
    </criterionValue>
    <criterionValue>
      <criterionID>Crit1</criterionID>
      <value>
        <real>0.16705883</real>
      </value>
    </criterionValue>
    <criterionValue>
      <criterionID>Crit2</criterionID>
      <value>
        <real>0.16705883</real>
      </value>
    </criterionValue>
    <criterionValue>
      <criterionID>Crit3</criterionID>
      <value>
        <real>0.14588235</real>
      </value>
    </criterionValue>
    <criterionValue>
      <criterionID>Crit4</criterionID>
      <value>
        <real>0.1882353</real>
      </value>
    </criterionValue>
    <criterionValue>
      <criterionID>Crit5</criterionID>
      <value>
        <real>0.20705882</real>
      </value>
    </criterionValue>
  </criteriaValues>
</XMCDA></weights>
      <performances xsi:type="xsd:string"><XMCDA xmlns="http://www.decision-deck.org/2009/XMCDA-
2.1.0">
  <performanceTable xmlns="">
    <alternativePerformances>
      <alternativeID>Alt1</alternativeID>
      <performance>
        <criterionID>Crit4</criterionID>
        <value name="cloudtaxonomy:Performance">
          <real>1.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit2</criterionID>
        <value name="cloudtaxonomy:MemorySize">
          <real>0.29411766</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit0</criterionID>
        <value name="cloudtaxonomy:CPUCores">
          <real>0.33333334</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit1</criterionID>
        <value name="cloudtaxonomy:CPUSpeed">
          <real>0.16666667</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit5</criterionID>
        <value name="price">
          <real>0.62380683</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit3</criterionID>
        <value name="cloudtaxonomy:DiskSize">
          <real>1.0</real>
        </value>
      </performance>
    </alternativePerformances>
    <alternativePerformances>
      <alternativeID>Alt2</alternativeID>
      <performance>
        <criterionID>Crit4</criterionID>
        <value name="cloudtaxonomy:Performance">
          <real>1.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit2</criterionID>
        <value name="cloudtaxonomy:MemorySize">
          <real>0.29411766</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit0</criterionID>
        <value name="cloudtaxonomy:CPUCores">
          <real>1.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit1</criterionID>
        <value name="cloudtaxonomy:CPUSpeed">
          <real>1.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit5</criterionID>
        <value name="price">
          <real>0.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit3</criterionID>
        <value name="cloudtaxonomy:DiskSize">
          <real>0.0</real>
        </value>
      </performance>
    </alternativePerformances>
    <alternativePerformances>
      <alternativeID>Alt3</alternativeID>
      <performance>
        <criterionID>Crit4</criterionID>
        <value name="cloudtaxonomy:Performance">
          <real>1.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit2</criterionID>
        <value name="cloudtaxonomy:MemorySize">
          <real>0.0</real>
        </value>
      </performance>
      <performance>
```

```
        <criterionID>Crit0</criterionID>
        <value name="cloudtaxonomy:CPUCores">
          <real>1.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit1</criterionID>
        <value name="cloudtaxonomy:CPUSpeed">
          <real>0.6666667</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit5</criterionID>
        <value name="price">
          <real>0.2981471</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit3</criterionID>
        <value name="cloudtaxonomy:DiskSize">
          <real>1.0</real>
        </value>
      </performance>
    </alternativePerformances>
    <alternativePerformances>
      <alternativeID>Alt4</alternativeID>
      <performance>
        <criterionID>Crit4</criterionID>
        <value name="cloudtaxonomy:Performance">
          <real>0.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit2</criterionID>
        <value name="cloudtaxonomy:MemorySize">
          <real>0.018382354</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit0</criterionID>
        <value name="cloudtaxonomy:CPUCores">
          <real>0.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit1</criterionID>
        <value name="cloudtaxonomy:CPUSpeed">
          <real>0.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit5</criterionID>
        <value name="price">
          <real>1.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit3</criterionID>
        <value name="cloudtaxonomy:DiskSize">
        <real>0.4473684</real>
      </value>
    </performance>
  </alternativePerformances>
  <alternativePerformances>
    <alternativeID>Alt5</alternativeID>
    <performance>
      <criterionID>Crit4</criterionID>
      <value name="cloudtaxonomy:Performance">
        <real>0.0</real>
      </value>
    </performance>
    <performance>
      <criterionID>Crit2</criterionID>
      <value name="cloudtaxonomy:MemorySize">
        <real>1.0</real>
      </value>
    </performance>
    <performance>
      <criterionID>Crit0</criterionID>
      <value name="cloudtaxonomy:CPUCores">
        <real>0.33333334</real>
      </value>
    </performance>
    <performance>
      <criterionID>Crit1</criterionID>
      <value name="cloudtaxonomy:CPUSpeed">
        <real>0.375</real>
      </value>
    </performance>
    <performance>
      <criterionID>Crit5</criterionID>
      <value name="price">
        <real>0.20830993</real>
      </value>
    </performance>
    <performance>
      <criterionID>Crit3</criterionID>
      <value name="cloudtaxonomy:DiskSize">
        <real>0.45394737</real>
      </value>
    </performance>
  </alternativePerformances>
  <alternativePerformances>
    <alternativeID>Alt6</alternativeID>
    <performance>
      <criterionID>Crit4</criterionID>
      <value name="cloudtaxonomy:Performance">
        <real>0.0</real>
      </value>
    </performance>
    <performance>
      <criterionID>Crit2</criterionID>
      <value name="cloudtaxonomy:MemorySize">
        <real>0.37132353</real>
      </value>
    </performance>
    <performance>
      <criterionID>Crit0</criterionID>
```

```
        <value name="cloudtaxonomy:CPUCores">
          <real>0.0</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit1</criterionID>
        <value name="cloudtaxonomy:CPUSpeed">
          <real>0.104166664</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit5</criterionID>
        <value name="price">
          <real>0.7922515</real>
        </value>
      </performance>
      <performance>
        <criterionID>Crit3</criterionID>
        <value name="cloudtaxonomy:DiskSize">
          <real>0.17105263</real>
        </value>
      </performance>
    </alternativePerformances>
  </performanceTable>
</XMCDA></performances>
    </submitProblem>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Web Service:	**Message Number:** 2

Response from the *web service,* to Message Number 1

```
<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <submitProblemResponse>
      <message id="o7f95441ac850" xsi:type="xsd:string">The problem submission was
successful!</message>
      <ticket id="o123ebc8" xsi:type="xsd:string">grLoQlr2ZjIKyaye</ticket>
    </submitProblemResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Web Service: *PrometheePreference-J-MCDA.py*	**Message Number:** 3
Message to request the solution of the problem sent by Message Number:1	

```
<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <requestSolution>
      <ticket id="o123ebc8" xsi:type="xsd:string">grLoQlr2ZjIKyaye</ticket>
    </requestSolution >
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Web Service:	**Message Number:** 4
Response from the *web service,* to Message Number 3. Contains the preferences values which are required for the next *step* of the workflow.	

```
<?xml version="1.0" encoding="UTF-8"?><SOAP-ENV:Envelope xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:ZSI="http://www.zolera.com/schemas/ZSI/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Header/>
  <SOAP-ENV:Body>
    <requestSolutionResponse>
      <service-status id="o92c350" xsi:type="xsd:int">0</service-status>
      <ticket id="od2b4e0" xsi:type="xsd:string">KJn1rn2VqhZLPUdd</ticket>
      <messages id="od71f90" xsi:type="xsd:string"><?xml version="1.0" encoding="UTF-8"?>
<xmc:XMCDA xmlns:xmc="http://www.decision-deck.org/2009/XMCDA-2.1.0">
  <methodMessages>
    <logMessage>
      <text>Everything is ok.</text>
    </logMessage>
  </methodMessages>
</xmc:XMCDA></messages>
      <preference id="od7bc80" xsi:type="xsd:string"><?xml version="1.0" encoding="UTF-8"?>
<xmc:XMCDA xmlns:xmc="http://www.decision-deck.org/2009/XMCDA-2.1.0">
  <alternativesComparisons>
    <pairs>
      <pair>
        <initial>
          <alternativeID>Alt1</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt1</alternativeID>
        </terminal>
        <value>
          <real>0.0</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt1</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt2</alternativeID>
        </terminal>
        <value>
          <real>0.14588235</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt1</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt3</alternativeID>
        </terminal>
        <value>
          <real>0.16705883</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt1</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt4</alternativeID>
        </terminal>
        <value>
          <real>1.0</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt1</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt5</alternativeID>
        </terminal>
        <value>
          <real>0.33411765</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt1</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt6</alternativeID>
        </terminal>
        <value>
          <real>0.7977412</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt2</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt1</alternativeID>
        </terminal>
        <value>
          <real>0.49882352</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt2</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt2</alternativeID>
        </terminal>
        <value>
          <real>0.0</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt2</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt3</alternativeID>
        </terminal>
        <value>
          <real>0.54117644</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt2</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt4</alternativeID>
        </terminal>
        <value>
          <real>0.85411763</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt2</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt5</alternativeID>
        </terminal>
        <value>
          <real>0.6870588</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt2</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt6</alternativeID>
        </terminal>
        <value>
          <real>0.6870588</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt3</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt1</alternativeID>
        </terminal>
        <value>
          <real>0.49882352</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt3</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt2</alternativeID>
        </terminal>
        <value>
          <real>0.14588235</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt3</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt3</alternativeID>
        </terminal>
        <value>
          <real>0.0</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt3</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt4</alternativeID>
        </terminal>
        <value>
          <real>0.8329412</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt3</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt5</alternativeID>
        </terminal>
        <value>
          <real>0.6258824</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt3</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt6</alternativeID>
        </terminal>
        <value>
        <real>0.8329412</real>
      </value>
    </pair>
    <pair>
      <initial>
        <alternativeID>Alt4</alternativeID>
      </initial>
      <terminal>
        <alternativeID>Alt1</alternativeID>
      </terminal>
      <value>
        <real>0.0</real>
      </value>
    </pair>
    <pair>
      <initial>
        <alternativeID>Alt4</alternativeID>
      </initial>
      <terminal>
        <alternativeID>Alt2</alternativeID>
      </terminal>
      <value>
        <real>0.14588235</real>
      </value>
    </pair>
    <pair>
      <initial>
        <alternativeID>Alt4</alternativeID>
      </initial>
      <terminal>
        <alternativeID>Alt3</alternativeID>
      </terminal>
      <value>
        <real>0.16705883</real>
      </value>
    </pair>
    <pair>
      <initial>
        <alternativeID>Alt4</alternativeID>
      </initial>
      <terminal>
        <alternativeID>Alt4</alternativeID>
      </terminal>
      <value>
        <real>0.0</real>
      </value>
    </pair>
    <pair>
      <initial>
        <alternativeID>Alt4</alternativeID>
      </initial>
      <terminal>
        <alternativeID>Alt5</alternativeID>
      </terminal>
      <value>
        <real>0.0</real>
      </value>
    </pair>
    <pair>
```

```
        <initial>
          <alternativeID>Alt4</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt6</alternativeID>
        </terminal>
        <value>
          <real>0.14588235</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt5</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt1</alternativeID>
        </terminal>
        <value>
          <real>0.54117644</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt5</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt2</alternativeID>
        </terminal>
        <value>
          <real>0.3129412</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt5</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt3</alternativeID>
        </terminal>
        <value>
          <real>0.16705883</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt5</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt4</alternativeID>
        </terminal>
        <value>
          <real>0.8117647</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt5</alternativeID>
        </initial>
        <terminal>
```

```
          <alternativeID>Alt5</alternativeID>
        </terminal>
        <value>
          <real>0.0</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt5</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt6</alternativeID>
        </terminal>
        <value>
          <real>0.8117647</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt6</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt1</alternativeID>
        </terminal>
        <value>
          <real>0.16705883</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt6</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt2</alternativeID>
        </terminal>
        <value>
          <real>0.3129412</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt6</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt3</alternativeID>
        </terminal>
        <value>
          <real>0.16705883</real>
        </value>
      </pair>
      <pair>
        <initial>
          <alternativeID>Alt6</alternativeID>
        </initial>
        <terminal>
          <alternativeID>Alt4</alternativeID>
        </terminal>
        <value>
          <real>0.54117644</real>
          </value>
        </pair>
        <pair>
          <initial>
            <alternativeID>Alt6</alternativeID>
          </initial>
          <terminal>
            <alternativeID>Alt5</alternativeID>
          </terminal>
          <value>
            <real>0.0</real>
          </value>
        </pair>
        <pair>
          <initial>
            <alternativeID>Alt6</alternativeID>
          </initial>
          <terminal>
            <alternativeID>Alt6</alternativeID>
          </terminal>
          <value>
            <real>0.0</real>
          </value>
        </pair>
      </pairs>
    </alternativesComparisons>
</xmc:XMCDA></preference>
      </requestSolutionResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

References

[1] H. M. Alabool and A. K. Mahmood, “Trust -Based Service Selection in Public Cloud Computing Using Fuzzy Modified VIKOR Method,” vol. 7, no. 9, pp. 211–220, 2013.

[2] P. Costa, J. C. Lourenço, and M. Mira, “Evaluating Cloud Services using a Multiple Criteria Decision Analysis Approach,” no. Dm.

[3] Z. U. Rehman, O. K. Hussain, and F. K. Hussain, “Iaas Cloud Selection using MCDM Methods,” 2012 IEEE Ninth Int. Conf. E-bus. Eng., pp. 246–251, Sep. 2012.

[4] Z. U. Rehman, O. K. Hussain, and F. K. Hussain, “Multi-criteria IaaS Service Selection Based on QoS History,” 2013 IEEE 27th Int. Conf. Adv. Inf. Netw. Appl., pp. 1129–1135, Mar. 2013.

[5] M. Sun, T. Zang, X. Xu, and R. Wang, “Consumer-Centered Cloud Services Selection Using AHP,” 2013 Int. Conf. Serv. Sci., pp. 1–6, Apr. 2013.

[6] J. Araújo, “CloudAid.” [Online]. Available: https://github.com/jorgearj/CloudAid .

[7] J. Araújo, J. Cardoso, and C. Ferreira, “CloudAid,” 2013.

[8] “Internet Traffic Report.” [Online]. Available: http://www.internettrafficreport.com/.

[9] J. Araujo, J. Cardoso, C. F. da Silva, and P. Melo, “Cloud Services Aggregation,” Trans. Serv. Comput., 2014.

[10] “IBM News room - 2013-10-24 IBM Study Reveals Businesses Using Cloud Computing for Competitive Advantage Can Generate Double Revenue and Profit Compared to their Peers - United States,” 24-Oct-2013. [Online]. Available: http://www-03.ibm.com/press/us/en/pressrelease/42304.wss.

[11] “SPIN - SPARQL Inferencing Notation.” [Online]. Available: http://spinrdf.org/.

[12] “Composing the Semantic Web: Introducing SPIN: the SPARQL Inferencing Notation.” [Online]. Available: http://composing-the-semantic-web.blogspot.pt/2009/01/introducing-spin-sparql-inferencing.html.

[13] “Composing the Semantic Web: Understanding SPIN Functions.” [Online]. Available: http://composing-the-semantic-web.blogspot.pt/2009/01/understanding-spin-functions.html.

[14] D. Bouyssou, “Outranking Methods,” pp. 1–12.

[15] J. Figueira, S. Greco, and M. Ehrgott, Multiple CriteriaDecision Analysis: State of the Art Surveys. Springer,2005.

[16] C. Zopounidis and M. Doumpos, Multicriteria Decision Aid Classification Methods. 2004.

[17] "LinkedUSDL core - github." [Online]. Available: https://github.com/linked-usdl/usdl-core.

[18] "Project Management." [Online]. Available: http://en.wikipedia.org/wiki/Project_management.

[19] "Propostas de Estágio 2013/2014 - CloudAid2." [Online]. Available: http://estagios.dei.uc.pt/cursos/mei/2013-2014/propostas-de-estagio/?id=1412.

[20] "Modified Waterfall Model." [Online]. Available: http://en.wikipedia.org/wiki/Modified_waterfall_models.

[21] F. Consortium, "FMC - Books about the Fundamental Modeling Concepts."

[22] "Apache Jena - Home." [Online]. Available: http://jena.apache.org/.

[23] "About XMCDA — XMCDA." [Online]. Available: http://www.decision-deck.org/xmcda/about.html.

[24] "Weighted sum." [Online]. Available: http://en.wikipedia.org/wiki/Weighted_sum_model.

[25] "The Analytic Hierarchy Process." [Online]. Available: http://www.dii.unisi.it/~mocenni/Note_AHP.pdf.

[26] "JavaScript Object Notation." [Online]. Available: http://en.wikipedia.org/wiki/JSON.

[27] "JSON ." [Online]. Available: http://www.w3schools.com/json/.

[28] "The Decision Deck project — Decision Deck." [Online]. Available: http://www.decision-deck.org/project/.

[29] "The Decision-Deck Project - Developing a Multiple Criteria Decision Analysis Software Platform." [Online]. Available: http://ercim-news.ercim.eu/en72/rd/the-decision-deck-project-developing-a-multiple-criteria-decision-analysis-software-platform.

[30] "Design, execute and share MCDA methods — diviz." [Online]. Available: http://www.decision-deck.org/diviz/.

[31] “Sample workflows — diviz.” [Online]. Available: http://www.decision-deck.org/diviz/workflows.html.

[32] “Balsamiq. Rapid, effective and fun wireframing software.” [Online]. Available: http://balsamiq.com/.

[33] “What is Functional Testing? - Definition from Techopedia.” [Online]. Available: http://www.techopedia.com/definition/19509/functional-testing.

[34] “Functional Testing.” [Online]. Available: http://en.wikipedia.org/wiki/Functional_testing.

[35] “MoSCoW Method for Requirements Prioritization ~ Business Analysis.” [Online]. Available: http://www.businessanalysis.in/2013/06/moscow-method-for-requirements.html.

[36] “Supplementary Specification / FURPS | Refslund.” [Online]. Available: http://refslund.me/cs/systemdevelopment/miscellaneous/specification.

[37] P. Wang, Y. Mu, W. Susilo, and J. Yan, “Privacy preserving protocol for service aggregation in cloud computing,” Software: Practice and Experience, vol. 42, no. 4,pp. 467–483, 2012.

[38] J. Cardoso and A. Sheth, “Semantic e-workflow composition,” Journal of Intelligent Information Systems (JIIS),vol. 21, no. 3, pp. 191–225, 2003.

[39] T. Erl, Service-Oriented Architecture: Concepts, Technology,and Design. Prentice Hall PTR, 2005.

[40] “Posts about royce waterfall model on Galib’s virtual identity.” [Online]. Available: https://imgalib.wordpress.com/tag/royce-waterfall-model/. [Accessed: 25-Jan-2014].

[41] E. Sirin, J. Hendler, and B. Parsia, “Semi-automaticcomposition of web services using semantic descriptions,” in Web Services: Modeling, Architecture and Infrastructure Workshop, 2002, pp. 17–24.

[42] D. Liu, N. Li, C. Pedrinaci, J. Kopeck ́ y, M. Maleshkova,and J. Domingue, “An approach to construct dynamic service mashups using lightweight semantics,” in Current Trends in Web Engineering, ser. LNCS. Springer,2012, vol. 7059, pp. 13–24.

[43] “Cloud Computing: Cloud Pricing Models – Part 4 - CloudTweaks.com: Cloud Information.” [Online]. Available: http://cloudtweaks.com/2012/04/cloud-computing-cloud-pricing-models-part-4/.

[44] “The Cloud Computing Blog: Cloud Pricing Models.” [Online]. Available: http://cloud-articles.blogspot.pt/2012/10/cloud-pricing-models.html.

[45] “The present and future of cloud pricing models - Thoughts on Cloud.” [Online]. Available: http://thoughtsoncloud.com/2013/06/present-future-cloud-pricing-models/.

[46] “Cloud Pricing Models that Work.” [Online]. Available: http://newswire.telecomramblings.com/2013/04/cloud-pricing-models-that-work/.

[47] M. Al-Roomi, S. Al-Ebrahim, S. Buqrais, and I. Ahmad, “Cloud Computing Pricing Models: A Survey,” Int. J. Grid Distrib. Comput., vol. 6, no. 5, pp. 93–106, Oct. 2013.

[48] T. H. E. C. Pricing, “THE CLOUD PRICING CODEX,” no. December, 2013.

[49] “451 Research Home.” [Online]. Available: https://451research.com/.

[50] D. Barrigas and J. Araujo, “Linked USDL Pricing API,” 2014. [Online]. Available: https://github.com/jorgearj/USDLPricing_API.

[51] D. Barrigas and J. Araújo, “Linked USDL Pricing API Tar,” 2014. [Online]. Available: https://jorgearj.github.io/USDLPricing_API/.

[52] D. Barrigas, “CloudAid2,” 2014. [Online]. Available: https://github.com/dguedesb/CloudAid2.

[53] D. Barrigas, “ServiceGatherer,” 2014. [Online]. Available: https://github.com/dguedesb/CloudAid2-ServiceGatherer.

[54] Wikipedia, “Adjacency List.” [Online]. Available: http://en.wikipedia.org/wiki/Adjacency_list.

[55] S. J. C. Vol, “FINDING ALL THE ELEMENTARY,” vol. 4, no. 1, pp. 77–84, 1975.

[1]–[55]