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

Diploma Thesis No. 3728

# Utility-based Analysis of Evolving Cloud Application Topologies

Florian Hannes Frech

**Course of Study:** Computer Science

**Examiner:** Prof. Dr. Dr. h. c. Frank Leymann
**Supervisor:** Dipl.-Inf. Santiago Gómez Sáez
**Commenced:** July 13, 2015
**Completed:** January 19, 2015

**CR-Classification:** C.2.4, C.4, D.2.8

## Abstract

Today, Cloud consumers have access to a wide spectrum of Cloud offerings. On the one hand, this is a profitable situation, since there is a larger spectrum of possibilities to migrate the application to the Cloud. However, on the other hand, consumers face the challenge of selecting the offering that promises the highest benefit. The challenge even grows larger when taking into consideration the possibility to distribute the application components. Approaches like TOSCA support developers in in the portable description of composite Cloud applications and tools provide the selection of the most cost-effective Cloud offerings that fulfill a set of requirements. Besides that, there is a lack of decision support that goes beyond the mere look on Cloud offerings' technical data and operational costs. Developers should be supported with the necessary mechanisms and tools towards evaluating and analysing the trade-off between different aspects and involve different stakeholders' interests. Moreover not only isolated Cloud offerings should be evaluated, but also the outcome of applications under a distributed deployment with respect to evolving workloads. Since utility functions facilitate the analysis of users' satisfaction, i.e. performing the trade-off between different aspects, this thesis presents a concept that uses utility functions in order to evaluate applications' topologies. This concept makes application distribution alternatives comparable. Based on this concept, a utility calculation framework is specified. The framework provides support for creating customized utility functions, calculates the utility of alternative topologies, and offers decision support. Furthermore, this thesis presents a prototypical implementation of the utility calculation framework, which is further evaluated using a realistic application and data.

# Contents

# List of Figures

# List of Tables

# List of Listings

# 1 Introduction

This chapter includes the motivation and problem statement, research challenges, the acronyms which occur in the thesis and the thesis' structure.

## 1.1 Motivation and Problem Statement

Cloud Computing has rapidly gaining popularity, the number of Cloud services has increased and a wide spectrum of Cloud offerings is available. Business expenses are the main driver for this development. Cloud Computing scores in particular thanks to its benefits of reduced infrastructural cost and dynamic access to computational resources [ASL13]. Today, a number of Cloud providers offer a range of various Cloud services with individual performance attributes and pricing models. As a result, the decision for a certain Cloud offering has an impact on both, the operational expenses and the information system's performance.

The study in [GSAGL14] separates an application topology into application specific components and application independent sub-topologies. [GSAGL14] argues that there are multiple deployment alternatives for the application components, hence the application topology is only an element from the set of application topology alternatives. When going from migrations of type III (Migrate the whole software stack) to type IV (Cloudify) [ABLt13], the exploitation of the potential in Cloud Computing increases. However, the issue of selecting the optimal deployment gains in complexity. Heterogeneous distributed components are combined to provide applications' functionality [BBKL14]. Each component's performance is influenced by the specific deployment and the application's performance comes from the interplay of these components. When questioning the fulfilment of requirements defined for the application, these dependencies must be suggested. It is essential to point out that the application's workload has a fundamental role to play. The determination of the optimal distribution requires the consideration of workload and is made difficult by the fact that the application workload may oscillate over time [Sáe14].

Certain standards and tools support developers in designing topologies, choosing Cloud offerings and deploying applications in the Cloud. The TOSCA standard allows for the standardized and portable description of composite Cloud applications and their management [BBKL14]. Other standards for describing applications and there topologies are the *Generalized Topology Language* (GENTL) [ARSL14] and Blueprints [PvdH11]. The TOSCA modelling tool *Winery* supports the graph-bases modelling of application topologies. The main components are the *Element Manager* and the *Topology Modeller* [KBBL13]. Finally, a TOSCA

Container enables the automation of provisioning, management and termination of applications based on these descriptions [BBKL14, KBBL13]. The *Nefolog* system offers a knowledge base and a decision support system [XA13]. The knowledge base contains different Cloud providers, offerings and various configurations. A REST interface allows for the candidate search and the cost calculation. *MiDSuS* utilizes the *Nefolog* system and provides its functionalities through a graphical user interface. The GENTL Environment [ARSL14] also utilizes the *Nefolog* system. The environment provides the visualization of topology data, an annotation model and a transformation application, which allows for the import of TOSCA and Blueprint models [ARSL14].

## 1.2 Research Challenges

The approach in [AGSLW14] provides a method to explore the space of alternative deployments for application specific components. Furthermore, it suggests utility functions for the evaluation of these alternatives. Utility functions have the ability to consider multiple dimensions [STFG08]. This allows for the evaluation of topologies taking into account various requirements and interests from different stakeholders.The first research challenges are to examine approaches for using utility theory in software architectures and derive the concept and specification for enabling the utility based analysis of Cloud application distributions.

Besides the tools and standards mentioned above, there is a lack of decision support that goes beyond the mere look on Cloud offerings' technical data and operational costs. Others aspects can be e.g. the fulfilment of specified functional and non-functional requirements under the applications' workload, the end-users' satisfaction and the generated revenue. Decision makers should be supported in the formulation of utility functions which break all relevant aspects into one axis. A decision support tool should also enable the calculation of utility of topologies based on these utility functions. Therefore, further research challenges are the definition of the architecture, the specification and the design of a loosely coupled, RESTful framework which closes the gap in decision support.

The last research challenges are the prototypical implementation of the framework and the necessary evaluation whether the implementation provides the intended decision support.

## 1.3 Definitions and Conventions

This section covers abbreviations occurring in this work and definitions which are necessary to understand this thesis.

#### Definitions

#### List of Abbreviations

| | |
|---|---|
| **API** | Application Programming Interface |
| **AWS** | Amazon Web Services |
| **Amazon EC2** | Amazon Elastic Compute Cloud |
| **B2B** | Business-to-Business |
| **B2C** | Business-to-Consumer |
| **BPEL** | WS-Business Process Execution Language |
| **BPEL4WS** | BPEL for Web Services |
| **DBaaS** | Database-as-a-Service |
| **DMS** | Database Management System |
| **FK** | Foreign Key |
| **GENTL** | Generalized Topology Language |
| **GUI** | Graphical User Interface |
| **HTTP** | Hypertext Transfer Protocol |
| **HTML** | Hypertext Markup Language |
| **IaaS** | Infrastructure-as-a-Service |
| **JAXB** | Java Architecture for XML Binding |
| **JAX-RS** | Java API for RESTful Web Services |
| **Java EE** | Java Platform, Enterprise Edition |
| **JSON** | JavaScript Object Notation - Data-interchange format |
| **KPI** | Key Performance Indicator |
| **NIST** | National Institute of Standards and Technology |
| **PaaS** | Platform-as-a-Service |
| **PHP** | PHP: Hypertext Preprocessor - Server side scripting language |
| **PK** | Private Key |
| **POJO** | Plain Old Java Object |
| **QoS** | Quality of Service |
| **REST** | Representational state transfer - software architectural style |
| **RESTful** | Characteristic that expresses a system's conformity to the constraints of REST |
| **RPN** | Reverse Polish Notation |
| **RDBMS** | Relational Database Management System |
| **SaaS** | Software-as-a-Service |
| **SLA** | Service Level Agreement |
| **SOAP** | (originally) Simple Object Access Protocol |
| **SOA** | Service-oriented Architecture |
| **SOC** | Service-oriented Computing |
| **SQL** | Structured Query Language |
| **TOSCA** | Topology and Orchestration Specification for Cloud Applications |
| **UDDI** | Universal Description, Discovery and Integration |
| **URI** | Uniform Resource Identifier |
| **URL** | Uniform Resource Locator |
| **UUID** | Universally Unique Identifier |
| **WS** | Web Services |
| **WSDL** | Web Services Description Language |
| **XML** | Extensible Markup Language |

## 1.4 Outline

The document's structure is described below:

**Chapter 1 - Introduction** Includes the motivation and problem statement, research challenges, the acronyms which occur in the thesis and the thesis' structure.

**Chapter 2 - Fundamentals** Concepts and technologies in the thesis' sphere.

**Chapter 3 - Related Works** Existing approaches for using utility theory in software architectures and a comparative analysis.

**Chapter 4 - Concept and Specification** A Concept which enables utility theory based analysis of Cloud application distributions. Furthermore, requirements, use cases and system overview for the RESTful framework.

**Chapter 5 - Design** Description of the RESTful framework's design and discussions about attempts and utilized algorithms.

**Chapter 6 - Implementation** This chapter contains a description of the prototypical implementation of the RESTful framework.

**Chapter 7 - Evaluation** Evaluation of concept and implementation based on a decision problem.

**Chapter 8 - Outcome and Future Work** In the final chapter, a conclusion of the work is given and a outlook for further development is presented.

# 2 Fundamentals

This chapter summarizes concepts, technologies and approaches this thesis relies on.

## 2.1 Cloud Computing

Cloud Computing is a model for enabling access to a shared pool of computing resources [MG11]. These resources are delivered on-demand to the consumer as services, comparable to the delivery of water and gas [GVB11]. [JM12]. Cloud Computing is characterized by (1) the provision of computing capabilities on demand, (2) the access to its capabilities over network, (3) the pooling of different physical and virtual resources to serve multiple consumers dynamically according to their demands, (4) the consumers' possibility to elasticity provision, release and scale Cloud capabilities in a rapid manner and (5) the transparency generated by the monitoring, controlling and reporting of resource usage [MG11].

Strong arguments for Cloud Computing are decreased costs and higher flexibility. The sharing of resources and costs among many consumers allow the better utilization of infrastructures [JM12]. Cloud Computing improves the use of distributed resources and solves scalability problems in distributed computing [RCL09]. As a result infrastructural costs decrease while flexibility increases [ASL13].

Another advantage for the consumers of computing resources arises from the available on demand and corresponding payment models. The risks in long-term resource planning are minimized. Consumers increase or even decrease their resources with respect to their demand on a short-term basis [AFG+10]. Using-based payment schemes ensures that consumers pay as their demand increase or decrease [RCL09].

### 2.1.1 Actors

Cloud providers make their services accessible through internet based interfaces [VRMCL08a]. Providers serve multiple consumers based on a multi-tenant model [MG11]. Furthermore, consultings support consumers to select and implement relevant services. [LRBK10]

[LRBK10] distinguishes different kind of Cloud providers: (1) Infrastructure providers supply the scalable computing and storage services needed to run applications within the cloud, (2) service providers develop applications that are offered to consumers and access hardware and infrastructure of infrastructure providers, (3) platform providers provide an environment within which cloud applications can be deployed and (4) aggregators offering

services which are created by the combination of already existing services. Aggregators are customers and providers at the same time.

### 2.1.2 Service Models

The most prominent service models in cloud computing are *Software as a Service* (SaaS), *Platform as a Service* (PaaS) and textitInfrastructure as a Service (IaaS). Furthermore, *Database as a Service* (DBaaS) will be introduced in this section.

**Infrastructure as a Service** IaaS offers services like storage, CPU and memory. These resources are delivered as storage and virtual machines of different size (combinations of CPU, memory and local storage) [LWW+10]. Consumers have no control over the underlying infrastructure but have control over operating systems and storage [MG11]. IaaS consumers profit especially from flexibility and the usage-based payments [RCL09].

**Platform as a Service** PaaS provides developers a platform for the deployment of applications based on programming languages, libraries, services and tools supported by the provider [MG11]. Developers can develop, test and deploy their applications on these platforms [RCL09] without operating and managing the underlying infrastructure [MG11]. PaaS providers utilizes IaaS offers by requesting virtual machines and storage and deploying application containers in the virtual machines [LWW+10].

**Software as a Service** SaaS uses common resources and a single instance of an application to server multiple consumers simultaneous [RCL09] (economics-of-scale principle [LS10]). The offered application is hosted run and administrated in large web data centers and provided as a service [LWW+10]. Consumers use the application remotely over the internet without controlling the underlying infrastructure. Consumers have at most control about user-specific application settings [MG11].

**Database as a Service** Early SaaS applications were build on relational database technologies. The gap between the applications' functional requirements (e.g. multi-tendency) and the limited suitable of classic database systems leads to complex infrastructures and extensive maintenance [LS10]. The need for easy to use persistence layer with classic database features results in Data Base as a Service (DBaaS) offerings [LS10]. The structure of Cloud databases is still complex, since Cloud databases hold the data on different data centres while consumers are provided with a easy and complete access over services. [AS13]

### 2.1.3 Cloud Models

Cloud models can be defined by four types: (1) *Private Cloud*, (2) *Public Cloud*, (3) *Community Cloud* and (4) *Hybrid Cloud* [MG11].

**Private Cloud** A *private Cloud* is provisioned only for the use by consumers of a single organisation [MG11]. This model offers the highest degree of control over performance, reliability and security [Sin15].

**Public Cloud** In a *public Cloud* services are offered for open use by general public [MG11]. Consumers' benefit of the public Cloud model is the elimination of initial investments in the underlying Cloud infrastructure [Sin15]. The downside of this model is the deficient in fine grained control over data, network and security settings [Sin15].

**Hybrid Cloud** *Hybrid Clouds* are mixtures of the previous introduced models [MG11]. Some services can run in a *Private Cloud* while other services run in a *Public Cloud*. *Hybrid Clouds* can provide a more fine-grained control than *public Clouds*, while still exploit on-demand service expansion and payment models [Sin15].

**Community Cloud** *Community Cloud* consumers belonging to organisations with shared concerns [MG11]. Computing resources are shared within a community [Sin15].

### 2.1.4 Migration Types

[ABLt13] describes four migration types for applications: (1) *Type I* - replace one or more component with Cloud offerings. Such a migration type could require configurations, rewriting and adaptations to cope with incompatibilities. *Type II* - migrate application functionality to the Cloud. One or more application layers are migrated in order to provide selected functionality from the Cloud. *Type III* - migrate the software stack of the application to the Cloud, often by VMs. *Type IV* - the application functionality is served by a composition of services running on the Cloud.

### 2.1.5 Payment Models

The common payment model is the pay-per-use model [VRMCL08b]. Users have only to pay for what they use. The disadvantage is a lack of acceptance when users want to control their budget [PZJ14]. In contrast, subscription pricing is billing a lumpsum payment on a recurring basis and will not impose additional costs per unit [PZJ14]. A two-part tariff combines recurring lumpsum payments with additional costs per unit [PZJ14].

## 2.2 Cloud Application Topologies

Different cloud providers offering similar services in different manners. As a result developers are often locked to a specific platform environment because of the expense of migrate applications to other platforms [BSW14]. Enterprise applications are often composed of multiple components. The components functionalities are orchestrated into more complex applications [BBKL14]. In order to create portable cloud applications, developers need a machine-readable format for modelling application topologies [BBKL14]. The management of the components inside topologies (e.g. deployment, configuration, communication to other components) should also be covered by an topology language [BBKL14].

### 2.2.1 Specifications and Existing Approaches

[ARSL14] identifies a set of common fundamental concepts in different topology languages. All approaches use a graph-based view of application topologies. Components are modelled as nodes and connectors as edges. Components can be assembled into groups, marking subgraphs. Furthermore, components and connectors can be described by attributes. [AGSLW14] notes that different approaches describe application topologies, middleware components and cloud offerings involved using the typed graph model.

#### GENTL

The GENeralized Topology Language (GENTL) is a extensible and technology-independent language based on four concepts common in topology languages [XA13]. (1) Topology model - The graph model represents components (software artefacts and services) by nodes and connectors (relationships between components, e.g. hosted on, connected to) by edges. (2) Groups - Components and connectors can form sub-topologies. These sub-topologies are represented by groups. (3) Attributes - Attributes store related informations for components, connectors, groups or even the whole topology. Simple (name-value) and composed attributes are supported. (4) Annotations - GENTL allows the annotation of components, groups and the whole topology. [XA13] utilizes annotations of the whole topology to store projected costs (cost-annotated topologies).

A GENTL model begins with a topology element. The topology element composes components, groups and topology attributes. Thereby groups enables the organisation in subgraphs. Connectors represents relationships between a source- and a target-component in topologies. Attributes are either simple attributes or composite attributes which allows nested attribute composition. Topology attributes capture informations regarding the whole topology, while attributes of components, groups and connectors capture the informations in their respective spher [XA13].

#### TOSCA

The purpose of the "OASIS Topology and Orchestration Specification for Cloud Applications" (TOSCA) are portable cloud applications and the automation of their deployment and management [BSW14]. [BBKL14], [BSW14] explains how TOSCA addresses the following main challenges in cloud computing: (1) Automated management, (2) portability of applications and (3) interoperability and reusability of application components. Since, in general the creator of a IT solution has the knowledge how to manage the solution, TOSCA faces challenge (1) by management plans. Management plans are workflows included in the topology. They are portable and executed automated and thereby providing automated self-service management. Basically challenge (2) results from vendor lock-ins. TOSCA formalizes the application topology and its management. The description of topologies is standardized and management plans rely on portable workflow languages. TOSCA defines components in a reusable and interoperable manner. These components can be combined vendor-independent and thereby simplifying challenge (3).

As mentioned previously, TOSCA compromises two main concepts: (1) Application topologies and (2) management plans. TOSCA is using a XML-based modelling language to describe (1) as typed topology graphs and workflows to describe (2) [BBKL14].

#### Winery

Winery is a graph-based modelling tool for TOSCA-based Cloud applications. The environment provides the a graph based modelling of topologies and the definition of components and relationship types [KBBL13]. Winery contains two GUI-based components (HTML5-based). First, the Topology Modeler allows for the convenient modelling of application topologies with the graphical visualization of elements and their combinations. Second, the Element Manager can be used to provide and configure node types and relationship types [KBBL13]. Both components utilizes the Repository component which is responsible to store and provide data [KBBL13].

## 2.3 Application Workload

[GSAGL14] defines application workload a "*description of a set of business transactions which are probabilistically distributed for a time interval, have an impact on the application state and define the behavioural characteristics of its corresponding users*". The application workload profile is compromised of workload samples; each sample includes a usage profile, a workload mix and a behavioural model [GSAGL14]. The usage profile describes the end user in terms of the evolution of arrival rates and the specification of their requests, the workload mix is a set of transactions that can be preformed on the application and the behavioural model defines the distribution of the workload mixes transactions over time based e.g. on popularity, probability of occurrence, etc. [GSAGL14].

The performance of an application depends on the distribution of the application topology [GSALS14]. Specified performance requirements are affected [GSAGL14]. It is therefore necessary to carry out preliminary compliance tasks, e.g. specifying required resources [GSALS14]. To this end, it can be essential to achieve knowledge about the application's performance under the expected application workload. Since application workload can fluctuate over time, it may be necessary to adapt the application topology to secure the fulfilment of predefined performance requirements[GSAGL14].

A fundamental aspect in considering a Cloud application's performance is the evolution of the application workload behavior [GSAGL14].

## 2.4 Cloud Application Distribution

Cloud computing brings the opportunity for applications partially or completely composed of cloud offerings [AGSLW14]. Offered services like Database as a Service can be integrated as a part of new applications or existing applications migrated to the cloud [AGSLW14].

Standards like TOSCA enable developers to build portable and interoperable topology models of application stacks. These models can be used for the distributed deployment of applications across cloud providers [AGSLW14], [ARXL14].

### 2.4.1 Optimizing the Distribution

TOSCA et al. describe application topologies and middleware components and cloud offerings involved, but that's a limited view. The description is only a possible instantiation of the application, it doesn't contains alternative instantiations [AGSLW14]. Although developers using TOSCA et al. for the portable and interoperable modelling of applications and their distributed deployment, their is still a lack of support that will facilitate the search for e.g. cost-optimal design and distributions for these applications. Numerous cloud service offerings with different pricing models and various performance characteristics make it difficult to identifying the optimal distribution of an applications in the cloud [ARSL14]. [ARXL14] identifies a need for guidance and support in identifying the most cost efficient distribution in the set of available cloud offerings. Thus support should facilitate two main challenges: (1) how applications should be distributed among cloud solutions and (2) which cloud offering (and configuration) should be used [ARXL14]. While [ARXL14] aims at cost-efficient, [AGSLW14] aims at the fulfilment of a more comprehensive set of characteristics. The main challenges are still similar: (1) infer possible topologies for a given application and (2) select the optimal topology for a given set of characteristics.

The selection and configuration of cloud offerings for the deployment of a application has a direct impact on the applications operating costs. Developers can optimize these operating costs both in the design of new applications and the re-engineering of existing applications [ARXL14]. The suggested process contains 4 stages: (1) Application modelling - In a first step the developer models the application topology independent of specific cloud offerings,

typically using a directed acyclic graph. (2) Mapping offering - The model from (1) is used to identify cloud offerings in which the application can be deployed. The result is a set of so-called enriched topologies. Analogous to [AGSLW14], a single enriched topology contains only offers from one cloud provider. [AGSLW14] explains this constraint with expected latencies between providers. (3) Cost calculation - Projected costs are calculated for the enriched topologies and added as an annotation based an a presumed usage profile and public available informations about cloud offers. The result is a set of so-called cost-annotated topologies. (4) Optimal topology selection - The set of cost-annotated topologies displays the operating cost optimal enriched topology with respect to a usage profile. Different usage profiles can be evaluated by looping back to (3).

The approach in [AGSLW14] also provides support in identifying the optimal distribution from possible distributions of the application stack across cloud offerings. As distinct from [ARXL14], this approach not only aims for an operating cost optimal distribution. Different criteria allow the selection of a optimal distribution in a more comprehensive context.

Figure 2.1 shows a distributed deployed application. The front-end tier is deployed on another infrastructure solution then back-end and persistence tiers. The figure indicates alternative topologies (dotted lines). It's possible to choose other infrastructure solutions, utilize other PaaS-offers or even separate back-end and persistence tier by switching from a database server deployed on a platform to a DBaaS offer. It's possible to deploy the basic application with different topologies. The application's appearance can be different, indeed. Essential characteristics of the application like e.g. deployment time, performance and operational expense depend on the (cloud) offers in the respective topology. Furthermore, different topologies can include different pricing models [AGSLW14]. All these factors can be relevant when determining the value of the application. Anyway, the evaluation process is similarly constructed to [ARXL14]. First, based on a model of the application's topology alternative distributed scenarios are generated. Second, these alternatives are evaluated. The difference is the evaluation with respect to multiple criteria.

### 2.4.2 Application Topology Fundamentals

The approach in [AGSLW14] lists definitions for central terms in the context of application topologies. These definitions are summarized in this section.

**Application Topology** The labeled graph $G = (N^L, E^L, s, t)$ with $N$ as a set of nodes, $E$ a set of edges, $L$ a set of labels and source and target functions $s, t$ defines the application topology. If $L$ only contains elements $< name : type >$ of nodes and elements $< type >$ for edges, the topology graph is called typed [AGSLW14].

Standards like TOSCA use the typed typed topology graph model to describe applications, middleware components and cloud offerings under a unified model [AGSLW14].

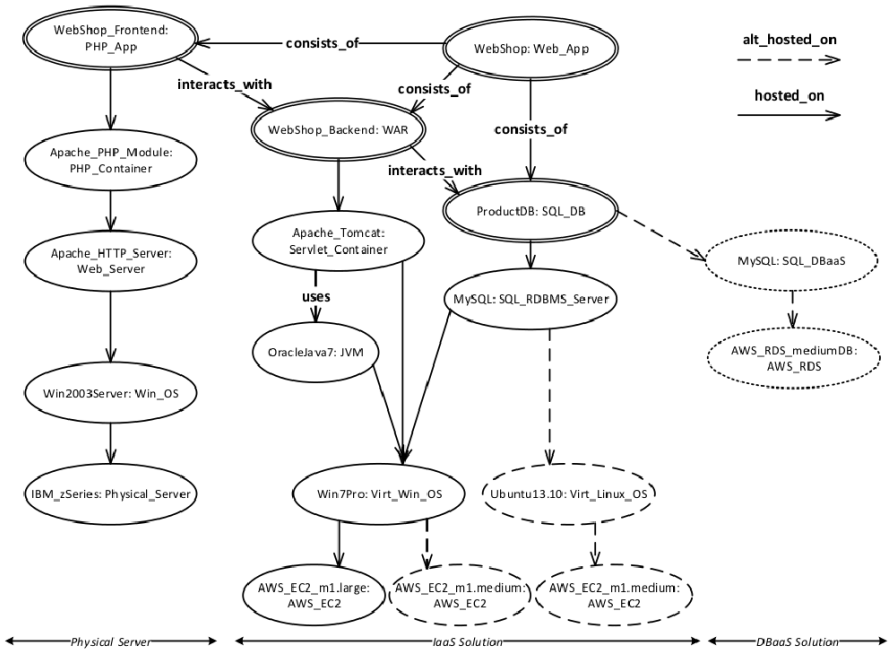


**Figure 2.1:** Web Shop Application Topology

From [AGSLW14]

**Viable Topology** In order to manage the above mentioned challenge of inferring possible topologies for a given application [AGSLW14] extends the typed graph model towards a typed graph with inheritance model. This approach is following [BEDL+03]. This graph model reports abstract nodes (e.g. web server) and contains inheritance clan relations for nodes. The clan morphism $clan(n)_I = \{n' \in N | \exists path\ n' \rightarrow^* n\}$ defines the inheritance clan relations. By navigating through the inheritance-type edges in the graph and using the clan relations, alternative typed topology graphs can be found. These typed topologies are called viable topologies (w.r.t. a typed graph with inheritance) [AGSLW14].

**$\mu$-, $\alpha$- and $\gamma$-Topology** The typed graph with inheritance for a viable topology is defined as its $\mu$-topology. The $\alpha$-topology denotes the application-specific sub-graph and the $\gamma$-topology the non application-specific and reusable sub-graph [AGSLW14].

The division into $\alpha$- and $\gamma$-topology is functional and guided by the applications particular need [AGSLW14].

## 2.5 Cloud Consumers' Requirements

Cloud costumers challenging the fact that distributed solutions have specific requirements that need to be met by integrated cloud services [GVB11]. With the increasing number of cloud offerings it becomes more difficult for consumers to decide which offering can fulfil

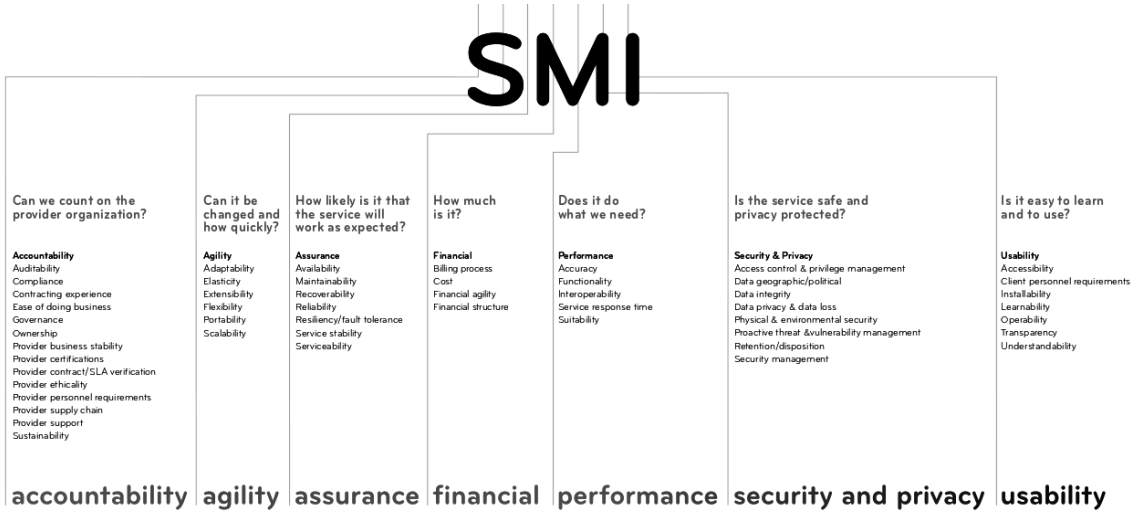


**Figure 2.2:** CSMIC Framework, v2.1

From [CSM14]

these requirements. Furthermore, these decisions require trade-offs between different requirements [GVB11]. Thereby the most appropriate cloud offers with respect to the complete solution should be selected [YMBD14].

There are two types of requirements: functional and non-functional. Functional requirements describe what the service is supposed to do, while non-functional requirements describe how the service is supposed to be. The term quality of service (QoS) is is synonymous with non-functional requirements. In cloud computing various cloud providers offer functionally equivalent services but these services differ in their non-functional qualities [KDM13]. Usually developers select cloud services as partial solution to their requirements [GGS10]. Cloud consumers aim should be the selection of offers that fulfil functional and non-functional requirements.

The Service Measurement Index (SMI) framework addressing the need for industry-wide, globally accepted measures for calculating the benefits and risks of cloud computing services [SP12]. The SMI framework (current version 2.1 [CSM14]) includes 7 major characteristics and their sets of attributes. These characteristics are (1) Accountability, (2) Agility, (3) Assurance, (4) Financial, (5) Performance, (6) Security and Privacy and (7) Usability. Figure 2.2 displays theses characteristics and their attributes. The attributes are functional and non-functional requirements.

[GVB11] outlines the meaning of the seven categories: The attributes in (1) determine the trust consumers build on cloud providers. Since agility is one of the outstanding characteristic in cloud computing, the attributes in (2) reveal a provider's ability to provide this fundamental opportunity by its offers. Characteristic (3) refers to the compliance of cloud services' expected or promised performance. Costs are a strong argument when comparing alternatives, thereby attributes from category (4) have a strong influence. Attributes in (5)

describe features and functions of cloud services. Hosting data in other organisations control is a critical issue, thereby attributes in category (6) are vital in several domains. The ease which a service can be used is described with attributes in (7). These attributes determines how fast cloud consumers can switch to cloud services.

The SMI framework categories and included attributes summarize the potentially relevant functional and non-functional requirements. Cloud consumers can select a set of attributes relevant their particular solutions. [GVB11] also lists metrics for the usually relevant quantitative attributes in the context of IaaS clouds: These attributes are (1) Service Response Time, (2) Sustainability, (3) Suitability, (4) Accuracy, (5) Transparency, (6) Interoperability, (7) Interoperability, (8) Availability, (9) Reliability, (10) Cost, (11) Adaptability, (12) Elasticity and (13) Usability.

[KDM13] lists some QoS and how to determine their values: (1) Availability - The users' reachability of the service. The percentage of availability is determined based on the mean time to fail (MTTF) and the mean time to repair (MTTR) by $\frac{MTTF}{MTTF+MTTR}$. (2) Reliability - The service's ability to function according to performance requirements in SLA. [KDM13] uses defects per million (DPM) to determine the percentage of a services' reliability: $\frac{1{,}000{,}000-DPM}{1{,}000{,}000}$. (3) Response time - The time between sending a request and receiving a response. (4) Usability - Rating for the easiness of using, learning and installing a service. The rating is based on the users' feedback and reflect the degree of satisfaction regarding this facet. (5) Security - Characteristic of being secure regarding applications, hardware components and users' data. [KDM13] determines security with the fulfilment of expected security and security provided by a service. (6) Cost - The monetary expense of accessing and using a service. The measurement unit is money (US dollar) and its tendency is low.

## 2.6 Service-orientated Architecture and Computing

SOA emerged in response to a shift in business organizations after the 1990s which requires flexible and responsive IT environments [EAA+04]. Seamless connections with participants (with heterogeneous systems and applications) in the supply chain demands new approaches in software architecture [EAA+04].

### 2.6.1 Service-oriented Computing

*Service-oriented Computing* (SOC) is a paradigm "*that utilizes services as fundamental elements for developing applications*" [PG03]. These services are autonomous, platform-independent entities [PTDL07]. SOC eliminates the dependence of programming languages and operating systems and any application component can transformed into a reusable, network-available service [PTDL07].

### 2.6.2 Service-oriented Architecture

Heterogeneity, interoperability and variable requirements are easier manageable in a loosely coupled, location transparent and protocol independent architecture [EAA+04]. [Org06] summarizes SOA as a "*paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains*". Putting this paradigm into practice brings more value from the use of self-owned and foreign controlled capabilities.

#### Roles

SOA services deliver application functionality to end-users, applications or other services in distributes systems [EAA+04]. Following [EAA+04] these services are handled by entities of three different roles, whereby entities can play different roles side by side: (1) Service consumer, (2) Service provider and (3) Service registry. Entities of (1) are applications, software modules ore other services that require a service. Entities of (2) are network-addressable and handling requests from (1). Entities of (3) enables service discovery. Service providers publish its service and interfaces contracts to service registries. Service registries contain repositories of available services and can be queried for provisioning service providers interfaces. Thus service consumers enquire services from service registries and execute these services according to the interface contract of the respective service provider.

#### Operations

Beside these three roles, there are tree operations in SOA [EAA+04]: (1) publish, (2) find and (3) bind and invoke. Service providers providers publish service descriptions with the help of service registries in order to make them accessible to service consumers. With find-operations service consumers locate services by querying service registries and finally bind and invoke the services according to the received service descriptions.

#### Concept

[Org06] identifies three key concepts in the SOA paradigm: (1) visability, (2) interaction and (3) effect. (1) refers to the possibilities matching needs to capabilities, typically by widely accessible and understandable descriptions for e.g. functions, requirements, constraints, mechanisms and policies. (2) refers to the use of capabilities, usually by messages for information exchange and invoking actions. The result of interactions is referred by (3). This could be e.g. the return of informations or the change in the state off entities.

SOA provides scalability and evolving by making the fewest possible assumptions about the network. That also leads to more agile and responsive infrastructures and simplifies the integration of functionalities over ownership boundaries [Org06].

#### Web Services

In a nutshell the web service approach is an enabler for the programming language independent integration of heterogeneous applications over the internet [EAA+04]. [OMN+04] provides a definition of web services: *"A Web service is a software system designed to support interoperable machine-to-machine interaction over a network."* Web services use open Internet-based standards. The *Simple Object Access Protocol* (SOAP) is used for transmitting data, the *Web Service Description Language* (WSDL) for service definition and the *Business Process Execution Language for Web Services* (BPEL4WS) for service orchestration [PTDL07]. The *Universal Description, Discovery, and Integration* standard (UDDI) enables the location of web services and the discovery of their details [PG03]. SOA is commonly implemented with the use of web services [EAA+04].

## 2.7 Utility Theory

A fundamental cornerstone of this thesis is utility functions' capability to combine the evaluation of different objectives into one axis [STFG08]. This section presents definitions of utility functions and provides an insight into the their usage in economics.

### 2.7.1 Definitions

Different approaches provide deviating definitions for utility functions. [Fis70, JR01] provides a mathematically attempt, [Nor99], [Joh07] a more business-orientated attempt.

The approaches in [Fis70, JR01] using binary relations on the countable set of $X$. Relations are *preference relation*, *strict preference relation* and *indifference relation*.

| | | |
|---|---|---|
| $\succeq$ | *preference relation* | $\forall x_1, x_2 \in X : x_1 \succeq x_2 \vee x_2 \succeq x_1$ |
| | | $\forall x_1, x_2, x_3 \in X : x_1 \succeq x_2 \wedge x_2 \succeq x_3 \Rightarrow x_1 \succeq x_3$ |
| $\succ$ | *strict preference relation* | $\forall x_1, x_2 \in X : x_1 \succ x_2 \Rightarrow x_1 \succeq x_2 \wedge x_2 \not\succeq x_1$ |
| $\sim$ | *indifference realtion* | $\forall x_1, x_2 \in X : x_1 \sim x_2 \Rightarrow x_1 \succeq x_2 \wedge x_2 \succeq x_1$ |

With the binary relation strict preference $\succ$ on the countable set of $X$ [Fis70] describes a basic property of utility functions $u : \mathbb{R}^n_+ \to \mathbb{R}$

$$x \succ y \Leftrightarrow u(x) > u(y) \tag{2.1}$$

This equivalence expresses that the utility of $x$ is higher than $y$ if and only if $x \succ y$.

[JR01] notes that a utility function is a device for summarising the informations contained in a consumer's preference relation. Therefore, [JR01] defines a utility function as follows, essentially utility function $u : \mathbb{R}^n_+ \to \mathbb{R}$ representing the preference relation.

$$u(x) \geq u(y) \Leftrightarrow u \succeq y \tag{2.2}$$

[Nor99], [Joh07] considers utility functions from the perspective of an investor. Utility functions are functions of wealth $U(w)$ with w > 0. Thereby two characteristics are important: (1) non-satiation and (2) risk aversion. Basically (1) means more wealth is always better, the investor is never satisfied. This characteristic of $U(w)$ is secured if $U'(w) > 0$ for all $w$. Characteristic (2) describes the decreasing marginal utility of $U(w)$. The difference of the utility of 1 \$ and 2 \$ should be bigger than the difference of the utility of 1000\$ and 1001\$. (2) is secured if $U''(w) < 0$ for all $w$. Following [Nor99], [Joh07], utility functions are twice-differentiable functions of wealth $U(w)$ with $U'(w) > 0$ and $U''(w) < 0$.

#### Positive Affine Transformations

$\succeq$ is a preference relation on $\mathbb{R}^n_+$ and utility function $u(x)$ represents it. Then $v(x)$ also represents the relation if and only if $v(x) = f(u(x))$, where $f : \mathbb{R} \to \mathbb{R}$ is strictly increasing [JR01]. The less mathematical approach is that it is possible to scale a utility function by multiplying or translate it with any positive constant [Nor99].

### 2.7.2 Usage in Economics

Chapter 3 analyses the utilization of utility theory in various software architectures. First, the utilization of utility theory in financial risk-management is considered. [Joh07] lists common utility functions for this purpose. Among others: (1) exponential, (2) logarithm and (3) isoelastic functions. [Nor99] gives a more detailed description of these utility functions. These functions are explained below.

#### Exponential Utility Function

Since the definition of utility functions in [Nor99, Joh07] requires $U'(w) > 0$ and $U''(w) < 0$ the basic form of the (negative) exponential utility function (1) is

$$U(w) = -e^{-Aw} \tag{2.3}$$

with constant $A > 0$. The upper boundary of $U(w)$ is obviously 0 and the coefficient of risk aversion $A$ determines how fast the function approximates to 0. $U''(w) = -A^2e^{-Aw}$, thereby $A$ has an exponentially impact on the marginal utility. With e.g. $u(x) = 1 - e^{-Aw}$ positive exponential utility functions can be formalized [Joh07]. Figure 2.3 shows the characteristic of a positive exponential utility functions with different risk aversions $A$.

#### Logarithm Utility Functions

The natural logarithm

$$U(w) = ln(w) \tag{2.4}$$

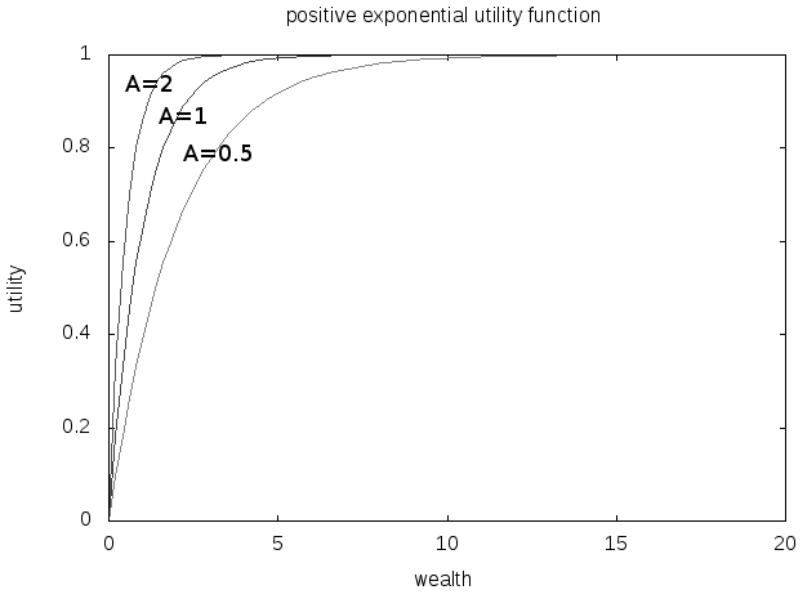


**Figure 2.3:** Positive Exponential Utility Function

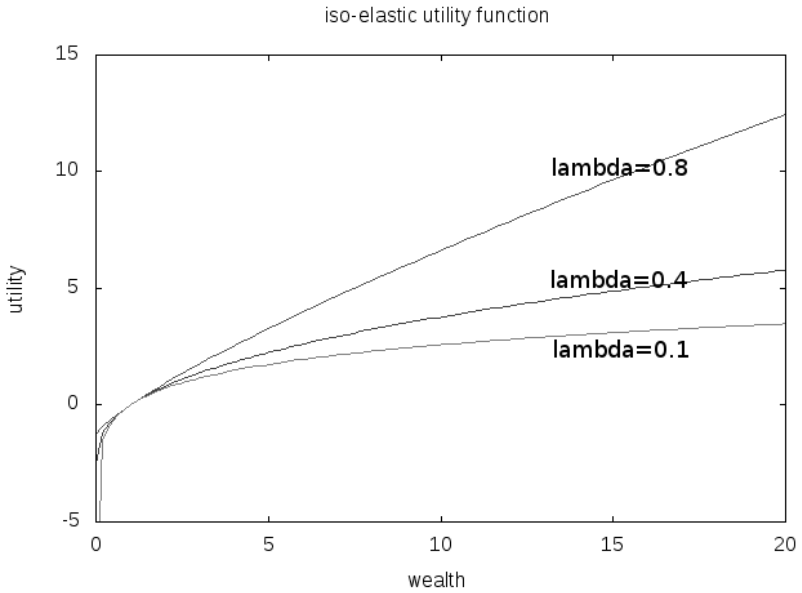


**Figure 2.4:** Iso-Elastic Utility Function

distinguishes from the (negative) exponential function. Logarithmic functions growing without boundaries.

#### Iso-Elastic Utility Functions

$$U(w) = \begin{cases} \frac{w^{\lambda}-1}{\lambda} & \lambda < 0, \lambda \neq 0 \\ ln(w) & \lambda = 0 \end{cases} \tag{2.5}$$

Logarithm utility functions are obviously a special case of iso-elastic utility functions. The special character of iso-elastic utility functions is the constant attitude towards risk regardless the current wealth [Nor99]. If a iso-elastic utility function is used to determine the optimal amount of money to invest in a risky investment, the same percentage of the available amount of money will come up. This characteristic is unique within the classes of utility functions.

## 2.8 REST

Representational State Transfer (REST) is a software architectural style and was first discussed in Roy T. Fielding's dissertation [Fie00]. The only mandatory implementation details are the use of HTTP verbs (GET, PUT, POST and DELETE), naming of resources using nouns (e.g. URI) and the interconnection of resources with URIs [otSS11]. REST is a set of design criteria and not a architecture [RR07].

### 2.8.1 Resources

*"A resource is everything that's important enough to be referenced as a thing in itself"* [RR07]. A resource can represent physical or abstract objects. A resource can be stored and represented as a stream of bits. That addresses e.g. a document or even the result of an algorithm [RR07]. The minimum requirement for being a resource, is having one URI [RR07].

#### URI

REST relies on uniform named resources. Uniform Resource Identifiers (URI) are used to meet this requirement [otSS11]. A URI is name and address of a resource [RR07].

### 2.8.2 Constraints

REST combines constraints of already existing network-based architectural styles with additional constraints that define a uniform interface [Fie00].

**Separation of concerns** This constraint demands the separation of concerns of user interface and data storage. The constraint improves the portability of user interfaces, increases scalability and allows for the independent development of components [Fie00].

**Stateless** The server doesn't store any application state [RR07]. Session state is completely covered by clients [Fie00], if necessary the client must send the application state as part of the request [RR07].

**Cacheable** A cache acts as a mediator between client and server [Fie00]. Responses define themselves as cacheable or not (implicitly or explicitly) [Fie00].

**Uniform interface** [Fie00] names four constraints in order to abtain a uniform interface: (1) identification of resources, (2) manipulation of resources through representations, (3) self-descriptive messages and (4) hypermedia as the engine of application state (HATEOAS). [RR07] explains HATEOAS as followed: The server does not store application states, but guides the client's path by serving links and forms inside hypertext representations.

**Layered System** Each layer provides services to the layer above it and using services of the layer below it [Fie00]. This architectural style improves evolvability and reusability, but comes with the costs of additional overhead [Fie00].

### 2.8.3 HTTP commands

A fundamental tenet of REST is the utilization of the HTTP commands GET, PUT, POST, DELETE [otSS11]. All Interactions between clients and resources are mediated through this set of verbs [RR07]. In addition the HEAD command can be used.

**GET** The GET command is used to receive a resource's representation. GET commands shouldn't cause side-effects [otSS11]. Therefore, GET commands has to be implemented in the way that resources aren't altered. Thus it guaranteed that the GET command is idempotent.

**PUT** PUT commands are used to update, replace or create a resource [otSS11]. PUT should only be used to create a resource if clients can decide the resources URI. If that were not the case, the definition of PUT as an idempotent method [FGM+99] would be undermined.

**POST** POST commands are used to change, update or replace a resource [otSS11]. In contrast to PUT commands, POST commands are not defined as idempotent [FGM+99]. This allows for the creation of resources without the resources' URI in the request [otSS11].

**DELETE** DELETE commands removing a resource. The server should confirm the successfully or failed removal with the response [otSS11]. This command is also idempotent.

**HEAD** The HEAD command is identical to the GET command except that the server must only return the message header [FGM+99]. The HEAD command can be used to obtain resources' meta data or check for the existence of a certain resource [otSS11].

## 2.9 Nefolog and MiDSuS Cost Calculation Framework

Since various cloud providers offer cloud services with similar features, there is a need for support in selecting cost-efficient offers. [XA13] introduces Nefolog, a decision support system for selection offering candidates and calculating costs. Furthermore MiDSuS, a decision support system for different migration types based on Nefolog, is introduced.

### 2.9.1 Nefolog

Nefolog provides two main decision support services: (1) Candidate search and (2) cost calculation. (1) provides the search for suitable cloud offerings by the comparison of user demands and the data in a knowledge base. The knowledge base contains necessary information for the offerings of several providers (Google, Amazon Web Service, et al.) including available configurations with different performance characteristics and costs. Offerings are divided in 6 different service types, e.g. web service, SQL DB and infrastructure. (2) calculates total costs based on upfront costs, data transfer costs and service costs. Nefolog provides corresponding cost functions for cloud offers. The services are offered in a RESTful manner. XML and JSON data format is supported.

### 2.9.2 MiDSuS

MiDSuS is a decision support system for finding the best cost cloud offers for different migration types. The system has a graphical user interface and utilizes the Nefolog services. First, users select their migration type. Second, candidate offerings are found based on the users demand. Then MiDSuS calculates costs for user selected offerings and last, the results are presented.

# 3 Related Works

This chapter presents different approaches which are utilizing utility functions. Approaches from Cloud computing, Service-oriented architecture and the provisioning of storage systems are presented. The approaches are subsequently compared.

## 3.1 Approaches in Cloud Computing

Two approaches [MF11, KM14] utilizes utility functions to break different aspects into one axis. Another work [SP12] doesn't utilizes utility but provides a valuable approach.

### 3.1.1 Utility-based Resource Allocation for Virtual Machines

The approach in [MF11] addresses the trade-off between quality of service constraints and the minimization of operational cost in the field of IaaS Clouds. The challenge is to dynamically allocate resources to virtual machines (VMs) considering this problem. [MF11] tackles the problem on two tiers. (1) Local controllers using utility functions to allocate CPU shares to VMs in a way that maximizes the local node's utility and (2) a global controller maximizes the result of a global system utility function by initiating live migrations of VMs to other nodes. Local nodes are optimized by giving higher CPU shares to VMs with higher priorities. The global controller periodically collects VMs' CPU requirements from the local nodes and migrates (if appropriate) selected VMs to other physical machines in order to maximise the global utility function's result.

Consumers run their applications on VMs offered by a Cloud provider. The Cloud provider's objective should be maximizing his profit. In this case profit is determined by the amount of money earned from consumers minus the operating costs of the infrastructure for a given time interval. This preference should be represented by the utility function.

A VM's utility function represents the amount of money paid by the consumer for using the VM in a control interval. [MF11] defines this utility function as a linear function of CPU resources the VM gets from the provider, but also mentioned that it can also be the function of performance metrics laid down in SLAs. A local nodes utility function calculates the profit by a single node in a control interval. Finally, the global system utility function represent the profit produced by the entire system in a control interval.

#### Utility Functions

$U_{j,i}$ is the utility function for VM $i$ on node $j$. With node $j$'s costs $C_j$ and a total number of $m$ VMs hosting on the node, the node's utility function is defined by:

$$N_j = U_{j1} + U_{j2} + ... + U_{jm} - C_j \tag{3.1}$$

$U_{ji}$ is defined by $U_{ji} = \alpha_{ji} S_{ji}$ where $\alpha_{ji}$ is the amount of money paid per unit of CPU resource allocated and $S_{ji}$ is number of allocated VM units.

The global system utility function $U_g$ is given by:

$$U_g = N_1 + N_2 + ... + N_n \tag{3.2}$$

The global system contains $n$ nodes.

#### Practical Application

Each local node controller maximizes his node utility function. A monitoring component measures the average CPU utilization of VMs in every control interval. If the total CPU capacity is not enough to supply all VMs, the node utility function is maximized by first satisfying requests of VMs that offers a higher utility per unit of CPU resource. VMs with lower utility per unit of CPU resource will never get CPU resources.

The global controller queries the local node controllers in each control interval for all VMs' CPU requirements and CPU shares. The problem of finding the VM mapping which maximizes the global system utility function is NP-hard. Therefore, [MF11] uses a heuristic algorithm that suggests, if available, a list of live migrations that increases the global system utility.

### 3.1.2 Price and QoS Competition in Cloud Market

Cloud providers set different prices for each configuration of there services. A user's demand may be met by a number of providers. It is supposed, that the user chooses a service (and thereby a provider) that maximizes the utility obtained by choosing the service minus the payment for the service [KM14].

The approach in [KM14] understands that providers facing the challenge to set a price that maximizing their profit in a competitive cloud market. Therefore, providers determine Cloud users' utility. A user's utility is determined by the benefit the user receives by finishing his task (importance) and how quickly the user wants this task to be finished (urgency).

Utility function $U_j^i$ calculates user $j$'s utility by choosing cloud provider i. $U_j^i$ is calculated with respect to the request rate at cloud user $j$ $\lambda_j$, a benefit factor $d$, a waiting cost factor $w$

and the expected finish time for user $j$'s request when choosing provider $i$ $f_j^i$ (this ratio is a simplification of the actual concept in [KM14]).

$$U_j^i = d * \lambda_j - w * f_j^i \tag{3.3}$$

The approach assumes that users will choose a provider if utility minus the payment is higher than a value $R$. With $p_i$ (*usage price per resource unit at cloud provider i*) this can be expressed by:

$$U_j^i - p_i * \lambda_j \geq R \tag{3.4}$$

This approach is characterised by the fact of using factors to value importance and urgency. It is necessary to assign appropriate values to $d$, $w$ and $R$ to meet the users' actual behavioural with $U_j^i - p_i * \lambda_j \geq R$. However, this makes it possible to solve the trade-off between a monetary facet ($p_i * \lambda_j$) and the utility achieved with respect to importance and urgency.

### 3.1.3 Service Measurement Index

The Service Measurement Index (SMI) framework addressing the need for industry-wide, globally accepted measures for calculating the benefits and risks of cloud computing services [SP12]. Since the SMI framework is intended for decision makers who considering moving services to cloud providers, there is a service evaluation method given. This method is based on measures for the attributes inside the SMI categories (see section 2.5) defined by the framework-user. These measures needs to be clearly and relatively simple defined for each (or each selected) attribute [C, 2012]. The SMI framework supports some customizations of the relative importance of each measure. First the 7 categories can be ranked by percentage weights and second it's possible to assign relative weights to the attributes within the categories (see section 2.5). While the SMI framework (version 2.1) contains 51 attributes distributed among 7 categories, it isn't necessary to use measures for all attributes. Typically users will select the most important categories in their decision-making process and declare a small number of measures within. Attributes are valued, e.g. on a scale from 0 to 5. This procedure makes it possible to trade-off different aspects.

This approach could be useful in the trade-off between e.g. quality of service constraints and the minimization of operational cost. Therefore, this approach is mentioned, even it is not the explicit utilization of utility theory.

## 3.2 Approaches in Service Oriented Architecture

Three approaches are presented. The approaches [YZL07, Max05, HS10] utilizes utility functions. From our perspective, the first approach requires some adjustments. These adjustments are also presented. The last approach is interesting because it calculates not only utility but also the expected utility. Thus, the probability of a certain quality is also considered.

### 3.2.1 QBroker

[YZL07] takes a look at QBroker and clarifies the need for such approaches. In Service Oriented Architecture (SOA) composite services can be created by integrating atomic services based on standardised protocols. Since various potential atomic services with similar and compatible functionality may be offered at different QoS levels, the necessary selections affects the performance of composite services. Thus performance management is an important challenge for distributed SOA systems. Besides economic factors, the integrated services have to fulfil appropriate QoS levels. [YZL07] indicates four factors in charge of the complexity of service selection: (1) various services with different QoS levels are candidates for one functionality, (2) composite services could require various performance constraints, (3) there may exist more ways to build a service and (4) actual QoS may differ from promised QoS.

[YZL07] presents the description of composite services based on service classes connected by different structures (Sequential, AND split, XOR split, Loop, AND join, XOR join). For every service class $S_i$ exists a one-to-many relationship between $S_i$ and atomic services $s_{ij}$ from class $S_i$. In summary, the QBroker compute the best choice of atomic services for a described composite service. If there are XOR-connections, QBroker also compute the best alternative. In this context "best choice" means the composite service (determined by chosen alternatives and chosen atomic services) fulfils QoS requirements and achieves the highest utility in the set of possible choices. QBroker computes the solution either by solving a multidimensional multi choice knapsack problem or by multi-constrained optimal path selection. But that part is out of focus of this thesis. However it's worthwhile to examine how QBroker calculates the utility of composite services and included atomic services based on QoS.

In [YZL07] utility calculation is based on the user-defined utility function $\mathcal{F}$. Since QBroker uses service classes and atomic services, $\mathcal{F}_{ij}$ is the utility function for atomic service $j$ of service class $i$ where $x$ QoS attributes to be maximized and $y$ QoS attributes to be minimized:

$$\mathcal{F}_{ij} = \sum_{\alpha=1}^{x} (\omega_\alpha * (\frac{q_{ij}^\alpha - \mu^\alpha}{\sigma^\alpha})) + \sum_{\beta=1}^{y} (\omega_\beta * (1 - \frac{q_{ij}^\beta - \mu^\beta}{\sigma^\beta})) \tag{3.5}$$

where $(0 < w_\alpha, w_\beta < 1)$ are the weights for each QoS attribute, and $\mu, \sigma$ are the average and standard deviation of the QoS values for all candidates in the service class. Together with $\sum_{\alpha=1}^{x} \omega_\alpha + \sum_{\beta=1}^{x} \omega_\beta = 1$ the utility function is normalized and will not be biased by attributes with large values.

#### Improvement

The approach shows problems but it seems to be paying off, to improve the utility function. $\mathcal{F}_{ij}$ provides a unusual dealing with negative QoS values (e.g. the cost attribute). The utility function in [YZL07] separates QoS attributes to be minimized from QoS attributes to be maximized and evaluate them different. First, there seems to be an inconsistency. For example if

a QoS attribute to be maximized $g_{ij}^{\alpha}$ is equal to the average $\mu^{\alpha}$ then $\omega_{\alpha} * (\frac{q_{ij}^{\alpha} - \mu^{\alpha}}{\sigma^{\alpha}}) = 0$ while the same situation for QoS attributes to be minimized ($q_{ij}^{\beta} = \mu^{\beta}$) results in $\omega_{\beta} * (1 - \frac{q_{ij}^{\beta} - \mu^{\beta}}{\sigma^{\beta}}) = \omega_{\beta}$. Since both cases should be rated the same, $\mathcal{F}_{\rangle|}$ should be adjusted. Meeting the average will be evaluated with zero, therefore the minimization handling will be adjusted.

$$\mathcal{F}_{ij}^{*} = \sum_{\alpha=1}^{x} (\omega_{\alpha} * (\frac{q_{ij}^{\alpha} - \mu^{\alpha}}{\sigma^{\alpha}})) + \sum_{\beta=1}^{y} (\omega_{\beta} * (-\frac{q_{ij}^{\beta} - \mu^{\beta}}{\sigma^{\beta}})) \tag{3.6}$$

However, providing the attributes with minus-signs could avoid different evaluations. E.g. a atomic service's cost attribute is then specified with minus-sign. In $\mathcal{F}_{ij}$ the average of the QoS attribute in service class $i$ would also appear with switched sign while the standard deviation leave unchanged (square root of the variance). Whereas $(-\frac{q_{ij}^{\beta} - \mu^{\beta}}{\sigma^{\beta}}) = \frac{-q_{ij}^{\beta} + \mu^{\beta}}{\sigma^{\beta}}$, $\mathcal{F}_{ij}^{*}$ can be simplified to

$$\mathcal{F}_{ij}^{**} = \sum_{\alpha=1}^{x+y} (\omega_{\alpha} * (\frac{q_{ij}^{\alpha} - \mu^{\alpha}}{\sigma^{\alpha}})) \tag{3.7}$$

### 3.2.2 Agent-Based Trust Model

In [Max05] a framework for web service selection based on consumer's QoS preferences is discussed. Since WSDL-based service description hasn't the capabilities of representing non-functional service attributes, [Max05] discusses a trust model rested on a shared conceptualization of QoS. In this context trust is the aggregation of historical levels of quality and the conformity between the providers quality advertisement and the consumers quality needs. The trust-concept extents the utility-concept or more precisely is build around consumers utility.

With $I_s$ as the set of all service implementations of service $s$, selection of the 'best' service implementation is defined by

$$i = argmax_{i \in I_s} \{trust(i, \phi_d)\} \tag{3.8}$$

With $\phi_d$ as the set of qualities applicable to the application domain $d$, the equation identifies the implementation with the highest trust value for domain $d$.

Whereas the meaning of the trust function is obvious, the function's structure is more complicated. First, the aggregation of historical levels of quality - the *Service Quality Reputation* $R_{Q_i}$- with respect to quality $Q$ for service implementation $i$ is calculated by

$$R_{Q_i} = \frac{1}{n} \sum_{k=0}^{n} q_k \delta^{-t(q_k)} \tag{3.9}$$

where $\{q_0 ... q_n\}$ is the set of quality values collected from agents that previously selected implementation $i$, $\delta \in \mathbb{R}$ is the quality $Q$'s damping factor and $t(q)$ depends on the time

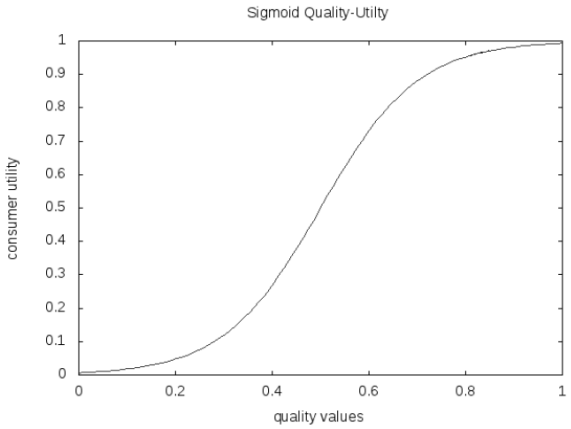


**Figure 3.1:** Sigmoid Utility Function

Derived from [Max05]

for which $q$ was collected ($t(q) = 1$ for the latest collected value and $t(q) > 1$ for all other values). Values $q$ expressing a quality level between 0 and 1.

For every quality $Q$ in $\phi_d$ (the set of all qualities in domain $d$) [Max05] defines a vector $\pi_Q = (\pi_{min}, \pi_{preferred}, \pi_{max})$ representing the consumers preferences for quality $Q$ by the minimum acceptable, preferred and maximum acceptable value. A similar vector is given by the advertisement of implementation $i$'s provider. $\alpha_Q = (\alpha_{min}, \alpha_{typical}, \alpha_{max})$ represents the providers information about $i$'s quality $Q$.

Determining the preferred value for qualities requires qualitative analysis. In [Max05] utility functions are used to transfer the value of a specific quality into utility. Thereby values becomes comparable, utility indicates with quality's value provides better utility in the service's domain. [Max05] considers only two different kinds of utility functions. The first kind is a linear shaped utility function. The utility is linear with the quality's value. The second kind is the sigmoid utility function. The s-shaped curve describes qualities with insufficient utility up to a certain threshold value. The utility increases rapidly after the threshold and flatten out again. Figure 3.1 shows a sigmoid utility function. The definition of the function is given by:

$$u(q) = \frac{1}{1 + e^{-\alpha q + \beta}} \tag{3.10}$$

Constants $\alpha$ and $\beta$ are used to express consumers utility.

[Max05] evaluates the conformity of the users preferences, the advertisement and the reputation of quality $q$ for an implementation $i$ based on the vector
$\vec{Q}_i = \langle Q_{min}, \alpha_{typical}, \pi_{preferred}, Q_{max}, R_Q^{(i)} \rangle$
, where $Q_{min} = min(\alpha_{min}, \pi_{min})$, $Q_{max} = max(\alpha_{max}, \pi_{max})$ and $R_Q^{(i)}$ is the service quality reputation regarding quality $Q$ in implementation $i$. With

$$moment(\vec{x}, a) = \frac{1}{n-1} \sum_{i=1}^{n} (a - x_i)^2 \tag{3.11}$$

($\vec{x} = \langle x_1, x_2, ..., x_n \rangle$) the conformity can be calculated by $moment(\vec{Q}_i, q_{preferred})$, hence the quality $Q$'s trust in implementation $i$ can be calculated by

$$qTrust(\vec{Q}_i, q_{preferrred}) = moment(\vec{Q}_i, q_{preferrred})^{-\frac{1}{2}} \tag{3.12}$$

At last [Max05] utilizes relationships between different qualities. $\rho(Q_a, Q_b)$ maps two different qualities to $[-1, 1]$. $\rho(Q_a, Q_b)$ rates the correlation from $Q_a$ to $Q_b$ from oppositely (smaller value) to positively (higher value). With

$$Q_x * Q_y = \begin{cases} 1 & dir(Q_x) = dir(Q_y) \\ -1 & otherwise \end{cases} \tag{3.13}$$

(direction $dir(Q)$ indicates if the higher or lower values for quality $Q$ are preferred) the average quality relationship can be calculated by:

$$\varrho(Q_j) = \frac{1}{n-j} \sum_{m=j+1}^{n} \rho(Q_j, Q_m) \times (Q_j * Q_m) \tag{3.14}$$

where $\langle Q_1, ..., Q_n \rangle$ is the set of qualities the user has preferences for, ordered descending by their importance.
In a coarse-grained describing $\varrho(Q_j)$ handles three cases. (1) if $\rho(Q_j, Q_m) \approx 0$ then $Q_m$ has no (significant) influence on the average quality relationship. (2) if $(Q_j * Q_m) = 1$ and $\rho(Q_j, Q_m) > 0$ or $(Q_j * Q_m) = -1$ and $\rho(Q_j, Q_m) < 0$ the average quality relationship is increasing. In this case a higher value for quality $Q_j$ comes with higher values for the less important quality $Q_m$. (3) if $(Q_j * Q_m) = 1$ and $\rho(Q_j, Q_m) < 0$ or $(Q_j * Q_m) = -1$ and $\rho(Q_j, Q_m) > 0$ the average quality relationship is decreasing. In this case a higher value for $Q_j$ comes with a smaller value for $Q_m$.
Robustness and availability is a example for (2). Higher robustness normally comes with better availability. Cost and capacity is also a example for (2). A more expensive service likely provides higher capacity. The utility of the first quality (cost) decreases while the seconds quality utility increases.

Finally the trust function for service implementations can be formulated:

$$trust(i_p) = \frac{1}{n} \sum_{j=0}^{n} w_j \times qTrust(Q_j, q_{pref}) \times [1 + \varrho(Q_j)] \tag{3.15}$$

$1 \geq w_j > 0$ weights the important of quality $Q_i$. All qualities of implementation $i_p$ fulfil the users requirements and for all qualities $Q_0 ... Q_n$ the user has preferred values, minimum acceptable values and maximal acceptable values.

On the one side $trust(i_p)$ utilizes several aspects. The first term ($w_j$) rates the importance of quality $Q_j$ in the service's domain and tunes the impact on the overall result. In [Max05] a simple sequence like $1, \frac{1}{2}, ..., \frac{1}{n-1}, \frac{1}{n}$ is suggested. Since $\langle Q_1, ..., Q_n \rangle$ is ordered descending by importance for the service's domain, the sequence at least fulfil this order of qualities. The $[1 + \varrho(Q_j)]$-term takes the correlations between different qualities into account and increase

the impact of qualities with higher average quality relationships (visa versa). The third term ($qTrust(Q_j, q_{pref})$) doesn't distinguishes between qualities with different directions. $qTrust$ just evaluates the conformance between the quality's advertisement, the user's demand and the implementation's service quality reputation.

On the other side [Max05] requires some assumptions. Users need ratings in $[0, 1]$ for qualities and knowledge about preferred, minimum and maximum acceptable values. Also providers advertisements have to contain typical, minimum and maximum achieved values for all in the evaluation included qualities. Besides that, correlations have to be defined and qualities have to ranked.

### 3.2.3 Adaptive Service Selection Framework

While [Max05] addresses atomic service-selection, [HS10] also addresses service selection for composited applications. [HS10] detects three main challenges in service selection in service orientated computing: (1) Collecting informations about QoS offered by services, (2) defining the consumers preferences for QoS and (3) making decisions based in the collected informations and the defined preferences for QoS.

The service orientated environment is modelled by a set of providers $P = \{P_1, ..., P_m\}$ and a set of consumers $C = \{C_1, ..., C_n\}$. In [HS10] providers are synonymic with service implementations. Providers offers services with the same functionality than other providers but provides different levels of qualities. $Q = \{Q_1, ..., Q_l\}$ represents the levels for the provider-specific qualities.

This approach takes account of consumers preferences of qualities and the providers' quality distribution. A utility functions and the quality distributions determine the expected utilities for providers.

While [Max05] only mentioned monotonic increasing functions, [HS10] emphasizes the need for other functions. E.g. utility functions should also reflect qualities where consumers prefer medium values over small or low values. Generally any utility function should be used. Three examples are given: (1) logistic functions - a special case of sigmoid functions with rapid increase close to the desired value, (2) logarithm functions - characterized by diminishing return and (3) Gaussian functions - describe qualities with decreasing utility beside a maximum point. Figure 3.2 shows a Gaussian utility function. The consumer prefers a medium high quality value. Both, higher and lower values decreases the consumers utility.

The second factor for the expected utility is the quality distribution. $Q_k(x_j)$ is the probability density function of the probability distribution that controls the quality $x_j$ of $Q_k$ for provider $P_j$. The quality distribution can be learned by probabilistic trust models based experience, referrals or composition.

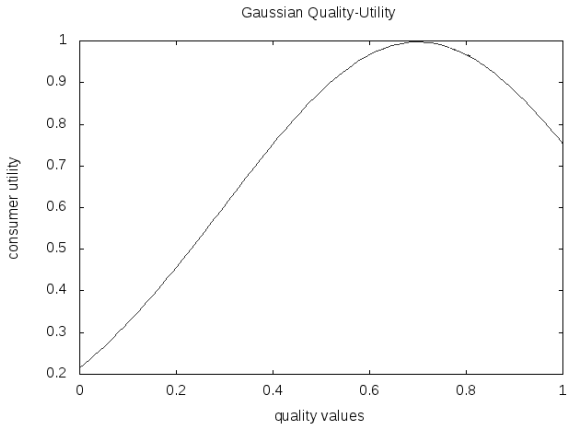


**Figure 3.2:** Gaussian Utility Function

Derived from [HS10]

The expected utility regarding only quality $Q_k$ for consumer $C_i$ with utility function $U_k(x)$ and providers $P_j$'s quality distribution $Q_k(x_j)$ is

$$EU_k^i(P_j) = \int_{x_j} U_k(x_j)Q_k(x_j)dx_j \tag{3.16}$$

## 3.3 Approach in Provision Storage Systems

[STFG08] suggests the usage of utility functions in provisioning storage systems. Coming up with the offer meeting predefined requirements at the lowest price is wasting the potentials in trade-off. Most requirements are flexible based on the costs to implement them. Replacing fixed requirements with utility functions provides a way to balancing quality and benefits of services against the costs required to provide them.

Choosing the alternative with the most value requires appropriate trade-offs among competing objectives. Utility functions comply with this requirement. Utility functions collapse multiple axis of interest into a single utility value. Thus complex alternatives can be compared, even by automated tools.

Various metrics must be combined with respect to their relative importance. Utility functions uses system metrics to rank potential configurations. Metrics must be normalized to each other. An appropriate normalization is the use of a common scale. If all metrics have an business impact, a monetary unit presents such a scale.

[STFG08] uses an online retailer as example. This company determine its online transaction processing (OLTP) workload generates on average 0.1$per I/O. The companies annualized

revenue thereby is

$$Revenue = \$0.01 * IOPS_{WL} * AV_{DS} * \frac{3.2 * 10^7 s}{1yr} \tag{3.17}$$

where $IOPS_{WL}$ is the throughput of workload (in I/O per second) and $AV_{DS}$ is the fractional availability of the dataset. Together with the annualized costs of repair during downtime (e.g. \$10000 per hour)

$$Cost_{downtime} = \frac{\$10000}{hr} * (1 - AV_{DS}) * \frac{8766}{1yr} \tag{3.18}$$

and the cost of losing the dataset (e.g. \$100M) scaled by the annual failure rate

$$Cost_{dataloss} = \$100M * AFR_{DS} \tag{3.19}$$

the utility can be calculated by

$$Utility = Revenue - Cost_{downtime} - Cost_{dataloss} \tag{3.20}$$

or with costs represented as negative utility ($Cost_{dataloos}$, $Cost_{downtime}$ have to be adjusted):

$$Utility = Revenue + Cost_{downtime} + Cost_{dataloss} \tag{3.21}$$

While this example is quite simple [STFG08] discusses more complex scenarios. In case of independent applications, the systems utility function could be summarize the utility of these applications. But with applications depending on each other, simple summation may not be appropriate. Instead the costs and benefits of the combined service could be examined.

## 3.4 Conclusion

The analysis of presented approaches shows two main strategies. The work in [MF11], [KM14] and [STFG08] faces the problem of breaking different aspect into one axis by calculating their financial impact. By contrast, [SP12], [YZL07], [Max05] and [Max05] calculates a ration for each aspect and expressing utility with a number that makes alternatives comparable.

[KM14] stands out because of the attempt to evaluate the users satisfaction with a monetary value. Formula 3.3 multiplies the request rate with benefit factor $d$ and the expected finish time with waiting cost factor $w$. These factors must not address a actual financial impact. This only ensures the comparability of an actual payment and the users rating in forula 3.4. [KM14] developed a method that enables the trade-off between e.g. cost and performance. The success of this approach strongly depends on the quality of the factors.

The approach in [MF11] only suggests factors with an comprehensible financial impact. It seems difficult to take non-financial factors into account. In this particular application, it's

not necessary. However [STFG08] shows that it is possible to value e.g. data safety with an amount of money by calculating the costs of data loss and the probability of data loss. This approach can be expanded to other requirements.

The approach in [SP12] doesn't utilize utility functions. However, the basic idea can be reused in utility functions. Each category gets a weight which expresses its impact on the decision. The same goes for attributes within the categories. Attributes are values within a easy scale. This makes different aspects comparable, surely not necessarily in a transparent, objective manner.

[YZL07], [Max05] and [HS10] doesn't calculate financial impacts. [YZL07] sidesteps the problem by subtracting the average value from every QoS attributes value and dividing the result with the standard deviation. Using the weights allows for a customized evaluation of alternatives. However, it is not possible to express the demand for the fulfilment of certain requirements with this utility function. Furthermore, the approach relies on already collected data (average and standard deviation for each QoS attribute).

[HS10] deals with the problem of uncertainty by including every possible outcome and the corresponding probabilities. Formula 3.16 calculates the expected value which taking all possible outcomes into account. This approach relies on a knowledge base, too. As with [YZL07], it is necessary to perform observations. [HS10] requires informations about the probability of different outcomes for each attribute and [YZL07] requires data to calculate the average and the standard deviation for each QoS attribute.

[Max05] and [HS10] using utility functions to transfer a quality's value into utility. What these utility functions all have in common is: $U(q) \rightarrow [0,1]$. A special characteristic of [Max05] is the handling of relationships between different QoS.

# 4 Concept and Specification

The aim of this chapter is to provide the conceptual foundations towards enabling the utility-based evaluation of the different deployment alternatives for cloud applications. Business and IT experts should be provided with decision support tools which aims at the selection of the optimal distribution of applications from the point of view of utility. For this purpose the framework must be able to handle at least application specific sub-graphs, regarded non-application specific sub-graphs and utility functions.

A concept for utility functions in the context of distributed computing is presented in section 4.1. Section 4.2 lists functional and non-functional requirements of the *utility calculation framework*. The following section 4.3 presents the framework's use cases. Finally, the comprehensive system in which the *utility calculation framework* is embedded is summarized in 4.4.

## 4.1 Utility Functions

The comprehensive range of Cloud offerings creates a set of alternative topologies for applications partially or completely hosted in the cloud. Each alternative provides different satisfaction for both, developers and business. Their satisfaction depends on the fulfilment of operational or business requirements. Utility functions can be used to value the satisfaction over time in order to compare the alternatives in an objective manner. Furthermore, the characteristics of the application workload can have a strong impact on the applications performance. The following concept includes these different factors to form the base for utility functions.

### 4.1.1 Concept

Calculating the utility of a viable topology determines the satisfaction created by the application under a certain distribution. Utility functions have the ability to combine the evaluation of different objectives (requirements) into one axis [STFG08].

Utility functions $U(T^i_\mu, W, R, T)$ depends on four parameters in this concept:

- $T^i_\mu$ a concrete $\mu$-distribution for the regarded application
- $W = \{w_0 ... w_m\}$ a set of workloads, either generated for time interval $T$ or observed during this interval.
- $R = \{r_0 ... r_n\}$ a set of requirements

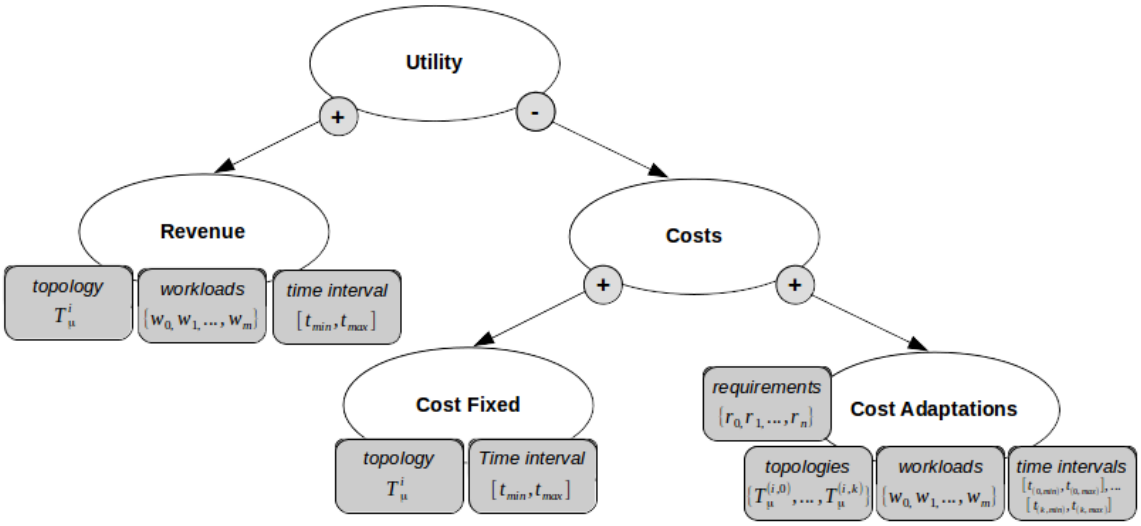


**Figure 4.1:** Utility Function: Concept Overview

- $T = [t_{min}, t_{max}]$ a time interval

The concept is focussed on the profitability of business applications, and therefore is based on the determination of the expected revenue $rev_{exp}(W, T)$ and the occurring cost $cost(T^i_\mu, W, R, T)$. The utility is the expected revenue minus the cost. Figure 4.1 shows an overview over the concept.

$$U(T^i_\mu, W, R, T) = rev_{exp}(T^i_\mu, W, T) - cost(T^i_\mu, W, R, T) \tag{4.1}$$

#### Revenue

The application's revenue depend on various factors which may vary over time. The approach here is to define a function for each factor which calculates the factor's value with respect to a time interval (e.g. month 1 of the application). Thus, it becomes possible to define a function $rev_{<unit>}$ which calculates the application's revenue per specified time unit with respect to a time interval $t$. *unit* has to be replaced with a time unit, e.g. $rev_{<month>}$ or $rev_{<day>}$. If $rev_{<unit>}$ is integrable, the definite integral on interval $[t_{min}, t_{max}]$ corresponds with the application's revenue in the same interval. Figure 4.2 shows the connection.

The *revenue per unit* $rev_{<unit>}$ can be calculated by:

$$rev_{<unit>}(T^i_\mu, W, t) = \sum_{j=0}^{j=m} p(w_j, t) * \overline{USER(t)} * \overline{TPU(w_j)} * \overline{RPT(t)} * \overline{sat(T^i_\mu, t)} * \overline{AV(T^i_\mu, t)} \tag{4.2}$$

$rev_{unit}$ is using the following factors and their functions:

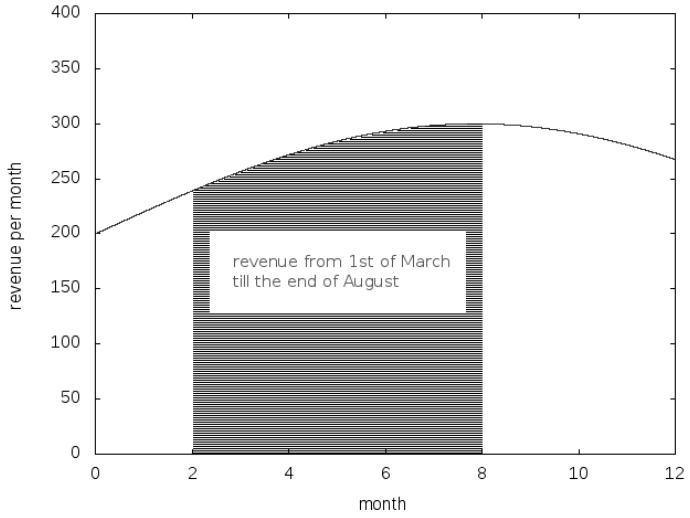


**Figure 4.2:** Example: Revenue per Month

Revenue per month and the revenue for a time interval

- $p(w_j, t)$ - *probability of receiving workload* $w_j$ *at time* $t$
- $\overline{TPU(w_j)}$ - *average number of transactions per user for workload* $w_j$
- $\overline{RPT(t)}$ - *average revenue per transactions at time* $t$
- $\overline{USER(t)}$ - *average number of users at time* $t$ *(e.g.* $\frac{users}{month}$*, or* $\frac{users}{day}$*)*
- $\overline{sat(T^i_\mu, t)}$ - *average user satisfaction at time* $t$ *of* $T^i_\mu$
- $\overline{AV(T^i_\mu, t)}$ - *average availability at time* $t$ *of* $T^i_\mu$

$\overline{sat(T^i_\mu, t)}$ returns the percentage of transactions which are aborted by users due to their dissatisfied experience under distribution $T^i_\mu$.

The integral over $rev_{<unit>}$ for a time interval $[t_{min}, t_{max}]$ defines $rev_{exp}(T^i_\mu, W, T)$ from equation 4.1:

$$rev_{exp}(T^i_\mu, W, T) = \int_{t_{min}}^{t_{max}} rev_{<unit>}(T^i_\mu, W, t)\, dt \tag{4.3}$$

#### Cost

Costs $cost(T^i_\mu, W, R, T)$ in 4.1 are calculated by the aggregation of costs for distribution $T^i_\mu$ for time interval $T$ and the adaptation costs to ensure the fulfilment of requirements $R =$

$\{r_0...r_n\}$ for time intervals $T_k$ in $[t_{min}, t_{max}]$.

$$\begin{aligned} cost(T_\mu^i, W, R, T) = \\ cost_{fixed}(T_\mu^i, T) \\ +[\sum_{k=1}^{o} cost_{adaptation}(T_\mu^{i,k}, W, R, T_k)] \end{aligned} \tag{4.4}$$

The first part $cost_{fixed}(T_\mu^i, T)$ can be evaluated with a tool like Nefolog. This corresponds with the sum of costs for Cloud offerings in $T_\mu^i$ for time interval $T$ and the defined hours of usage.

The second part is based on observations and/or predictions of workloads $\{w_0...w_m\}$. The assurance of the fulfilment of requirements $\{r_0...r_n\}$ may demand e.g. the horizontal or vertical scaling of Cloud instances, replicas, etc.. These adaptations occur costs at sub-intervals of $T$. $T_k$ represents one of these intervals. $T_\mu^{i,k}$ is the topology which arises from the adaptation of $T_\mu^i$ in time interval $T_k$.

### 4.1.2 Example

This example is based on assumptions. Two topologies $T_\mu^0$, $T_\mu^1$ come into consideration for hosting a web-shop application. $T_\mu^0$ relies on a Amazon EC2 m4.large instance, $T_\mu^1$ on a Amazon EC2 m4.xlarge instance. Both, a PHP application and a database is hosted on the respective instance with Ubuntu LTS 14.04 and a DMS.

General assumptions are:

- $W = \{w_0\}$
- $p(w_0, t) = 0.5$
- $R = \{r_1, r_2\}$
- Application is running from September till the end of the next January.
  Months: M8, M9, M10, M11, M12.
  $T = [M8, M12]$
- Hours of usage: 3600 (24/7)

Assumptions regarding $T_\mu^0$ and $T_\mu^1$ are summarized in tables and . The amounts for $cost_{fixed}(AV(T_\mu^0), T)$ and $cost_{fixed}(AV(T_\mu^1), T)$ are assumed and only coarse orientated at the actual costs[1].

The next assumption is that business experts have noted that *average revenue per transaction* varies over the year. From January till the May it is 0.10$ in average, the rest of the year it is

[1] https://aws.amazon.com/de/ec2/pricing/

| | |
|---|---|
| $\overline{sat(T_\mu^0, t)}$ | 0.9 |
| $\overline{AV(T_\mu^0)}$ | 0.99 |
| $cost_{fixed}(AV(T_\mu^0), T)$ | 600$ |
| $\sum_{k=1}^{o} cost_{adaptation}(T_\mu^{0,k}, W, R, T_k)$ | 390$ |

**Table 4.1:** Assumptions, $T_\mu^0$

| | |
|---|---|
| $\overline{sat(T_\mu^1, t)}$ | 0.85 |
| $\overline{AV(T_\mu^1)}$ | 0.95 |
| $cost_{fixed}(AV(T_\mu^1), T)$ | 700$ |
| $\sum_{k=1}^{o} cost_{adaptation}(T_\mu^{1,k}, W, R, T_k)$ | 50$ |

**Table 4.2:** Assumptions, $T_\mu^1$

0.15$.

$$\overline{RPT(t)} = \begin{cases} 0.10\frac{\$}{trans.} & t \bmod 12 < 5 \\ 0.15\frac{\$}{trans.} & t \bmod 12 \geq 5 \end{cases} \tag{4.5}$$

From November till the end of December 1000 users arrives per month in average, the rest of the year 700 users arrives per month.

$$\overline{USER(t)} = \begin{cases} 700\frac{users}{month} & t \bmod 12 < 10 \\ 1000\frac{users}{month} & t \bmod 12 \geq 10 \end{cases} \tag{4.6}$$

#### Revenue

Figure 4.3 shows $rev_{month}(T_\mu^0, W, t)$ and the area which corresponds with the revenue over time interval $[M8, M12]$. $rev_{month}(T_\mu^0, W, t)$ and $rev_{month}(T_\mu^1, W, t)$ aren't continuous functions. The definite integral may still be calculated. The integral can e.g. be approximated by step functions.

$$rev_{exp}(T_\mu^0, W, T) = \int_8^{13} rev_{month}(T_\mu^0, W, t) = 2067.12\$ \tag{4.7}$$

$$rev_{exp}(T_\mu^1, W, T) = \int_8^{13} rev_{month}(T_\mu^1, W, t) = 1873.40\$ \tag{4.8}$$

#### Cost

The example already includes the numbers for $cost_{fixed}(T_\mu^0, T)$, $cost_{fixed}(T_\mu^1, T)$, $\sum_{k=1}^{o} cost_{adaptation}(T_\mu^{0,k}, W, R, T_k)$, and $\sum_{k=1}^{o} cost_{adaptation}(T_\mu^{1,k}, W, R, T_k)$. Section 7.2 provides an example where these numbers are obtained by the analysis of occurring workload and the calculation with the use of the Nefolog cost-calculation framework.

$$cost(T_\mu^0, W, R, T) = 600.00\$ + 390.00\$ = 990.00\$ \tag{4.9}$$

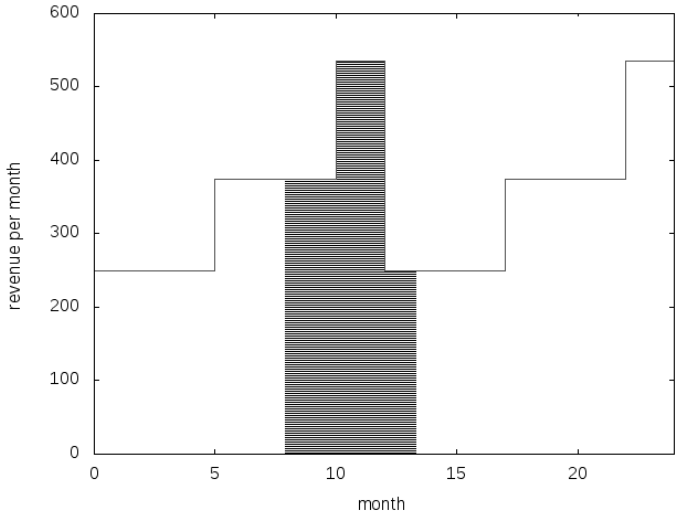


**Figure 4.3:** Web Shop: Revenue per Month

revenue per month for $T_\mu^0$ and the revenue for time interval $[8, 13]$ (striped)

$$cost(T_\mu^1, W, R, T) = 700.00\$ + 50.00\$ = 750.00\$ \tag{4.10}$$

#### Utility

Distribution $T_\mu^0$'s and $T_\mu^1$'s utility can be calculated with function 4.1:

$$U(T_\mu^0, W, R, T) = 2067.12\$ - 990.00\$ = 1077.12\$ \tag{4.11}$$

$$U(T_\mu^1, W, R, T) = 1873.40\$ - 750.00\$ = 1123.40\$ \tag{4.12}$$

Topology $T_\mu^0$ is promising the higher revenue and lower fixed costs for the distribution. Nevertheless, adaptation costs $T_\mu^0$ lead to higher utility under $T_\mu^1$.

## 4.2 Requirements

Based on the previous developed formalization of the utility-based approach, we focus in this section on analysing and extracting the requirements for the development of a framework capable of automating such calculations.

### 4.2.1 Functional Requirements

This section presents the functional requirements that the framework developed as part of this thesis should follow. More specifically, we extract the requirements and organize them based on the most relevant functionalities that these should support.

#### Repository

- **FR1:** The framework must provide a repository of applications, viable distributions, utility functions, functions and requirements (hereinafter called "resources").
- **FR2:** Framework users must be able to add resources, update existing resources, expand resources with meta-data which describes the resource in more detail and delete existing resources from the repository.
- **FR3:** Users must be able to browse the repository, locate existing resources, look at resources' representations, meta-data, identifiers and relationships to other resources.
- **FR4:** The repository must provide a pool of reusable utility functions.
- **FR5:** The repository must provide a pool of reusable functions of different types (cost- and revenue-functions) to compose customized utility functions.

#### Resources

- **FR6:** It shall be possible to add, update and delete relationships between different resources.
- **FR7:** Each resource must be provided with a *Universally Unique IDentifier* (hereinafter called UUID).
- **FR8:** Each resource can be clearly identified based on a *Uniform Resource Identifier* (URI).
- **FR9:** Distribution resources must be support a common topology language. The framework must be able to collect included Cloud offerings automatically. Therefore the framework must have knowledge about Cloud providers and their offerings (including pricing models).
- **FR10:** The repository must support meta-data for parameters occurring in (utility) functions.
- **FR11:** Users can request a list of parameters which occurs in a utility function and the describing meta-data.

#### Utility

- **FR12:** The framework has the ability to calculate the result of a (utility) function in the repository based on a given parameter assignment.
- **FR13:** The framework must offer the possibility to calculate the utility of different viable distributions and thereby provide decision support.
- **FR14:** The framework should be able to select the optimal distribution among alternatives based on utility functions.

#### Operability

- **FR15:** Users must be able to define functions for the system without the need for a certain machine-readable format.
- **FR16:** The framework should provide a loosely coupled application programming interface to the repository.
- **FR17:** The framework should include a authorisation procedure to lock unauthorized users from operations which have an impact on resources.

### 4.2.2 Non-Functional Requirements

The taxonomy suggested in [GB08] contains 16 non-functional requirements (C3.1 - C3.16) and corresponding metrics. This taxonomy serves as a basis for the definition of non-functional requirements.

**Usability** The framework should provide well documented, uniform interfaces. Users should be informed about operating errors to increase the number of functions understood. Graphical user interfaces should be understandable and intuitive to use.

**Reliability** The framework should keeping operating over time. Therefore, the framework should run on a proven environment, reliable infrastructure and utilizing Cloud services from reliable providers. Program logic should be transparency and well proven.

**Performance** Services should provide a acceptable response time (with respect to the users' perception).

**Safety** If services are used improperly or program logic is incorrect, then calculations could harm business (wrong decisions based on wrong numbers). Therefore, the system must provide high usability and program logic should be transparency and well proven.

**Security** The framework should protect data privacy, should prevent the unauthorized access to resources and the unauthorized modification of stored data.

**Accessibility** The framework should provide clearly defined interfaces tailored to a wide target group. Functionality should be offered over different interfaces.

**Interoperability** The system should support common data formats to increase interoperability.

**Availability** Unplanned downtime should be prevented. Therefore, the system should rely on components with high availability. At least *Availability Environment Classification* AEC-2 should be aimed.

**Extensibility** System components should be extended without effecting other components. The design should support extensibility.

**Testability** Services should support the measurement of test criteria.

**Modifiability** Services should be modified without effecting other services.

## 4.3 Use Cases

Two kinds of actors using the *Utility Calculation Framework*. These are IT experts and business experts. The *Utility Calculation Framework* provides the utility-based evaluation of the different deployment alternatives for cloud applications. The use of the system requires the access to other systems. These are the *Cost Calculation Framework* and the *Topology Modeler*. Both systems are describes in 4.4. Eight use cases are identified, which are described in the following section. Figure 4.4 given an overview over the mentioned use cases.

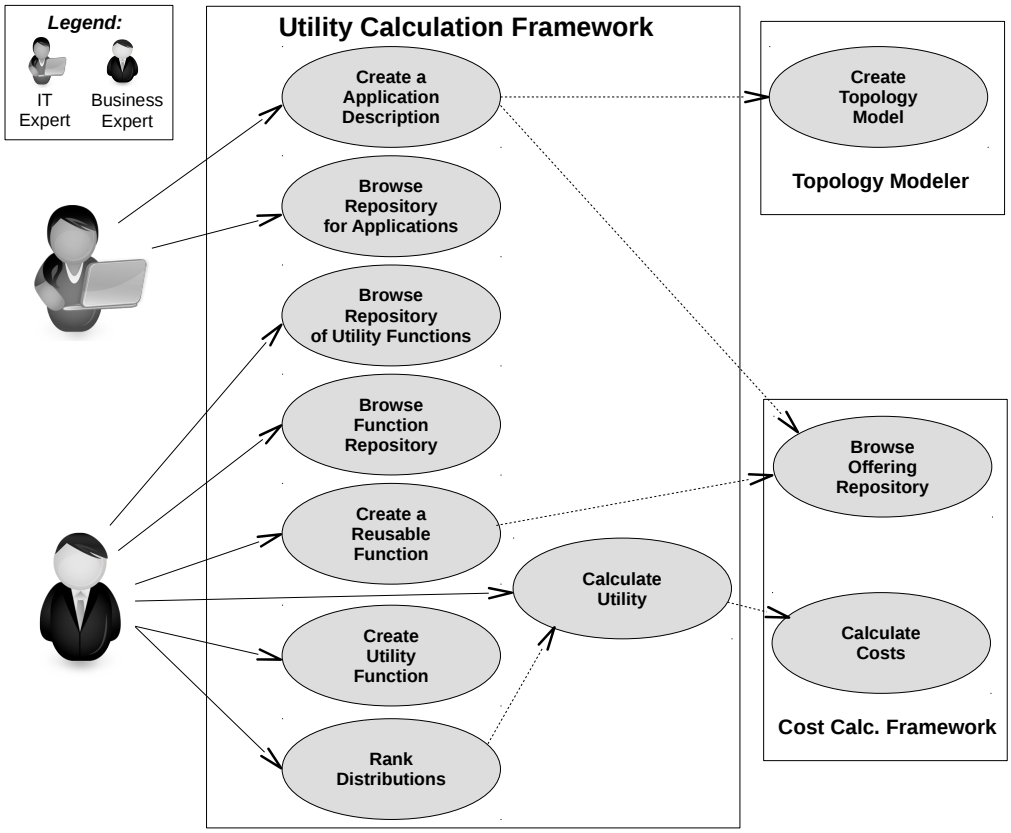


**Figure 4.4:** Use Case Diagram

### 4.3.1 Use Cases Description

| Name | **Browse Repository for Applications** |
|---|---|
| Goal | The IT expert is getting an overview over existing applications, requirements, possible viable distributions and utility functions. |
| Actor | IT expert |
| Pre-Condition | |
| Post-Condition | The system retrieves and provides the knowledge about existing applications in the repository, requirements, possible viable distributions and utility functions. |
| Post-Condition in Special Case | |
| Normal Case | 1. The actor requests for a list of existing applications. The actor can also request for applications of a specific type.<br>2. The system delivers a list with the applications.<br>3. The actors goes through the listed applications and request for related requirements, viable distributions and utility functions.<br>4. The system delivers the connected resources. |
| Special Cases | 2a. The repository doesn't contain any application.<br>a) The system returns empty list.<br>b) Terminate<br>4a. The repository doesn't contain related resources.<br>a) The system returns empty list. |

**Table 4.3:** Description of Use Case *Browse Repository for Applications*.

| Name | **Browse Repository of Utility Functions** |
|---|---|
| Goal | The actor is getting an overview over existing utility functions and their composition. |
| Actor | Business expert |
| Pre-Condition | |
| Post-Condition | The system retrieves and provides the knowledge about existing utility functions and their composition. |
| Post-Condition in Special Case | |
| Normal Case | 1. The actor requests for a list of existing utility functions. The actor can also request for utility functions which are already in use for a application of a specific application type.<br>2. The system delivers a list with the utility functions.<br>3. The actors goes through the listed utility functions and requests for their sub-functions.<br>4. The system delivers the sub-functions.<br>5. The actor examines the sub-functions, identifies reused functions, parameters and concepts. |
| Special Cases | 2a. The repository doesn't contain any utility function.<br>a) The system returns empty list.<br>b) Terminate |

**Table 4.4:** Description of Use Case *Browse Repository of Utility Functions*.

| Name | **Browse Function Repository** |
|---|---|
| Goal | The actor is getting an overview over existing functions, their parameters and type (e.g. revenue or cost). |
| Actor | Business expert |
| Pre-Condition | |
| Post-Condition | The system retrieves and provides the knowledge about existing functions. |
| Post-Condition in Special Case | |
| Normal Case | 1. The actor requests for a list of existing function. The actor can request for functions of a specific function type.<br>2. The system delivers a list with the requested functions.<br>3. The actors goes through the listed function and request for their parameters.<br>4. The system delivers the parameters. |
| Special Cases | 2a. The repository doesn't contain any function.<br>a) The system returns empty list.<br>b) Terminate<br>4a. The repository doesn't contain related resources.<br>a) The system returns empty list. |

**Table 4.5:** Description of Use Case *Browse Function Repository*.

| Name | **Create a Reusable Function** |
|---|---|
| Goal | The actor has developed a function and wants to add this function to the repository for reuse in customized utility functions. |
| Actor | Business expert |
| Pre-Condition | |
| Post-Condition | The repository contains a new function, additional meta-data and the set of occurring parameters. |
| Post-Condition in Special Case | The system returns a meaningful error message. |
| Normal Case | 1. The user passing the function, meta-data and the list of occurring parameters to the system.<br>2. The system is checking the consistency of function and parameters.<br>3. The system persists function and parameters.<br>4. The system delivers the URI of the created function to the actor. |
| Special Cases | 2a. The input isn't consistent, e.g. missing parameters.<br>a) System returns a meaningful error message.<br>b) Terminate<br>3a. The persistence of function and/or parameters failed.<br>a) System returns a meaningful error message.<br>b) Terminate |

**Table 4.6:** Description of Use Case *Create a Reusable Function*.

| Name | **Create a Application Description** |
|---|---|
| Goal | Application resources are wide-ranging. The use case includes the creation of the application resource and also the creation of requirement resources and resources for viable distributions. |
| Actor | IT expert |
| Pre-Condition | |
| Post-Condition | The repository contains a new application which representing the application, containing additional meta-data, connected requirements and viable distributions. |
| Post-Condition in Special Case | |
| Normal Case | 1. The actor requests for the creation of a new application.<br>2. The system creates the resource and returns the regarded URI.<br>3. The actor requests for the creation of requirements.<br>4. The system creates the requirements and returns the regarded URIs.<br>5. The actor is using the *topology modeler* to get models of the viable distributions.<br>6. The actor requests for the creation of viable distributions.<br>7. The system creates the viable distributions and returns the regarded URIs. |
| Special Cases | 2a. The creation fails.<br>a) The system returns a meaningful error message.<br>b) Terminate.<br>4a. The creation fails.<br>a) The system returns a meaningful error message.<br>b) Terminate.<br>7a. he creation fails.<br>a) The system returns a meaningful error message.<br>b) Terminate. |

**Table 4.7:** Description of Use Case *Create a Application Description*.

| Name | **Create Utility Function** |
|---|---|
| Goal | The actor has developed a utility function and wants to add this utility function to the repository and connect it to a existing viable distribution (or more than one). |
| Actor | Business expert |
| Pre-Condition | The reused functions are already stored in the repository. |
| Post-Condition | The repository contains a new utility function. |
| Post-Condition in Special Case | |
| Normal Case | 1. The actor browses the application repository to identify the regarded viable distribution(s).<br>2. The actor browses the function repository to identify reusable functions.<br>3. The actor browses the offering repository of the *cost calculation framework* to identify in the distribution contained offerings and configurations.<br>4. The actor passes the utility-function and meta-data to the system.<br>5. The system persists the utility function.<br>6. The system delivers the URI of the created utility function. |
| Special Cases | 1a. The repository is misses the distribution.<br>a) The actor creates the missing topology resources and repeats.<br>2a. The actor misses a function.<br>a) The actor creates the missing function resources and repeats.<br>3a. The *cost calculation framework* doesn't contain the offering or configuration.<br>a) The actor can not use the functionalities of the system and have to calculate distribution costs based on his own functions.<br>5a. The storage of the resource failed.<br>a) The system returns a meaningful error message.<br>b) Terminate |

**Table 4.8:** Description of Use Case *Create Utility Function*.

| Name | **Calculate Utility** |
|---|---|
| Goal | The actor wants to calculate the utility of a viable distribution |
| Actor | Business and it experts |
| Pre-Condition | Utility functions and the application description are stored in the repository |
| Post-Condition | Parameter assignments are stored for further use. |
| Post-Condition in Special Case | Missing application description and/or distribution are created. |
| Normal Case | 1. Actor browses the application repository to identify the regarded distribution.<br>2. Actor browses the repository to identify the regarded utility function.<br>3. Actor assigns values to parameters occurring in the utility function.<br>4. The system collects the stored assignments.<br>5. The system proofs if the assignment fits to the utility function.<br>6. The system uses the collected assignments to calculate the utility. |
| Special Cases | 1a. The repository is missing the application.<br>a) The IT expert has to create the application.<br>1b. The application resource doesn't contain the distribution.<br>a) The IT expert has to create the distribution.<br>2a. The repository is missing the utility function.<br>a) The Business expert has to create the utility function.<br>4a. No assignment found.<br>a) The system returns a meaningful error message.<br>b) Terminate<br>5a. The assignment contains incorrect values or is incomplete.<br>a) The system returns a meaningful error message.<br>b) Terminate<br>6a. The calculation fails.<br>a) The system returns a meaningful error message.<br>b) Terminate |

**Table 4.9:** Description of Use Case *Calculate Utility*.

| Name | **Rank Distributions** |
|---|---|
| Goal | The actors want a list of possible distributions ranked by the utility |
| Actor | Business and IT expert |
| Pre-Condition | Necessary values are already stored (assignment of values to occurring parameters) |
| Post-Condition | |
| Post-Condition in Special Case | Missing application description and/or connected distributions and utility functions are created. |
| Normal Case | 1. The actor browses the application repository to identify the application and possible distributions.<br>2. The actor is browsing the repository to identify the regarded utility functions.<br>3. The system collects already stored assignments (pre-condition) for each distribution.<br>4. The system proofs if the assignments fits to the utility functions.<br>5. The system uses the collected assignments to calculate the utility for each distribution.<br>6. The system ranks the distributions with respect to their utility and returns the result. |
| Special Cases | 1a. The repository is missing the application.<br>a) The IT expert has to create the application.<br>1b. The application doesn't contain viable distributions.<br>a) The IT expert has to create the distributions.<br>b) terminate<br>2a. The repository is missing the utility function.<br>a) The business expert has to create the utility function.<br>3a. No assignment found.<br>a) The system returns a meaningful error message.<br>b) Terminate<br>4a. The assignment contains incorrect values or is incomplete.<br>a) The system returns a meaningful error message.<br>b) Terminate |

**Table 4.10:** Description of Use Case *Rank Distributions*.

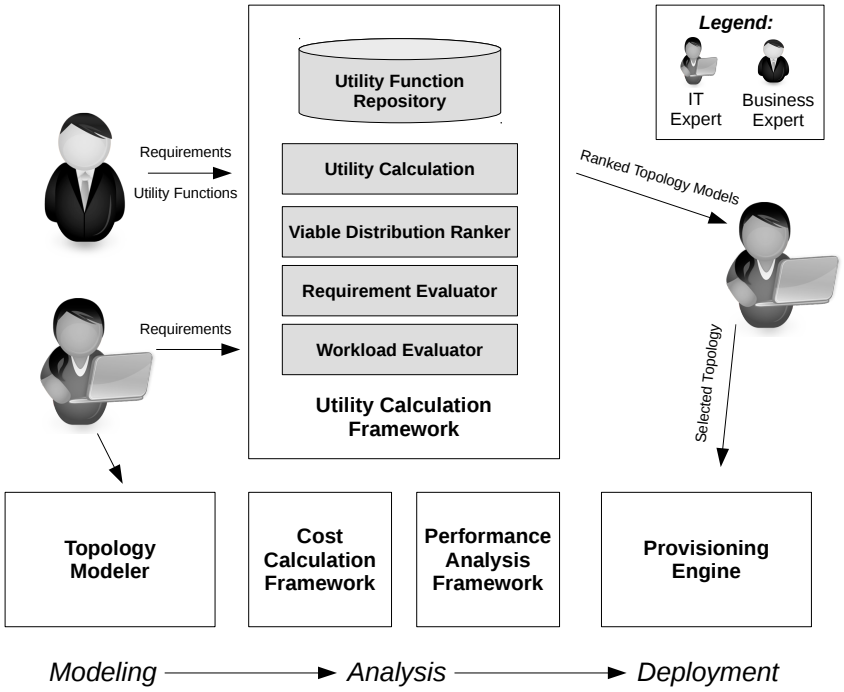


**Figure 4.5:** System Overview

## 4.4 System Overview

Figure 4.5 outlines the system in which business and IT experts acts to deploy a distributed application in the way which creates the highest utility.

### 4.4.1 Topology Modeler

IT experts using the *Topology Modeler* component to create models in a topology language (e.g. TOSCA) and related management plans. Thereby, several viable distributions for the same application held ready and can be deployed. The *Utility Calculation Framework* also uses these models.

### 4.4.2 Utility Calculation Framework

Both, business and IT experts, uses the *Utility Calculation Framework* to formulate functional and non-functional requirements for the application and components of the application.

The *Workload Evaluator* component evaluates available viable distributions under already observed or predicted workload using the *Performance Analysis Framework*. The *Requirement Evaluator* component uses the results to show how requirements are fulfilled. The need for (temporarily) adaptations of topologies can be determined.

Business experts use the *Utility Function Repository* to add new utility functions or select already existing utility functions for evaluations. The fixed costs of a regarded distributions can be determined with the *Cost Calculation Framework*. With the results of the workload evaluator, calculation of the expected revenue, the already calculated costs, the knowledge about necessary adaptations, known or predicted ratios, the utility of different viable distributions can be determined. The *Viable Distribution Ranker* component uses the *Utility Calculation* component to create a ranking of possible distributions with respect to their utility.

### 4.4.3 Provisioning Engine

The *Utility Calculation Framework*'s ranking allows for the deployment of an application under the optimal distribution (among evaluated viable distributions with respect to their utility). Application can be deployed by IT experts or even automatically in the *Provisioning Engine*.

# 5 Design

This chapter describes the design for the implementation of the *Utility Calculation Framework*. The architecture, the resource model, the components of the framework and the database's entity-relationship model are presented.

## 5.1 Architecture

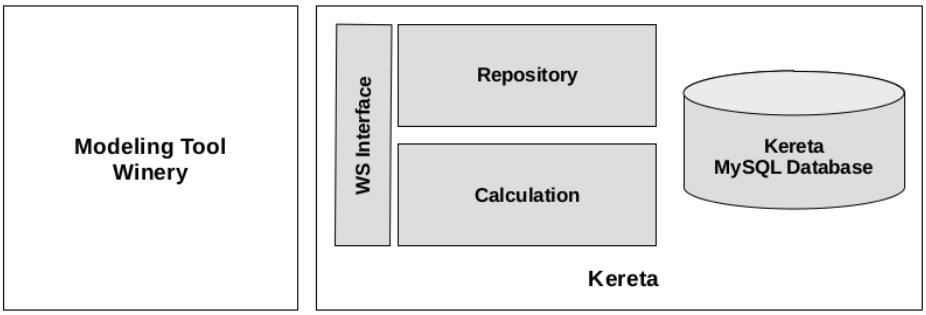


**Figure 5.1:** Design Overview

The previous introduced system overview (see section 4.4) contains the *Viable Distribution Ranker*, the *Utility Calculation* and the *Utility Function Repository*. These components of the *Utility Calculation Framework* are the parts of the framework which is designed in this chapter.

Figure 5.1 gives the design overview. The first module is Kereta. Kereta contains the *Repository* and the *Calculation* component. The *repository* allows for the management of resources described in the resource model (see section 5.2) and the *calculation* component extends the *repository* to enable the calculation of utility. Kereta uses a MySQL database to persist resources. The database is described in section 6.1.3. A REST API provides a standardized interface to the *Repository* and the utility calculation.

The Winery modeling tool has to be extended. The extension provides a convenient graphical user interface to the Kereta module and embeds its functionalities into the larger scope of Winery.

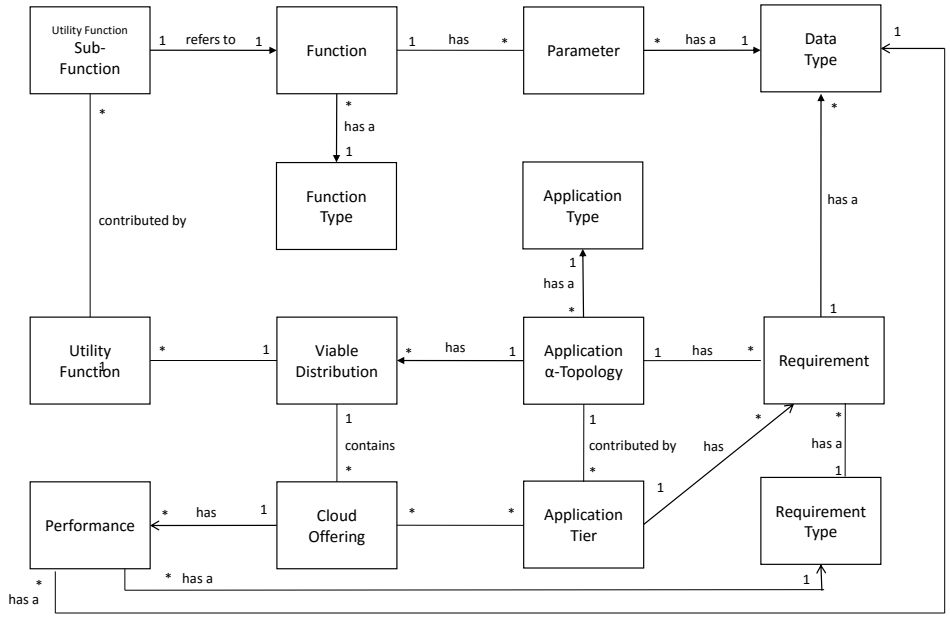


**Figure 5.2:** Resource model

## 5.2 Resource Model

The resource model (figure 5.2) summarizes the resources which are relevant to the utility based decision support. The resources' meaning and description is explained below.

### 5.2.1 Application-specific Topology

The *Application-specific Topology* is a description of the application independent of a concrete distribution. Possible viable topologies are represented by connected *Application Topologies*. Requirements which addressing the outcome of the comprehensive application are also connected. Different tiers of the application are represented by *Application-specific Components*.

One point to note is that [AGSLW14] mentioned that it is possible to move the whole subgraph of a *Application-specific Component* to the $\alpha$-topology. This indicates that the implementation requires a concrete hosting. This case is not considered in order to ease the resource model. This does not rise to restrictions in the *utility calculation framework*.

### 5.2.2 Application-specific Component

Multi-tiered applications can be seen as the aggregation of application-specific components [AGSLW14]. Figure 5.3 contains the MediaWiki's $\alpha$-topology. *MWiki_Front* and *MWiki_-DB* are application-specific components of the topology. These components are support by middleware solutions [AGSLW14] in the $\gamma$-topology of possible viable topologies.

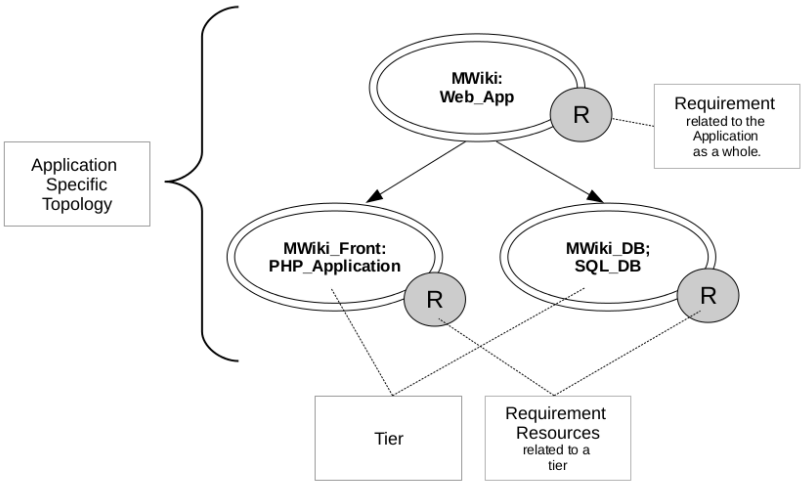


**Figure 5.3:** MediaWiki: $\alpha$-Topology and Requirements

Not only *Application-specific Topologies* can have *Requirements*. Tiers can have *Requirements*, too. Therefore, a *Application-specific Component* can be linked to *Requirements*. This enables the evaluation of the fulfilment of tier-specific requirements.

### 5.2.3 Requirement

*Requirements* can be either of type *functional* or *non-functional*. *Requirements* addressing the entire application or a specific tier. Requirements have a value and the the information how to interpret it. In the *utility calculation framework* interpretation can be *equal*, *lower* or *higher*. For instance: "location equal to EU", latency lower than 2000ms or availability higher than 99.99%.

### 5.2.4 Application Topology

A *Application-specific Topology* is restricted to the application's $\alpha$-topology. However, the *Application Topology* summarized the $\alpha$-topology and the reusable $\gamma$-topology in the $\mu$-topology.

*Application Topologies* contains *Application Subgraphs*. Existing *Utility Functions* are linked to *Application Topologies*.

### 5.2.5 Application Subgraph

A *Application Subgraph* contains the middleware solution which supports a *Application-specific Component* and the infrastructure in which it is deployed and provisioned [AGSLW14].

The *Performance* of *Application Subgraphs* can be measured or assumed. The *resource model* allows for the comparison of *Requirements* and *Performance*. For this purpose *Application Subgraphs* can be linked to a *Application-specific Component*.

### 5.2.6 Performance

*Application Subgraphs* have functional and non-functional performance attributes. Performance attributes can be compared with requirements like mentioned before. *Performance* includes a value and a interpretation analogous to *Requirements*.

### 5.2.7 Function

The *resource model* defines a *Function* as a reusable component of a *Utility Function* or another *Function*. *Functions* can be reused by different *Utility Functions* and called from various *Functions*. Links to *Utility Functions* are defined as *Utility Function Sub-Functions*.

*Functions* have a *Function Type* and join a set of *Parameters*.

### 5.2.8 Parameter

*Functions* are connected to the set of *Parameters* which occur in the function. These parameters are restricted to a defined set of *Data Types*.

### 5.2.9 Utility Function

A concept for *Utility Functions* is defined in 4.1. The concept includes a revenue and a cost function. The cost function itself is composed by a function for the fixed costs of the distribution and a function for adaptation costs. The *resource model* allows for the definition of *Utility Functions* with any number of sub-functions. In order to achieve reusability, *Functions* are independent of a concrete *Utility Functions* and can be linked by *Utility Function Sub-Functions* to *Utility Functions*.

### 5.2.10 Utility Function Sub-Function

*Utility Function Sub-Functions* connect *Utility Functions* to *Functions*.

### 5.2.11 Types

#### Application Type

Users can define application types. This ease the search for applications of the same type, their utility functions and function resources.

#### Requirement Type

*Requirements* can be either of type *functional* or *non-functional*.

#### Function Type

The utility function concept (see 4.1.1) suggests two function types: *revenue* and *cost*. Another type is predefined: *misc* (miscellaneous). Users are allowed to define additional function types.

The *resource model* shows that *Utility Function Sub-Functions* refer to *Functions*. These references are only work as intended in the *utility calculation framework* when the function type is *revenue* or *cost*.

As also indicated in the resource model, *Functions* can call *Functions*. In doing so, the function type doesn't matter.

#### Data Type

Requirement, performance and parameter resources have a *Data Type*. Predefined *Data Types* are *string*, *number*, *array of strings*, *array of numbers* and *array of arrays*.

The array types allow for values like, e.g. $[1, -2, 1.2]$, $[''a'', ''abc'', '''']$ or even nested constructions like $[[1, -2, 1.2], [3, 2], [-45, 97, 0, 1.1]]$

## 5.3 Kereta Repository

The resources from the model in figure 5.2 should be managed by the repository. For easy handling the resources have shorter names in the repository. The REST API uses the shorter names. Table 5.1 shows the names from the resource model and the names actually used in the repository. The table also shows which database tables are used to persist the resources. Resource names in the repository are singular and written in camel case (hereafter: resource-nouns).

| Resource Model | REST API | Database Table |
|---|---|---|
| Application-specific Topology | Application | kereta_application |
| Application-specific Component | Tier | kereta_application |
| Application Distribution | Distribution | kereta_distribution |
| Application Subgraph | Offering | kereta_offering |
| Requirement | Requirement | kereta_requirement |
| Performance | Performance | kereta_performance |
| Function | Function | kereta_function |
| Parameter | Parameter | kereta_parameter |
| Utility Function | UtilityFunction | kereta_utilityFunction |
| Utility Function Sub-Function | SubFunction | kereta_subFunction |
| ApplicationType | ApplicationType | kereta_applicationType |
| RequirementType | RequirementType | kereta_requirementType |
| FunctionType | FunctionType | kereta_functionType |
| DataType | DataType | kereta_dataType |

**Table 5.1:** Terminology in the Design

Relations between the terminology in the resource model, the REST API and database tables.

### 5.3.1 Nesting

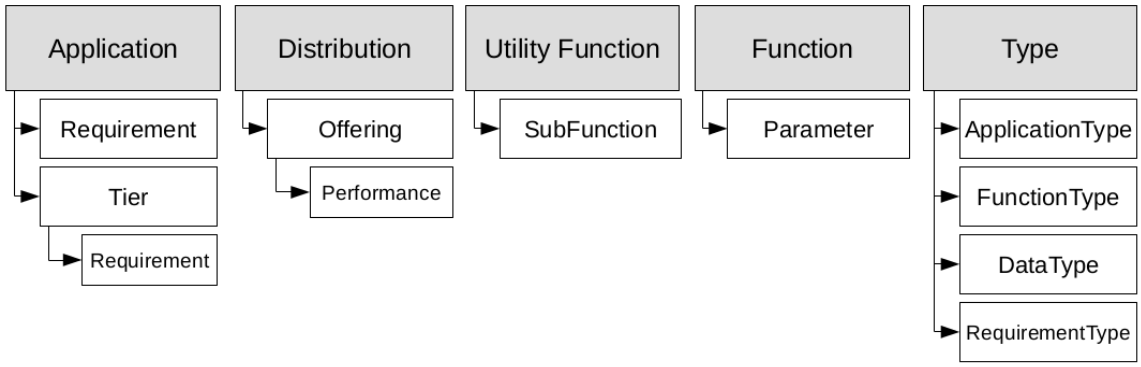


**Figure 5.4:** Resource Nesting

The repository should provide a convenient and intuitive REST-API to its resources and functionalities. A fundamental decision is the nesting of resources. The repository is nesting resources under four root resources and the category *Type*. Figure 5.4 shows the structure: *Application*, *Distribution*, *UtilityFunction* and *Function* are root resources of the repository. Other resources are accessible inside the superior resource. The nesting doesn't reflect all relationships, e.g. a *Distribution* resource is related to an *Application* resource. To avoid lengthy URIs, the HATEOAS constraint is used to reflect relationships not covered by the nesting. Each resource representation contains a *links*-section for these relations.

### 5.3.2 Identifiers

Root resources are provided with an 36 character long *universally unique identifier* (UUID). The remaining resources are nested within these *root resources* and are clearly addressable via the combination of the root resource's UUID and the identifier of the nested resource, e.g. */Function/0195448d-424e-4aca-b0ea-cb8442f4adf2/Parameter/a*. The root resources' UUIDs are assigned by the system.

When creating a root resource it isn't possible to force a specific UUID. In contrast, identifiers of nested resources must be defined by the API users. These user-defined identifiers must be unique within the nested resources (of the same type) of a specific root resource. The aim is a taxonomy which achieves both, clearly identified resources and provides human-user friendly and intuitive understandable URIs.

Since human users may struggle with 36 characters long UUIDs, it is possible to define a at most 8 character long alias for root resources. Aliases must be unique within the resource of the same type (siblings). Aliases allows for URIs like */Function/myFct* or even */Function/myFct/Parameter/a*. It is allowed to use letters (lower and upper case), numbers, "" and "_" in aliases. *Function* resources' aliases are restricted to letters (lower and upper case) and "_", otherwise mathematical expressions including function-calls may be misinterpreted.

### 5.3.3 Methods

The REST API provides the HTTP-methods GET, POST, PUT and DELETE without exception for each resource type. The methods act in a standard manner which is defined below. GET-, POST and PUT-methods response with a XML-representation of the suggested resource and a status code. DELETE-methods response only with a status code.

The REST-API uses HTTP status codes following the HTTP/1.1 standard[1]. Table 5.2 contains the status codes which can be returned.

| status code | description |
|---|---|
| 200 | OK |
| 201 | Created |
| 204 | No Content |
| 403 | Forbidden |
| 404 | Not Found |
| 405 | Method not Allowed |
| 406 | Not Acceptable |
| 409 | Conflict |
| 500 | Internal Server Error |

**Table 5.2:** HTTP status codes - HTTP/1.1 standard

[1]RFC 7231, 8.2.3. Registrations

For a more convenient debugging the REST API returns 4xx-codes with a XML-representation of the error-message. For instance, the following XML-document is returned if a resource is not found.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <error>
3   <message>resource not found</message>
4 </error>
```

**Listing 5.1:** Error Message - XML representation

Status codes can be evaluated in embedded solutions but human users may prefer textual content, especially when debugging in case of failures.

#### GET

The GET-method returns a set of resources, a specific resource or the result of a function (see section 5.3.5). If the URI identifies a specific resource, the REST API returns the XML-representation of the resource (e.g. */Function/myFct* or */Function/myFct/Parameter/a*). If the URI doesn't aim at a specific resource, a XML-document is returned which document element contains the set of regarded resources (e.g. */Function* or */Function/myFct/Parameter*).

Functions return their result in XML-documents, too. The structure of these documents is shown in section 6.1.2. URL parameters are used to pass parameters to functions supported by the REST API. The usage is defined in 5.3.5).

#### POST

The specification of the usage of UUIDs and user-defined identifiers has an impact on how different resources are created. Root resources are created by calling the POST-method for URI */{ressource-noun}*. The REST-API creates a resource and returns the XML-representation of this resource, which contains the automatically assigned UUID. Nested resources are created by calling the POST-method for the URI which includes the user-assigned identifier, e.g. */Function/myFct/Parameter/a*. If the identifier isn't unique within its siblings, the creation of the resource fails.

POST-methods neither expect nor proceed any input. POST-calls just result in the creation of a resource with the automatically assigned UUID or the user-defined identifier and the UUID of the primary *root resource*. In the single case where a resource is nested within in a nested resource, the identifier of the resource directly above and the *root resource*'s UUID is stored.

#### PUT

PUT-methods expect a (partially) XML-representation of the addressed resource. PUT-methods are not authorized to change properties, which have an impact on the resources URIs. POST-methods finally specify these parameters.

The document-element of the passed XML-representation must have the tag-name corresponding to the resource type (small letters and camel-case, e.g. *subFunction*). The same also goes for properties.

As stated in 4.2 each resource must be provided with an UUID and consequently clearly identified with this UUID. This functional requirement has been weakened to get more comfortable URIs. Resources of selected types (hereinafter called *root resources*) are still provided with an UUID, but every other resource is identified with respect to a root resource. For instance, a *function* resource is a root resource, *parameter* resources

#### DELETE

To limit the damaging consequences of incorrect use, the DELETE-method is only implemented for single resources. It isn't possible to call the DELETE-method for a set of resource, e.g. ".../Application". The only way to delete sets of resources is to delete each resource separately, e.g.*.../Application/{application-id}*. Deleting a resource triggers the deletion of nested resources.

DELETE-methods neither expect nor proceed any input. DELETE-calls just result in the deletion of the identified resource and its nested resources.

### 5.3.4 Representation

The REST API returns XML-representations of resources. Section 6.1 contains the representations for all resource. The GET-method for the root directory .../ returns a HTML document which contains a short description of the repository and an URI-overview.

### 5.3.5 Repository Functionality

The repository provides the management of resources. Besides that additional functionality must be implemented. On the one hand it serves for the evaluation of application distributions and on the other hand it increases the repository's reusability.

#### Reusability

Users can profit from already created resources, if the REST API provides functions which allow for the targeted search. In order to ease the reuse of function resources and the reproduction of utility function resources, it should be possible to search for these resources. In addition the repository should provide functionality to clone utility functions. The REST API provides these functions with URIs */Search/...* and */Clone/....*

**/Search?resource=Application&applicationType=[appType]** The URI parameter specifies the applications' type. The function returns all applications in the repository if [appType] is empty - otherwise only applications with the passed type are returned. The regarded *application type* must be exist.

**/Search?resource=UtilityFunction&applicationType=[appType]** The function returns all utility functions in the repository if [appType] is empty - otherwise only utility functions from applications with the passed type are returned. The regarded *application type* must be exist.

**/Search?resource=Function&applicationType=[appType]&functionType=[fctType]** It may helpful to see which functions are reused in utility functions of a specific application type. To allow for a targeted search, the function type (typically *revenue* or *cost*) is part of the URL parameters.

**/Search?resource=Function&functionType=[fctType]** The function allows for the search of *function* resources with a specific *function type* (typically *revenue* or *cost*).

/UtilityFunction/myUF/clone?query

**/UtilityFunction/{uf-id-or-alias}/clone?distributionId=[distr-id-or-alias]** The function clones the identified *utility function* resource and nested *(utility function) sub-function* resources and connect these resources to the identified *distribution* resource.

#### Evaluation

Evaluation aims at different resource types. It must be possible to evaluate *function*, *sub-function* and *utility function* resources for a defined parameter assignment. In cases of function resources, parameter assignment is done by a JSON object. Sub-function resources must allow the persistence of assignments. Calculation for sub-function and utility function resources must be provided by referencing a persisted assignment.

Assigned values are not stored in the database. A folder *parms* is create in the applications server's domain. The repository creates a XML file for a *utility function* resource (at the

moment of the first assignment). The following XML document shows a example for these files:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<assignments
  utilityFunction="ea324fa8-3aa2-49b6-9437-951df829a321">
  <subfunction number="1">
    <parms id="key1">
      <t>[4,10]</t>
      <j>[0,2]</j>
    </parms>
    <parms id="key2">
      <t>[1,3]</t>
      <j>[0,2]</j>
    </parms>
  </subfunction>
  <subfunction number="2">
    <parms id="key1">
      <Month>6</Month>
      <Hour>3000</Hour>
      <location_zone>EU</location_zone>
    </parms>
  </subfunction>
</assignments>
```

**Listing 5.2:** XML parameter assignment storage file

Users can assign values under different keys for the same sub-function of a utility function. The assignments can be reused be referencing the key.

**/Function/[fct-id-or-alias]/calc?[assignment]** This function calculates the result for a given assignment. [assignment] contains a URL parameter for each parameter occurring in the identified *function* resource. The usage is shown in section 6.1.2.

**/UtilityFunction/[uf-id-or-alias]/SubFunction/[sf-nbr]/assign?key=[key]&[assignment]** The function persists the assignment for the identified *(utility function) sub-function* resource under the defined *key*. The usage is shown in section 6.1.2.

**/UtilityFunction/[uf-id-or-alias]/SubFunction/[sf-nbr]/calc?key=[key]** The function requests the persistent assignment for for the identified *(utility function) sub-function* resource under *key*. The function calculates and returns the result. The usage is shown in section 6.1.2.

**/UtilityFunction/[uf-id-or-alias]/calc?key=[key]** The function requests the persistent assignment for each *(utility function) sub-function* resource of the identified *utility function* resource under *key*. The function determines each sub-result and *function type* (*revenue* or *cost*), then the function calculates and returns the utility.

#### Decision Support

*Application-specific topoplogy* resources can offer several viable distributions. Users should request decision support and get the *application distribution* resource which promises the highest utility or a overview about distributions and their expected utilities.

**/Application/[app-id-or-alias]/select?[query]** String [query] contains all suggested distributions and utility functions identifiers the keys for assignments for each distribution. The concrete construction is explained in section 6.1.2. The function determines each distributions utility and returned the distribution which promises the highest utility.

**/Application/[app-id-or-alias]/rank?[query]** This Function behaves like the previous *select*-function except the fact that it returns the ranked list of all suggested distributions.

## 5.4 Kereta Database

Kereta is using a MySQL database. Figure 5.5 shows the entity-relationship model of the database. Tables are named starting with "*kereta_*" and a name that identifies the reflected resource type, e.g. *kereta_utilityFunction*. Column names don't contain a prefix and composed names are separated by "_", e.g. *application_id*.

There is table for each resource type of the repository with one exception. Requirement resources can be related to application or tier resources (see 5.2.3). Thus, *application specific topology* and *application specific component* resources are stored in the same table (*kereta_application*). In cases of a *application specific topology* resources the *application_tier* attribute is 0. In cases of tier resources this attribute is $\in \mathbb{N}^*$.

Table $kereta\_offeringTier$ isn't reflected by a resource of the repository. This table stores the relationships between offering and tier resources.

## 5.5 Kereta Calculation

The calculation module enables the evaluation of functions. The module parses string representations of formulas into a tree representation and enables the calculation of results for different parameter assignments. Figure 5.6 shows the process in more detail. first, the parser detects tokens, transfers the formula into the reverse polish notation and then determine the

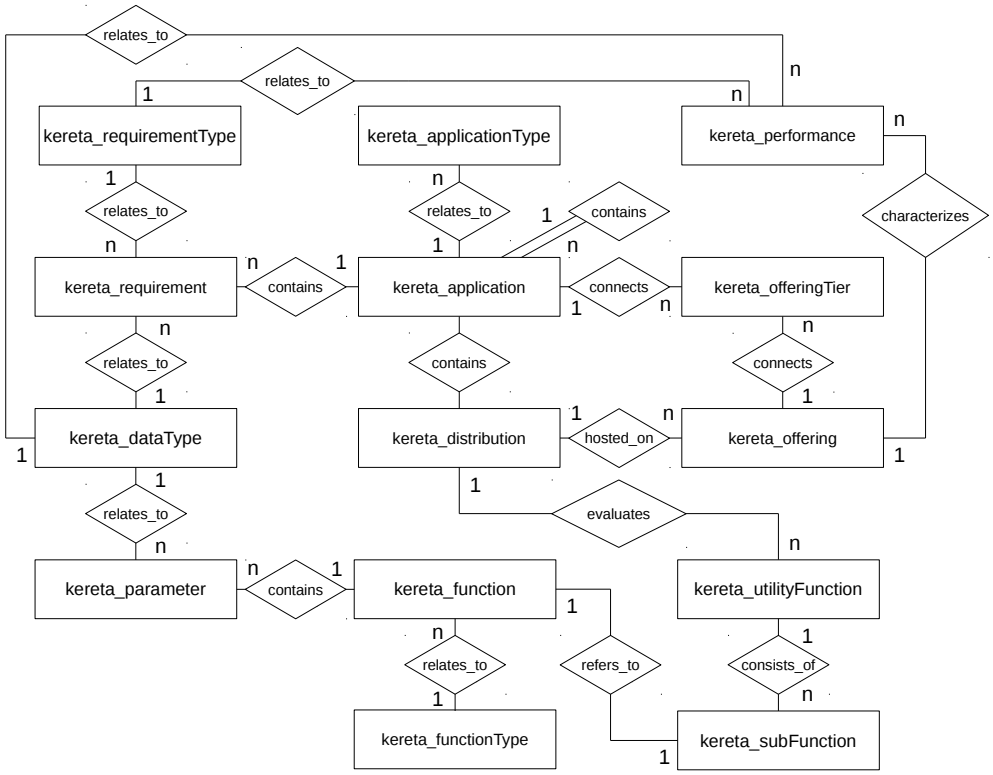


**Figure 5.5:** Kereta Database - Entity-Relationship Diagram

For reasons of clarity, columns have been omitted.

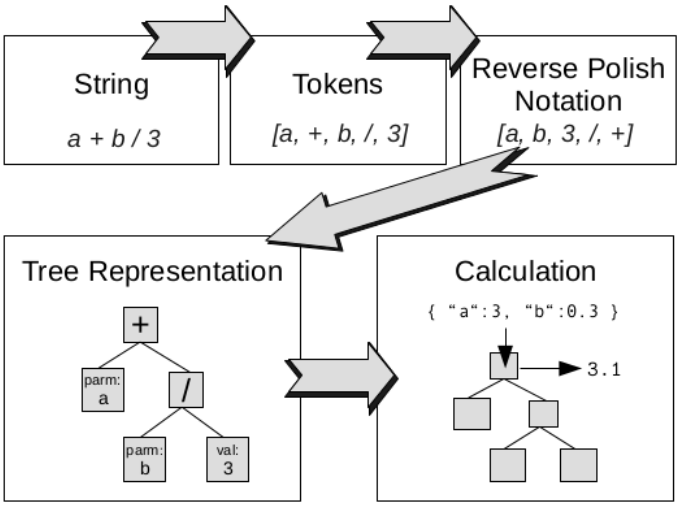


**Figure 5.6:** Calculation Process

tree representation. Section 5.5.2 handles the parsing process, section 5.5.3 the structure of the tree representation and the evaluation process. First, section 5.5.1 defines syntax and semantic for string representations of formulas.

### 5.5.1 Syntax and Semantic

The parser can't handle implicit multiplication and signs. Boolean expression (e.g. as part of if-statements) has to be surrounded with brackets and the combination of characters which identifies operators (e.g. SUM, AND, MIN, etc.) are not allowed inside parameter-names. All operators are identified by 1 or 3 characters. Thus the if-operator is called *IFF* and the or-operator *ORR*. *INT* (integral) and *SUM* operators are followed by a appendix "_{c}". {c} must be a single letter. Operator- and parameter-names are case-sensitive. Commas can be used, but are unnecessary. For instance *ROT(a,b)* is the same as *ROT(a b)*.

#### Values and Parameters

Values can be formed following this regular expression:

```
1 [0-9]+[[.][0-9]+]?
```

**Listing 5.3:** Regular Expression for Value

Signs are not allowed. The parser handles signs (+,-) as binary operation. Negative signs can be avoided by subtracting the number from zero, e.g. $-10 = (0 - 10)$

Parameter names can be formed following this regular expression:

```
1 [a-z,A-Z][a-z,A-T,0-9]*[[_][a-z,A-Z]+]?
```

**Listing 5.4:** Regular Expression for Parameter

Each letters behind the "_" is a single index, e.g. $A1_{x,y,z}$ is expressed by *A1_xyz*.

#### Constants

Expressions can contain constants. If there occurs a parameter without a value assignment, Kereta checks if it is a known constant. Implemented constants are:

- $e = \lim_{n\to\infty}(1+\frac{1}{n})^n \approx 2.71828$ (Euler's number e)
- $pi = \pi \approx 3.14159$ (Pi)

#### Parenthesis

The parser supports two kinds of brackets: [] and (). Human users can use different kinds of brackets to make functions more readable. The parser itself doesn't distinguish different kinds of brackets.

#### Basic Operators

The parser handles the following basic operators: *a* and *b* can be parameters, variables or

| function | use | syntax | description | precedence | associativity |
|---|---|---|---|---|---|
| addition | + | a + b | adding a to b | 1 | left |
| subtraction | - | a - a | subtracting b from a | 1 | left |
| multiplication | * | a * b | multiply a with b | 2 | left |
| division | / | a / b | dividing a through b | 2 | left |
| exponentiation | ^ | a ^b | $a^b$ | 3 | right |

**Table 5.3:** Basic Operators

any statement that can be evaluated.

**Example** Mass–energy equivalence: $E = mc^2$:

```
1 m*c^2
```

**Listing 5.5:** Example: Expression for Einstein's Mass–Energy Equivalence

#### Boolean Operators

The parser handles the following boolean expressions: $a$ and $b$ can be parameters, variables

| function | use | syntax | description |
|---|---|---|---|
| or | ORR | ORR(a,b) | $\begin{cases} 1 & a \neq 0 \; or \; b \neq 0 \\ 0 & else \end{cases}$ |
| xor | XOR | XOR(a,b) | $\begin{cases} 1 & a \neq 0 \; xor \; b \neq 0 \\ 0 & else \end{cases}$ |
| and | AND | AND(a,b) | $\begin{cases} 1 & a \neq 0 \; and \; b \neq 0 \\ 0 & else \end{cases}$ |
| lower than | < | <(a,b) | $\begin{cases} 1 & a < b \\ 0 & else \end{cases}$ |
| bigger than | > | >(a,b) | $\begin{cases} 1 & a > b \\ 0 & else \end{cases}$ |
| equal | EQU | EQU(a,b) | $\begin{cases} 1 & a = b \\ 0 & else \end{cases}$ |
| lower or equal | LEQ | ESM(a,b) | $\begin{cases} 1 & a <= b \\ 0 & else \end{cases}$ |
| bigger or equal | BEQ | EBG(a,b) | $\begin{cases} 1 & a >= b \\ 0 & else \end{cases}$ |
| not | NOT | NOT(a) | $\begin{cases} 1 & a = 0 \\ 0 & else \end{cases}$ |

**Table 5.4:** Boolean Operators

or any statement that can be evaluated.

**Example** Logical consequence: $\neg A \vee B$:

```
1 ORR(NOT(A), B)
```

**Listing 5.6:** Example: Expression for Logical Consequence

#### Functions

The parser handles the following functions: $a$, $b$ and $d$ can be parameters, variables or any statement that can be evaluated. Index $c$ must be a single letter (lower or upper case).

| function | use | syntax | description |
|---|---|---|---|
| square | SQU | SQU(a) | $a^2$ |
| square root | SQR | SQR(a) | $\sqrt{a}$ |
| root | ROT | ROT(a, b) | $\sqrt[b]{a}$ |
| if | IFF | IFF(d, a) | $\begin{cases} a & d \neq 0 \\ 0 & else \end{cases}$ |
| if ... else ... | IFE | IFE(d, a, b) | $\begin{cases} a & d \neq 0 \\ b & else \end{cases}$ |
| minimum | MIN | MIN(a, b) | $\begin{cases} a & a \leq b \\ b & else \end{cases}$ |
| maximum | MAX | MAX(a, b) | $\begin{cases} a & a \geq b \\ b & else \end{cases}$ |
| sine | SIN | SIN(a) | sine of a ($2\pi$) |
| cosine | COS | COS(a) | cosine of a ($2\pi$) |
| tangent | TAN | TAN(a) | tangent of a ($2\pi$) |
| factorial | FAC | FAC(a) | a!, factorial of a |
| sum | SUM | SUM_c(a) | $\sum_{c=c_{min}}^{c_{max}}(a)$ |
| product | PCT | PCT_c(a) | $\prod_{c=c_{min}}^{c_{max}}(a)$ |
| definite integral | IGR | IGR_c(a) | $\int_{c_{min}}^{c_{max}}(a)dc$ |

**Table 5.5:** Function Operators

**Definite Integrals** Integrals can be defined by *IGR_c*, where *c* must be a single letter (lower or upper case). For instance, $\int_{x_{min}}^{x_{max}} 3^x \, dx$ is notated by *IGR_x(3 ^ x)*.The parameter assignment for $x$ must be an array with two values $[x_{min}, x_{max}]$ when evaluating the function.

Index *c* can be used as parameter:

```
1 IGR_x(x)
```

**Listing 5.7:** Example: Expression for Integral over x

**Sum and Product** Sums can be defined by *SUM_c*, where *c* must be a single letter (lower or upper case). E.g. $\sum_{x=x_{min}}^{x_m ax} x + a_x$ is notated by *SUM_x(x+a_x)*. The parameter assignment for $x$ must be an array with two values $[x_{min}, x_{max}]$ when evaluating the function.

Index *c* can be used as parameter and parameter index.
For instance, $\sum_{x=x_{min}}^{x_{max}}(x + a_x)$

```
1 SUM_x(x + a_x)
```

**Listing 5.8:** Example: Expression for a Sum

It is even possible to use parameters with more than one index.
For instance, $\sum_{x=x_{min}}^{x_{max}} \sum_{y=y_{min}}^{y_{max}} (x * y + a_{xy})$

```
1 SUM_x(SUM_y(x * y + a_xy))
```

**Listing 5.9:** Example: Expression for Nested Sums

**Example** *if* $(a > b)$ *{* $a^2$ *} else {* $\frac{a}{b}$ *}*:

```
1 IFE((a>b), a^2, a/b)
```

**Listing 5.10:** Example: Expression for an IF-ELSE Statement

Quadratic formula: $x = \frac{-b+\sqrt{b^2-4ac}}{2a}$:

```
1 (0 - b + SQR(SQU(b) - 4*a*c)) / (2*a)
```

**Listing 5.11:** Example: Quadratic Formula

#### Function Calls

In the repository stored function resources can have a alias (see 6.1.1). Function resources which have a alias can be called from other functions.

| function | use | syntax | alt. syntax | description |
|---|---|---|---|---|
| function call | FCT | FCT(alias, assignment) | | calls a function with a given parameter assignment |

**Table 5.6:** Function Call Operator

The *assignment* is binding a variable or parameter to each parameter of the called function. Individual parameter assignments are separated by "$" and a ":" divides the parameter from the called function from the assigned value or parameter.

**Example** Called function:

- alias = *remote*
- formula = *a* + *b* / 2

The called function has two parameters *a* and *b*. The function can be called in different variants. For instance:

```
1 FCT(remote, a:1.5$b:3)
2 FCT(remote, a:7$b:b)
3 FCT(remote, a:c_1$b:var)
```

**Listing 5.12:** Example: Function Calls

### 5.5.2 Parser

In the first step, the parser detects tokens from the input string (infix expression). Tokens can be of kind *operand*, *operator*, *parenthesis* or *comma*. The list of tokens is the input for *Dijkstra's Shunting-Yard Algorithm* [UK12].

#### Shunting-Yard Algorithm

*Dijkstra's Shunting-Yard Algorithm* is using a stack, the initially empty result list and is processing the input list's tokens from left to right. The algorithm in [UK12] is modified to handle operators from table 5.3, 5.4 and 5.5 in parallel. Table 5.3 contains the precedence and the associativity. Operators from the other tables are handled with precedence 6 and associativity left.

- If the token is of type *operand*, append the token to the result list.
- If the token is of type *comma*, pop tokens from the stack and append them to the result list until a token of type *parenthesis* arises. Reject the *comma* token and leave the *parenthesis* token on the stack.
- If the token (token A) is of type *operator*:
  - If the stack is empty or the stack's top token is of type *parenthesis* or from table 5.4 or 5.5, push token A to the stack.
  - If token A is left associated and the stack's top token's precedence is lower then token A's precedence, push token A to the top of the stack.
  - Else if token A is right associated and the stack's top token's precedence is lower or equal then token A's precedence, push token A to the top of the stack.
  - Else pop tokens from the stack and append them to the result list until the stack is empty or a token of type *parenthesis* is at the top of the stack. Finally, push token A to the top of the stack.
- If the token is of type *parenthesis*:
  - If the token is a opening *parenthesis*, push it to the stack.

– If the token is a closing *parenthesis*, pop tokens from the stack and append them to the result list until a token of type *parenthesis* arises. Reject both, token A and the *parenthesis* token from the stack.

After all tokens from the input list are processed, pop tokens from the stack and append them to the result list until the stack is empty. The result list complies with the reverse polish notation.

#### Tree Representation

The calculation module is using the previous generated list of tokens to generate a tree representation of the function. The tree consists of objects of type *Node*. A *Node* object has private members *operator*, *children*, *value*, *parameter*, *referencedFunction* and *referencedParameters*. Furthermore, it has a public method *calc(Json variables)* which returns a number. Depending on the *operator* member, some of the other members are actually used.

```
1  Class Node {
2    Operator operator;
3    Node[] children;
4    Double value;
5    String parameter;
6    String referencedFunction
7    HashMap<String,String> referencedParameters
8
9    public Double calc(Json variables)
10 }
```

**Listing 5.13:** Node Class

The algorithm is using a stack and processes the list of tokens (in postfix order) from left to right.

- If the token's type is *operand*:
  - If the token represents a number *A*, create a *Node* object with *VALUE* as member *operator* and set *value* to the *A*. Push the object to the stack.
  - If the token represents a parameter *P*, create a *Node* object with *PARM* as member *operator* and set *parameter* to *P*. Push the created object on the stack.
- If the token's type is *operator*
  - If the operators kind is *function call*, create a *Node* object. Set member *operator* to *FUNCTION*. Pop two objects from the stack. These objects should have *operator PARM* by mistake. Actually, it is the identifier of the called function and the regarded parameter assignment. Determine both and set member *referencedFunction* and *referencedParameters* to these values. Push the created object on the stack.

- Else if the operators kind is *sum*, *product* or *definite integral*, create a *Node* object. Set member *operator* to the kind of operation. Set member *children* to an array of size 1. Pop 1 object from the top of the stack and assign it to the one position of *children*. Push the created object on the stack.

- Else create a *Node* object. Set member *operator* to the kind of operation (e.g. ADD) and determine the operations arity $N$. Set member *children* to an array of size $N$. Pop $N$ objects from the top of the stack and assign them to *children*'s positions in reverse order. Push the created object on the stack.

The algorithm terminates with one object on the stack. This object is the tree representation of the function.

#### Example

Function $P_\lambda(k)$ calculates the probability of random variable $k$ in a Poisson distribution with average rate of success $\lambda$.

$$P_\lambda(k) = \frac{\lambda^k}{k!} * e^{-\lambda} \tag{5.1}$$

For the simplified representation $\lambda$ is expressed by $l$ in the infix expression:

```
1 l^k / FAC(k) * e^(0 - l)
```

**Listing 5.14:** Example: Poisson Distribution

In a first step unnecessary spaces are eliminated and tokens are separated.

```
1 [l, ^, k, /, FAC, (, k, ), *, e, ^, (, 0, -, l, )]
```

**Listing 5.15:** Parsing Process - Step 1

The algorithm determines the reverse polish notation.

```
1 [l, k, ^, k, FAC, /, e, 0, l, -, ^, *]
```

**Listing 5.16:** Parsing Process - Reverse Polish Notation

The parser uses the postfix expression to create a object-based tree representation of the function. Figure 5.7 shows the tree.

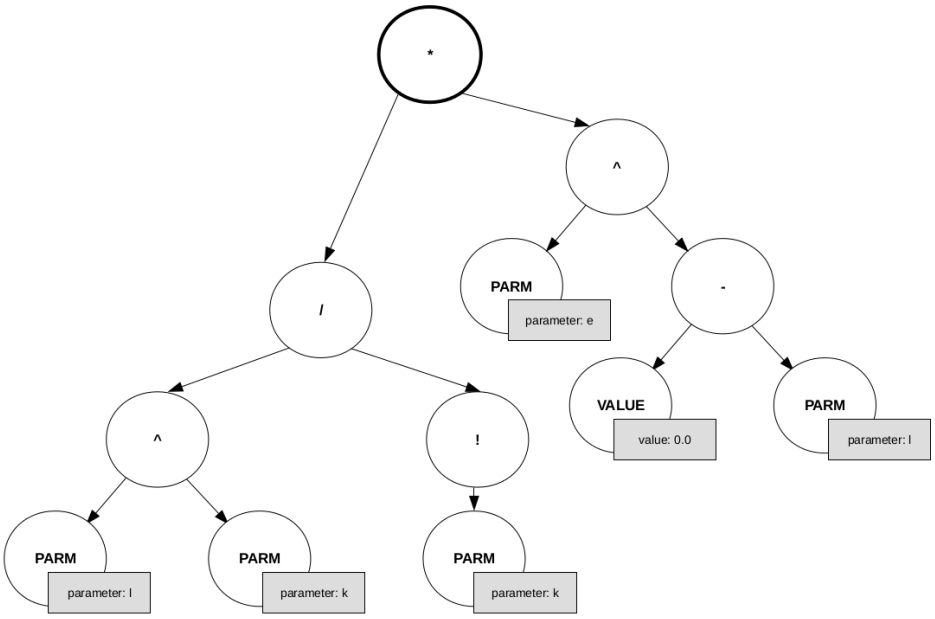


**Figure 5.7:** Tree Representation of $\frac{l^k}{k!}e^{-l}$

### 5.5.3 Calculation

Each *Node* object has an *operator* member. The member determines the size of array *children*. There are three operators defined which haven't child nodes. These operators represents parameters, values and referenced functions.

Method *calc(Json variables)* calculate the result of the tree starting with the node which calc-method is called. The input parameter is a Json-object, for instance $\{"l" : 3, "k" : 2\}$. If the node is a leaf, there are three options:

- The operator is of type *VALUE*:
  The method returns the value of member *value*.
- The operator is of type *PARM*:
  The method returns the value of the variable with name *parameter* from method input *variables*.
- The operator is of type *FUNCTION* (function reference):
  The method selects values from method input *variables* to assemble the assignment given in member *referencedParameters*. Then the function given by member *referenced-Function* is called with the assigned values. The result is returned.

In *Node* objects with any other operator the calc-method calls the calc-method of its children and execute the defined operation with the received values. Three operations have additional functionality. These operators are *sum* $\sum$, *product* $\prod$ and *definite integral* $\int$.

#### Sum and Product

The calc-function of nodes with operator *sum* or *product* need boundaries $x_{min}$ and $x_{max}$ ($\sum_{x=x_{min}}^{x_{x_{max}}}$ or $\prod_{x=x_{min}}^{x_{x_{max}}}$). The nodes' *parameter* attribute contains the name of the value in the input parameter *variables*. The value must be an array with entries $x = [x_{min}, x_{max}]$. The method calls the calc-method of the only child for each value $x_i$ from $x_{min}$ to $x_{max}$. The function input is adapted in each loop: $x = [x_{min}, x_{max}]$ is replaced with $x = x_i$. Each *value* with index $x$ in the function input is also replaced with the $x_i$-th entry.

- *node.operator = SUM*
- *node.parameter = a*
- function input = *{ "a":[0,3], "x_a":[3,5,8,13] }*

In this example the calculation is:

$$\begin{aligned} &node.children[0].calc(\{"a":0, "x\_a":3\})+ \\ &node.children[0].calc(\{"a":1, "x\_a":5\})+ \\ &node.children[0].calc(\{"a":2, "x\_a":8\})+ \\ &node.children[0].calc(\{"a":3, "x\_a":13\}) \end{aligned} \tag{5.2}$$

Products are evaluated analogue.

#### Definite Integral

The interval $[x_{min}, x_{max}]$ of definite integrals $\int_{x_{min}}^{x_{max}}$ is handled analogous to the boundaries of *sum* and *product* nodes.

Definite Integrals are approximated by dividing the interval $[x_{min}, x_{max}]$ in $n$ equally sized intervals and adding the area shown in figure 5.8.

$$\int_{x_{min}}^{x_{max}} f(x)dx \approx h * \sum_{i=1}^{n} f(x_i) \tag{5.3}$$

with

$$h = \frac{x_{max} - x_{min}}{n} \tag{5.4}$$

and

$$x_i = x_{min} - \frac{h}{2} + i * h \tag{5.5}$$

The number of slices $n$ is set to 1000 by default. The node class contains a method to change this number.

The evaluation is implemented as loop from $i = 1$ to $n$. Inside the loop $h$ and $x_i = x_{min} - \frac{h}{2} + i * h$ is calculated. The method calls the calc-method of the only child for $x_i$ . The function input is adapted in each loop: $x = [x_{min}, x_{max}]$ is replaced with $x = x_i$.

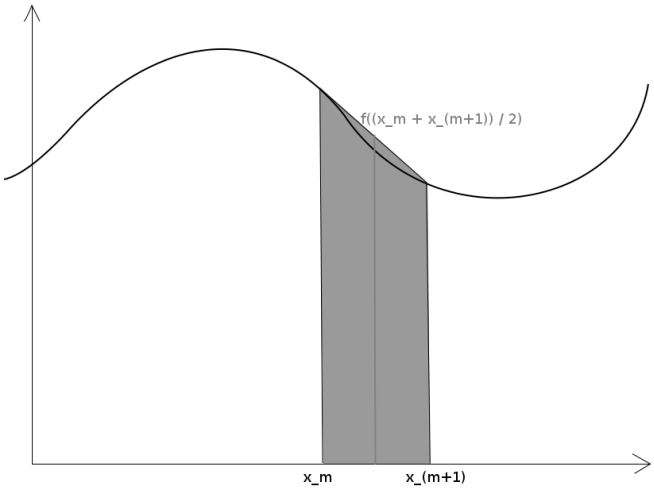


**Figure 5.8:** Integral approximation

#### Example

- *node.operator* = $INTEGRAL$
- *node.parameter* = *a*
- function input = *{ "a":[1,3]}*

In this example the calculation is:

$$\begin{aligned} & h * ( \\ & node.parameter[0].calc(\{"a" : a_1\}) \\ & node.parameter[0].calc(\{"a" : a_2\}) \\ & ... \\ & node.parameter[0].calc(\{"a" : a_n\}) \\ & ) \end{aligned} \tag{5.6}$$

# 6 Implementation

The repository is developed under the name *Kereta*. *Kereta* provides a REST-API, manages resources and offers functionalities which allow for the design of utility functions, and the calculation and comparing of distributed application deployments. *Kereta* is developed for *Java Runtime Environment v1.8* and using the *JAX-RS* reference implementation *Jersey*, furthermore *JAXB* and the *ASM*[1] and *Gson*[2] libraries are utilized. *Kereta* is running in the development process on Oracle's *Java EE 7* reference implementation *Glassfish 4.0*.

Data persistence relies an a MySQL database. *Kereta* using the JDBC driver *Connector/J 5.1.6*. In order to promote safety, only prepared statements are used.

## 6.1 Kereta

Table 6.1 shows the URIs of the repository and responded formats. The following sections covering the resources implementation and the realization of further logic (table 6.1, g.0 - g.2 and h.0 - h.1). In addition section 6.1.1 provides the XML representation of each resource.

### 6.1.1 Resources

The API provides XML-representations for resources contained in the repository. These representations include the resources properties which value isn't null[3].

Each XML-representation contains a *links*-element. Following the HATEOAS constraint, this element contains *link*-elements which has a *rel*-attribute and the corresponding inner text property. A *function*-resource's *links* element is given as an example:

```
1 <links>
2   <link rel="parameters">
3     /Function/0195448d-424e-4aca-b0ea-cb8442f4adf2/Parameter
4   </link>
5   <link rel="this">
6     /Function/0195448d-424e-4aca-b0ea-cb8442f4adf2
7   </link>
```

[1] ObjectWeb ASM: API for decomposing, modifying, and recomposing binary Java classes developed by the OW2 Consortium

[2] Gson: Library to serialize and deserialize Java objects from an to JSON devolped by Google

[3] JAXB standard behaviour

| Nbr. | Resource-URI | Repr. |
|---|---|---|
| a.0 | / | HTML |
| b.0 | /Function | XML |
| b.1 | /Function/{fct-id} | XML |
| b.2 | /Function/{fct-id}/Parameter | XML |
| b.3 | /Function/{fct-id}/Parameter/{parm-name} | XML |
| c.0 | /Application | XML |
| c.1 | /Application/{app-id} | XML |
| c.2 | /Application/{app-id}/Tier | XML |
| c.3 | /Application/{app-id}/Tier/{tier-nbr} | XML |
| c.4 | /Application/{app-id}/Distribution | XML |
| c.5 | /Application/{app-id}/Requirement | XML |
| c.6 | /Application/{app-id}/Requirement/{req-name} | XML |
| c.7 | /Application/{app-id}/Tier/{tier-nbr}/Requirement | XML |
| c.8 | /Application/{app-id}/Tier/{tier-nbr}/Requirement/{req-name} | XML |
| d.0 | /Distribution | XML |
| d.1 | /Distribution/{dstr-id} | XML |
| d.2 | /Distribution/{dstr-id}/UtilityFunction | XML |
| d.3 | /Distribution/{dstr-id}/Offering | XML |
| d.4 | /Distribution/{dstr-id}/Offering/{of-nbr} | XML |
| d.5 | /Distribution/{dstr-id}/Offering/{of-nbr}/Performance | XML |
| d.6 | /Distribution/{dstr-id}/Offering/{of-nbr}/Performance/{prf-name} | XML |
| e.0 | /UtilityFunction | XML |
| e.1 | /UtilityFunction/{uf-id} | XML |
| e.2 | /UtilityFunction/{uf-id}/SubFunction | XML |
| e.3 | /UtilityFunction/{uf-id} /SubFunction/{sf-nbr} | XML |
| e.4 | /UtilityFunction/{uf-id} /SubFunction/{sf-nbr}/NefologParameter | XML |
| f.0 | /Type | XML |
| f.1 | /Type/ApplicationType | XML |
| f.2 | /Type/FunctionType | XML |
| f.3 | /Type/DataType | XML |
| f.4 | /Type/RequirementType | XML |
| g.0 | /Function/{fct-id}/calc?[assignment] | XML |
| g.1 | /UtilityFunction/{uf-id}/calc?key=[key] | XML |
| g.2 | /UtilityFunction/{uf-id}/SubFunction/{sf-nbr} /assign?key=[key]&[assignment] | XML |
| g.3 | /UtilityFunction/{uf-id}/SubFunction/{sf-nbr}/calc?key=[key] | XML |
| g.4 | /Application/{app-id}/select?[query] | XML |
| g.5 | /Application/{app-id}/compare?[query] | XML |
| g.6 | /Distribution/{dstr-id}/check | XML |
| h.0 | /Search?[query] | XML |
| h.1 | /UtilityFunction/{uf-id} /clone?[query] | XML |

**Table 6.1:** Kereta URIs and the related representation.

```
8 </links>
```

**Listing 6.1:** Links - Snippet from Resources' XML Representation

Each resource contains at least a *link*-element with *rel*-attribute "this" which provides the URI of the resource. Relations not covered by the URI are expressed *link*-elements.

#### Function

Existing *Function* resources are accessible via URIs:

- */Function/*
- */Function/{fct-id}*
- */Function/{fct-alias}*

The first option returns a XML-document with document-element *functions*. This element contains the XML-representations of each *Function* resource in the repository. The other options return the XML-representation of the identified resource. Their representation is build like:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <function>
3    <id/>
4    <alias/>
5    <formula/>
6    <description/>
7    <functionType/>
8    <author/>
9    <create/>
10   <links>
11     <link rel="parameters"/>
12     <link rel="this"/>
13   </links>
14 </function>
```

**Listing 6.2:** XML Representation: Function Resource

The link sections of *function* resources provide the URI of nested *Parameter* resources.

#### Parameter

Existing *Parameter* resources are accessible via URIs:

- */Function/{fct-id-or-alias}/Parameter*
- */Function/{fct-id-or-alias}/Parameter/{name}*

The first option returns a XML-document with document-element *parameters*. This element contains the XML-representations of each *Parameter* resource in the repository. The second options return the XML-representation of the identified resource. Their representation is build like:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <parameter>
3    <name/>
4    <dataType/>
5    <defaultValue/>
6    <description/>
7    <functionId/>
8    <author/>
9    <create/>
10   <links>
11     <link rel="this">/>
12   </links>
13 </parameter>
```

**Listing 6.3:** XML Representation: Parameter Resource

#### Application

Existing *Application* resources are accessible via URIs:

- */Application*
- */Application/{app-id}*
- */Application/{app-alias}*

The first option returns a XML-document with document-element *applications*. This element contains the XML-representations of each *Application* resource in the repository. The other options return the XML-representation of the identified resource. Their representation is build like:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<application>
  <id/>
  <alias/>
  <name/>
  <description/>
  <applicationType/>
  <author/>
  <create/>
  <links>
    <link rel="this"/>
    <link rel="tiers"/>
    <link rel="requirements"/>
    <link rel="distributions"/>
  </links>
</application>
```

**Listing 6.4:** XML Representation: Application Resource

The link sections of *Application* resources provide the URIs to overviews about nested resources. These are *Tier* and *Requirement* resources. Another link provides the URI to the set of related *Distribution* resources.

- */Application/{app-id-or-alias}/Distribution*

Since *Distribution* resources are *root resources*, this URI returning a XML-document with document element *distributions* which contains the XML-representations of all *Distribution* resources related to the identified *Application* resource. A specific resource must be identified with URI */Distribution{dstr-id-or-alias}*. For easier operation, the nested URI allows the POST-method (actually *Distribution* resources are created with URI */Distribution*). This method creates a new resource and set the value for attribute *applicationId*.

#### Tier

The purpose of *Tier* resources is to provide a way to assign requirements to a specific tier of the application. Requirements can be assigned to the superior *Application* resource and to *Tier* resources. Existing *Tier* resources are accessible via URIs:

- *.../Application/{app-id-or-alias}/Tier*
- *.../Application/{app-id-or-alias}/Tier/{tierNbr}*

Identifier *tierNbr* can be any natural number ($tierNbr \in \{1, 2, 3, ...\}$). The first option returns a XML-document with document-element *tiers*. This element contains the XML-representations

of each *Tier* resource within the *Application* resource. The other options return the XML-representation of the identified resource. Their representation is build like:

```
<tier>
  <applicationId/>
  <tierNbr/>
  <name/>
  <description/>
  <author/>
  <create/>
  <links>
    <link rel="this"/>
    <link rel="requirements"/>
  </links>
</tier>
```

**Listing 6.5:** XML Representation: Tier Resource

The requirement link contains a URI to the tier's requirements. These *Requirement* resources differ from the superior *Application* resource's *Requirement* resources.

#### Requirement

*Requirement* resources can be related to an *Application* or an *Tier* resource. Existing *tier* resources are accessible via URIs:

- *Application/{app-id-or-alias}/Requirement*
- *Application/{app-id-or-alias}/Requirement/{name}*
- *Application/{app-id-or-alias}//Tier/{tierNbr}/Requirement*
- *Application/{app-id-or-alias}//Tier/{tierNbr}/Requirement/{name}*

Identifier *name* can be any string, but must be unique within its siblings. The first and third option returns a XML-document with document-element *requirements*. This element contains the XML-representations of each *Requirement* resource within the identified *Application* or *Tier* resource. The other options return the XML-representation of the identified *Requirement* resource. The representation is build like:

```
<requirement>
  <applicationId/>
  <applicationTier/>
  <name/>
  <value/>
  <demand/>
```

```
7    <dataType/>
8    <requirementType/>
9    <author/>
10   <create/>
11   <links>
12     <link rel="this"/>
13   </links>
14 </requirement>
```

**Listing 6.6:** XML Representation: Requirement Resource

Two cases can be distinguished: XML-element *applicationTier*'s inner text property is empty (related to a *Application* resource) or contains a natural number (related to a *tier* resource).

#### Distribution

Existing *Distribution* resources are accessible via URIs:

- */Distribution*
- */Distribution/{dstr-id}*
- */Distribution/{dstr-alias}*

The first option returns a XML-document with document-element *distributions*. This element contains the XML-representations of each *Distribution* resource in the repository. The other options return the XML-representation of the identified resource. Their representation is build like:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <distribution>
3    <id/>
4    <alias/>
5    <applicationId/>
6    <representation/>
7    <language/>
8    <langVersion/>
9    <author/>
10   <create/>
11   <links>
12     <link rel="utilityFunctions"/>
13     <link rel="application"/>
14     <link rel="offerings"/>
15     <link rel="this"/>
16   </links>
17 </distribution>
```

**Listing 6.7:** XML Representation: Distribution Resource

The link section of *Distribution* resources provides the URI of the set of nested *Offering* resources. Additional links provide the URIs of the related *Application* and *UtilityFunction* resources.

#### Offering

*Offering* resources represent Cloud offerings with a specific configuration. Since Nefolog2.9.1 provides a repository of Cloud providers, Cloud offerings and their configurations, *Offering* resources identifies the related resources in the Nefolog repository.

Existing *Offering* resources are accessible via URIs:

- *.../Distribution/{dstr-id-or-alias}/Offering/*
- *.../Distribution/{dstr-id-or-alias}/Offering/{offeringNbr}*

The first option returns a XML-document with document-element *offerings*. This element contains the XML-representations of each *Offering* resource in the repository. The other options return the XML-representation of the identified resource. The representation is build like:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <offering>
3    <number/>
4    <distributionId/>
5    <nefologConfiguration/>
6    <nefologConfigurationId/>
7    <nefologOfferingName/>
8    <nefologServiceType/>
9    <nefologProvider/>
10   <author/>
11   <create/>
12   <links>
13     <link rel="this"/>
14     <link rel="performances"/>
15   </links>
16 </offering>
```

**Listing 6.8:** XML Representation: Offering Resource

The performance link contains a URI to the offering's *Performance* resources.

#### Performance

*Performance* resources can be used to evaluate the fulfilment of requirements. These resource are similar to *Requirement* resources, except they nested within a *Offering* resource and express what is fulfilled (in contrast to what has to be fulfilled).

Existing *Performance* resources are accessible via URIs:

- *.../Distribution/{dstr-id-or-alias}/Offering/{offeringNbr}/Performance*
- *.../Distribution/{dstr-id-or-alias}/Offering/{offeringNbr}/Performance{name}*

The first option returns a XML-document with document-element *performances*. This element contains the XML-representations of each *Performance* resource in the repository. The other options return the XML-representation of the identified resource. The representation is build like:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<performance>
  <distributionId/>
  <offeringNumber/>
  <name/>
  <value/>
  <fulfilment/>
  <dataType/>
  <requirementType/>
  <author/>
  <create/>
  <links>
    <link rel="this"/>
  </links>
</performance>
```

**Listing 6.9:** XML Representation: Performance Resource

#### UtilityFunction

*Utility function* resources nesting a set of *SubFunction*(utility function) resources. In a nutshell, *SubFunction* resources are links to *Function* resources of function type *revenue* or *cost* (see 5.2.11).

Existing *utility function* resources are accessible via URIs:

- *UtilityFunction*
- *UtilityFunction/{uf-id}*

- *UtilityFunction/{uf-alias}*

The first option returns a XML-document with document-element *utilityFunctions*. This element contains the XML-representations of each *utilityFunction* resource in the repository. The other options return the XML-representation of the identified resource. Their representation is build like:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<utilityFunction>
  <alias/>
  <author/>
  <create/>
  <description/>
  <id/>
  <distributionId/>
  <links>
    <link rel="subFunctions/>
    <link rel="distribution/>
    <link rel="this">
  </links>
</utilityFunction>
```

**Listing 6.10:** XML Representation: Utility Function Resource

The link section of *utility function* resources provides the URI of the set of nested *SubFunction* resources and the related *Distribution* resource.

#### SubFunction

*SubFunction* resources are linking a reusable *Function* resource to a *UtilityFunction* resource. The linked *Function* resource can have parameters, thus values must be assigned before calculating utility. The API provides the parameter assignment to *SubFunction* resources. When calculating utility, the system uses the assignment to calculate the result of the linked *Function* resource.

Not only *Function* resources can be linked. *SubFunction* resources can be linked to the Nefolog cost-calculation functionality. This is done by setting the *functionId* attribute to:

```
nefolog$<nefolog-configuration-id>
```

**Listing 6.11:** Integration: Nefolog Cost Calculation

<*nefolog-configruation-id*> must be replaced with the id of the Nefolog offering configuration's id. Thereby, Nefolog cost-calculation can be used like a *function* resource.

Existing *sub-function* resources are accessible via URIs:

- *UtilityFunction/{uf-id-or-alias}/SubFunction/{sf-nbr}*
- *UtilityFunction/{uf-id-or-alias}/SubFunction/{sf-nbr}*

The first option returns a XML-document with document-element *subFunctions*. This element contains the XML-representations of each *SubFunction* resource in the repository. The second options return the XML-representation of the identified resource. The representation is build like:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<subFunction>
  <utilityFunctionId/>
  <number/>
  <functionId/>
  <author/>
  <create/>
  <links>
    <link rel="this/>
    <link rel="function/>
    <link rel="parameters"/>
  </links>
</subFunction>
```

**Listing 6.12:** XML Representation: Sub-Function Resource

The function link the URI of the linked *Function* resource (this XML element is missing if the resource is referring to Nefolog). The parameter link contains the URI of the set of parameters in the connected *Function* resource.

#### NefologParameter

- *UtilityFunction/{uf-id-or-alias}/SubFunction/{sf-nbr}/NefologParameter*

In cases where the *SubFunction* resource is referring to Nefolog, this URI returns a XML document with document-element *parameters* which contains a element *parameter* for each (possible) parameter of the Nefolog cost-calculation for the Nefolog-configuration-id specified in the *sub-function* resource. For instance (Nefolog-configuration-id: 290):

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<parameters>
  <parameter>Hour</parameter>
  <parameter>Month</parameter>
  <parameter>GBExternalNetworkEgress</parameter>
```

```
6   <parameter>location_zone</parameter>
7   <parameter>usage_pattern</parameter>
8 </parameters>
```

**Listing 6.13:** XML Representation: Nefolog Parameter

#### Type

The type URI returns links to each kind of types.

- */Type*

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <types>
3   <links>
4     <link rel="this"/>
5     <link rel="dataTypes"/>
6     <link rel="functionTypes"/>
7     <link rel="applicationTypes"/>
8     <link rel="requirementTypes"/>
9   </links>
10 </types>
```

**Listing 6.14:** XML Representation: Type Resources

#### DataType

Users are able to create customized *DataType* resources. This is reached by calling the POST-method for */Type/DataType/<dataType-name>*. *<dataType-name>* has to be replaced with the name of the *DataType* resource. The DELETE-method deletes a customized type.

- */Type/DataType*

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <types>
3   <type>number</type>
4   <type>string</type>
5   <type>array of strings</type>
6   <type>array of numbers</type>
7   <type>array of arrays</type>
8 </types>
```

**Listing 6.15:** XML Representation: Data Type Resource

#### FunctionType

Users are able to create customized *FunctionType* resources. This is reached by calling the POST-method for */Type/FunctionType<functionType-name>*. *<functionType-name>* has to be replaced with the name of the *FunctionType* resource. The DELETE-method deletes a customized type.

- */Type/FunctionType*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<types>
  <type>revenue</type>
  <type>cost</type>
  <type>misc</type>
</types>
```

**Listing 6.16:** XML Representation: Function Type Resource

#### ApplicationType

Users are able to create customized *ApplicationType* resources. This is reached by calling the POST-method for */Type/ApplicationType<applicationType-name>*. *<applicationType-name>* has to be replaced with the name of the *ApplicationType* resource. The DELETE-method deletes a customized type.

- */Type/ApplicationType*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<types>
</types>
```

**Listing 6.17:** XML Representation: Application Type Resource

#### RequirementType

Users are not allowd to create customized *RequirementType* resources. The initial set of resources (*functional*, *non-functional*) is not editable.

- */Type/RequirementType*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<types>
  <type>functional</type>
  <type>non-functional</type>
</types>
```

**Listing 6.18:** XML Representation: Requirement Type Resource

### 6.1.2 Functionality

The Kereta repository manages resources. Furthermore, there are functionalities to evaluate functions, sub-functions and utility functions, rank or select distributions based on their utility, search for existing resources and reuse solutions.

#### Function Calculation

API-users can get knowledge about parameter resources nested within a function resource. For instance, if a *Function* resource's alias is *myFct*, the GET-method for URI */Function/myFct/Parameter* returns a XML document containing a XML representation of each *Parameter* resource. Thereby, users get also knowledge about data types and default values (see section 6.1.1).

The REST API enables the evaluation of *Function* resources by URI (see table 6.1, g.0):

- */Function/{fct-id}/calc?[assignment]*

The query string includes a key-value pair for each parameter a value has to be assigned to. Different data types are allowed (see section 6.1.1)). Types *number* is straightforward: *<parameter-name>:<parameter-value>*. URI parameters are separated by &. Arrays are expressed according to JSON (square bracket notation and separate elements with commas). The type *array of arrays* is generic. The arrays inside the array can be of type *array of numbers* or even *array of arrays*.

**Example** The considered *Function* resources has alias *myFct*. Formula is

$$\sum_{x=0}^{n} \sum_{y=0}^{m} \frac{a_{x,y}}{b} \tag{6.1}$$

```
SUM_x(SUM_y(a_xy/b))
```

Parameters are $x$, $y$, $a_{x,y}$ and $b$. $x, y$ are arrays containing two numbers (see section 5.5.1), $b$ is a number and $a_{x,y}$ is obviously a array of arrays of numbers. For instance, $x = [0,1], y = [0,2], a_{x,y} = [[-1,2.2],[1,2],[-1.6,4]], b = 0.777$.

- */Function/myFct/calc?x=[0,2]&y=[0,1]&a_xy=[[-1,2.2],[1,2],[-1.6,4]]&b=0.777*

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<calculation>
  <result>8.49</result>
  <formula>SUM_x(SUM_y(a_xy/b))</formula>
  <toks>[SUM_x, (, SUM_y, (, a_xy, /, b, ), )]</toks>
  <rpn>[a_xy, b, /, SUM_y, SUM_x]</rpn>
  <parameters>
    { "a_xy": [[-1,2.2],[1,2],[-1.6,4]], "b": 0.777, "x": [0,2],
       "y": [0,1] }
  </parameters>
</calculation>
```

**Listing 6.19:** XML Representation: Function Calculation

The GET-method returns a XML document containing the result, the formulas tokens, reverse polish notation and parameter assignments. tokens and reverse polish notation can be used for debugging. Kereta doesn't round, however, returned values are rounded to two decimal places.

#### SubFunction Calculation

In contrast to the calculation of *Function* resources, the calculation of *SubFunction* resources is based on two steps. The first step is to assign values and a key, the second step is the evaluation (see table 6.1, g.2 and g.3).

The assignment works analogous to the calculation of *Function* resources, except the additional key parameter in the query string. The value for key can be any string.

- */UtilityFunction/myUF/SubFunction/1/assign?**key=myKey**&x=1&y=2*

The evaluation requires only the previously defined key:

- */UtilityFunction/myUF/SubFunction/1/calc?**key=myKey***

The GET-method for the assignment returns a XML document with document element *key* and the defined key as text content. The GET-method for the evaluation returns a XML-document similar to the calculation of *Function* resources. The *calculation* element contains attributes for class, the identifier of the *SubFunction* resource within the superior *UtilityFunction* resource and the function type of the referred *Function* resource (*revenue* or *cost*). The first part inside the document element looks exactly the same, but parameters are listed separated.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<calculation class="subfunction" number="" type="">
  <result/>
  <formula/>
  <toks/>
  <rpn/>
  <parameters>
    <parameter name="" />
    ...
  </parameters>
</calculation>
```

**Listing 6.20:** XML Representation: Sub-Function Calculation

#### UtilityFunction Calculation

The evaluation of a *UtilityFunction* resources is easy to operate, but previously values must be assigned for each *SubFunction* resource within using a single key. The Evaluation only requires the GET-method for the defined URI (table 6.1, g.1) with the specified key. The evaluation requires only the previously defined key:

- */UtilityFunction/myUF/calc?**key=myKey***

The method returns a XML document containing a *result* element as child of the document element. This element contains the calculated utility. The identifier of the related *Distribution* resources is also contained, just like the sub-calculations (the representations of *SubFunction* calculations). The class attribute distinguish the *calculation* element of *UtilityFunction* calculation from *SubFunction* calculation.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<calculation class="utilityFunction">
  <result/>
  <distributionId/>
  <subCalculations>
    <calculation class="subfunction" number="" type="" />
    ...
  </subCalculations>
</calculation>
```

**Listing 6.21:** XML Representation: Utility Function Calculation

#### Decision Support

Three URIs provides decision support (table 6.1, g.4, g.5 and g.6). *myApp* is an *Application* resource's alias and *myDstr* is an *Distribution* resource's alias.

- */Application/myApp/select?query*
- */Application/myApp/compare?query*
- */Distribution/myDstr/check*

The first two URIs are similar, except the first returns only the distribution promising the highest utility and the second returns a list of all available distributions and the expected utility. The query string looks the same. There must be a parameter for each considered *Distribution* resource separated by &. The parameter is build like:
*<dstr-id-or-alias>=<uf-id-or-alias>:<key>*
Parameter assignments must be performed before using the GET-method for the two URIs. This requires the assignment URI from *SubFunction* calculation.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <listing>
3    <calculation class="utilityFunction">
4      <result/>
5      <distributionId/>
6      <subCalculations>
7        <calculation class="subfunction" number="" type="">
8        ...
9      </subCalculations>
10   <calculation>
11   ...
12 <listing>
```

**Listing 6.22:** XML Representation: Select Distribution

The GET-method for the compare URI returns a XML document with document element *listing*. This element contains a *calculation* element for each suggested *Distribution* resource. These elements are the same as the document element of the *UtilityFunction* calculation.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <selection>
3    <calculation class="utilityFunction">
4    ...
5    <calculation>
6  <selection>
```

**Listing 6.23:** XML Representation: Compare Distribution

The GET-method for the select URI returns a XML document with document element *selection*. In contrast to the compare URI, this element contains only the *calculation* element for the *Distribution* resource which promises the highest utility.

The check URI returns a comparison of requirements and performances for a *Distribution* resource. The GET-method returns a XML-document with document element *comparison* and a element *check* for each *Requirement* resource connected to the related *Application* resource and its *Tier* resources. Requirements related to the *Application* resource are nested within the *global*, Requirements related to a specific *Tier* resource are organized within the *tier* element.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<comparison distributionId="">
  <global>
    <check>
      <name/>
      <dataType/>
      <requirementType/>
      <requirement>
        <value/>
        <demand/>
      </requirement>
      <performance>
        <value/>
        <fulfilment/>
      </performance>
    </check>
    ...
  </global>
  <tiers>
    <tier tierNbr="">
      <check/>
      ...
    </tier>
    ...
  </tiers>
<comparison>
```

**Listing 6.24:** XML Representation: Check Requirements

#### Reusability

Two URIs offering functionality to increase Kereta's reusability. *myApp* is an *Application* resource's alias and *myUF* an *UtilityFunction* resource's alias, of course, the resources IDs can

be used instead.

- */Search/myApp/select?query*
- */UtilityFunction/myUF/clone?query*

Kereta allows for the search for *Application*, *UtilityFunction* and *Function* resources. String query's parameters are *resource*, *applicationType* and *functionType*. The later two are optional and the last one only makes sense when searching for *Function* resources. Values for parameter *resource* can be *Application*, *UtilityFunction* or *Function*, values for parameter *applicationType* can be any user defined application type and values for parameter *functionType* can be *revenue*, *cost*, *misc* or any user defined function type. The method returns a XML document with document element *selection*. This element contains the XML representations of each selected resource.

*UtilityFunction* resources are complicated in structure. The are related to an *Distribution* resource, several *SubFunction* resources are nested and related to *Function* resource. The second URI's GET-method offloads to work to create all these resources and establish relationships. If an *UtilityFunction* resource already exists, the method creates a copy of the resource and nested *SubFunction* resources and automatically change their relationships. The string query must contain parameter *distribution*. The parameter's value is the targeted *Distribution* resource's ID or alias. The method returns the new *UtilityFunction* resource's XML representation.

### 6.1.3 Kereta Database

This section presents the database tables. The following table shows the database tables' columns, their name, data type and a description. Private keys are indicated with *PK* and foreign keys with *FK*.

#### kereta_application

| key | field | data type | description |
|---|---|---|---|
| PK | id | VARCHAR(36) | UUID |
| PK | tier | INT | 0 or the identifying number of the application specific component |
| | alias | VARCHAR(8) | at most 8 character long alias |
| | name | VARCHAR(128) | application name |
| | description | TEXT | description of the application |
| | application_type | VARCHAR(128) | application type, see 5.2.11 |
| | author | VARCHAR(128) | author of the resource |
| | create | TIMESTAMP | timestamp of resource creation |

**Table 6.2:** kereta_application columns

Table *kereta_application* persists *application-specific topologies* and *application-specific components* from the resource model. The *tier* property is 0 for *application-specific topologies* and $\in \mathbb{N}^*$ for *application-specific components*.

#### kereta_distribution

| key | field | data type | description |
|---|---|---|---|
| PK | id | VARCHAR(36) | UUID |
| | alias | VARCHAR(8) | at most 8 character long alias |
| | application_id | VARCHAR(36) | the associated application's id |
| | topology | TEXT | the viable topology in the specified language |
| | topology_language | VARCHAR(128) | the topology language in use |
| | topology_language_version | VARCHAR(128) | Version of the specified topology language |
| | author | VARCHAR(128) | author of the resource |
| | create | TIMESTAMP | timestamp of resource creation |

**Table 6.3:** kereta_distribution columns

Table *kereta_distribution* persists *application distribution* resources. The *tier* property is 0 for *application-specific topologies* and $\in \mathbb{N}^*$ for *application-specific components*. Attribute *topology* allows for the storage of the topology model in a topology language. The topology language in use can be specified with attributes *topology_language* and *topology_language_version*.

#### kereta_offering

| key | field | data type | description |
|---|---|---|---|
| FK | distribution_id | VARCHAR(36) | the superior distribution's id |
| PK | number | INT | the offering's identifier within the distribution resource |
| | nefolog_configuration | VARCHAR(128) | nefolog-name of the corresponding configuration |
| | nefolog_configuration_id | INT | nefolog-id of the corresponding configuration |
| | nefolog_offering_name | VARCHAR(128) | nefolog-name of the offering |
| | nefolog_service_type | VARCHAR(128) | nefolog-service type of the offering |
| | nefolog_provider | VARCHAR(128) | nefolog-provider of the offering |
| | author | VARCHAR(128) | author of the resource |
| | create | TIMESTAMP | timestamp of resource creation |

**Table 6.4:** kereta_distribution columns

The resource model contains *application subgraphs*. The repository reduces the resource to the included Cloud offering. The table persists informations about the Cloud offering from the Nefolog framework.

#### kereta_offeringTier

| key | field | data type | description |
|---|---|---|---|
| FK | application_id | VARCHAR(36) | application_id and application_tier identifies the connected tier resource |
| FK | application_tier | INT | |
| FK | offering_number | INT | distribution_id and offering_number identifies the connected offering resource |
| FK | distribution_id | VARCHAR(36) | |
| | create | Timestamp | timestamp of connection creation |

**Table 6.5:** kereta_offeringTier columns

Table *kereta_offeringTier* persists relationships between *application-specific component* and *application subgraph* resources. The table stores the two by two attributes to identify the connected resources.

#### kereta_requirement

| field | data type | description |
|---|---|---|
| application_id | VARCHAR(36) | the superior application's id |
| application_tier | INT | 0 if associated to the application, else the tier's identifier within the application resource |
| name | VARCHAR(128) | resource identifier and the requirement's name |
| value | VARCHAR(128) | the requirements value |
| demand | VARCHAR(1) | value is '<', '>' or '=' |
| data type | VARCHAR(128) | data type, see 5.2.11 |
| requirement_type | VARCHAR(36) | value is 'functional' or 'non-functional', see 5.2.11 |
| author | VARCHAR(128) | author of the resource |
| create | TIMESTAMP | timestamp of resource creation |

**Table 6.6:** kereta_requirement columns

Table *kereta_requirement* persists *requirement* resources for *application-specific topologies* and *application-specific components* from the resource model. The *application_tier* property is 0 for *application-specific topologies* and $\in \mathbb{N}^*$ for *application-specific components*.

| key | field | data type | description |
|---|---|---|---|
| FK | distribution_id | VARCHAR(36) | the superior distribution's id |
| FK | offering_number | INT | identifier of the offering within the superior distribution resource |
| PK | name | VARCHAR(128) | resource identifier and the performance's name |
| | value | VARCHAR(128) | the performance value |
| | fulfilment | VARCHAR(1) | value is '<', '>' or '=' |
| | data_type | VARCHAR(128) | data type, see 5.2.11 |
| | requirement_type | VARCHAR(36) | value is 'functional' or 'non-functional', see 5.2.11 |
| | author | VARCHAR(128) | author of the resource |
| | create | TIMESTAMP | timestamp of resource creation |

**Table 6.7:** kereta_performance columns

#### kereta_performance

Table *kereta_performance* persists *performance* resources for *subgraphs* from the resource model.

#### kereta_function

| key | field | data type | description |
|---|---|---|---|
| PK | id | VARCHAR(36) | UUID |
| | alias | VARCHAR(8) | at most 8 character long alias |
| | formula | TEXT | textual representation of the function |
| | description | TEXT | description of the function |
| | function_type | VARCHAR(128) | function type, see 5.2.11 |
| | author | VARCHAR(128) | author of the resource |
| | create | TIMESTAMP | timestamp of resource creation |

**Table 6.8:** kereta_function columns

*Function* resources are persisted in table *kereta_function*. Attribute *formula* contains the text-representation of the function.

#### kereta_parameter

*Parameter* resources are persisted in table *kereta_parameter*.

#### kereta_utilityFunction

Table *kereta_utilityFunction* persists *utility functions* from the resource model.

| key | field | data type | description |
|---|---|---|---|
| FK | function_id | VARCHAR(36) | the superior function resource's identifier |
| PK | name | VARCHAR(128) | the parameter's name and identifier within the function resource |
| | default_value | VARCHAR(128) | default value of the parameter |
| | description | TEXT | description of the parameter |
| | data_type | VARCHAR(128) | data type, see 5.2.11 |
| | author | VARCHAR(128) | author of the resource |
| | create | TIMESTAMP | timestamp of resource creation |

**Table 6.9:** kereta_parameter columns

| key | field | data type | description |
|---|---|---|---|
| PK | id | VARCHAR(36) | UUID |
| | alias | VARCHAR(8) | at most 8 character long alias |
| | distribution_id | VARCHAR(36) | the associated distribution's id |
| | description | TEXT | description of the utility function |
| | author | VARCHAR(128) | author of the resource |
| | create | TIMESTAMP | timestamp of resource creation |

**Table 6.10:** kereta_utilityFunction columns

#### kereta_subFunction

| key | field | data type | description |
|---|---|---|---|
| FK | utility_function_id | VARCHAR(36) | the superior utility function's id |
| PK | number | INT | identifier of the sub-function within the superior function resource |
| | function_id | VARCHAR(36) | the identifier of the called function ressource or the nefolog-configuration id |
| | author | VARCHAR(128) | author of the resource |
| | create | TIMESTAMP | timestamp of resource creation |

**Table 6.11:** kereta_subFunction columns

Table *kereta_subFunction* persists *(utility function) sub-functions* from the resource model. Resources are identified by the *utilty_function_id* and the *number* attribute.

#### Types

The four type-tables are build in a uniform manner. *kereta_requirementType* and *kereta_functionType* have a initial set of rows. Users can't add rows to *kereta_requirementType*. In all other cases users can define customized types.

- **kereta_requirementType**

- **kereta_applicationType**
- **kereta_functionType**
- **kereta_dataType**

| key | field | data type | description |
|---|---|---|---|
| PK | name | VARCHAR(128) | name of the type and identifier |
| | author | VARCHAR(128) | author of the resource |
| | create | TIMESTAMP | timestamp of resource creation |

**Table 6.12:** kereta type-tables columns

# 7 Evaluation

Section 7.2 validates the implementation of Kereta by running the workflow described in Section 7.1 on a concrete example. The example revolves around the question if a partially redeployment of the MediaWiki application in the Cloud increases the utility of the alternative deployment.

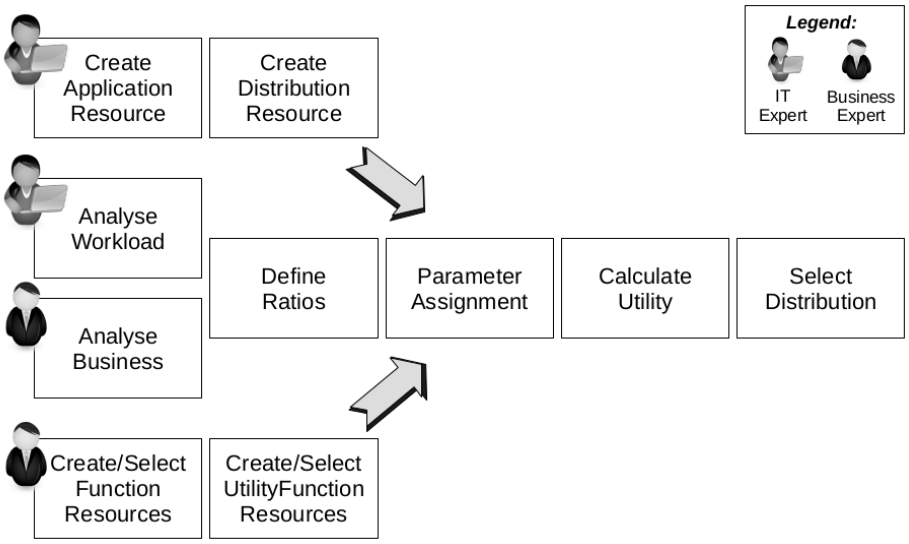


**Figure 7.1:** Decision Workflow

Calculating utility and selecting the viable topology which offers the highest utility

## 7.1 Workflow

The workflow shows in 7.1 combine the work of business and IT experts. IT experts have to create resources regarding the technical aspects of the suggested application and its possible distributions. This includes *Application*, *Tier*, *Requirement*, *Distribution*, *Offering* and *Performance* resources. The analysis of the defined workload also requires IT experts.

Meanwhile, business experts have to analyse the application's financial aspects, especially by creating the corresponding business model. Business experts are also responsible to deliver utility functions. Kereta offer a variety of possibilities to find a utility function: Users can start from scratch, create their own *Function* resources (e.g. customized revenue or cost functions) and assemble a customized utility function. Users can also search the repository

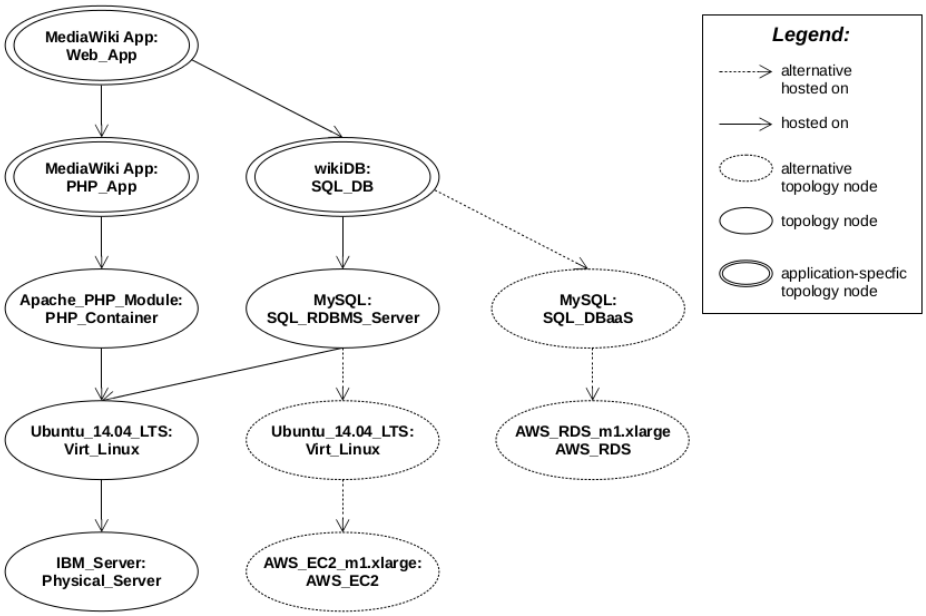


**Figure 7.2:** MediaWiki Application, Alternative Distributions

Derived from [SALS14].

for already created *Function* resources and assemble them in a new way (e.g. combine a revenue and a cost function from different utility functions). A mixture between the possibilities is also possible. Finally, users can reuse already existing *UtilityFunction* resources.

## 7.2 Practical Use

The MediaWiki application is hosted on a physical server. Both, the PHP-based frontend and the database are hosted on this machine (topology $T_{\mu}^{0}$). The question that arises is which advantages could other distributions offer. The PHP application should stay on the machine, the database can be either provided by a database server running on a IaaS Cloud offering or by a DbaaS Cloud offering. Two Cloud services are selected. First, a AWS EC2 m1.xlarge instance running a database server on a Linux operating system (topology $T_{\mu}^{1}$) and second, a Amazon RDS db.m1.xlarge for MySQL offering (topology $T_{\mu}^{2}$). Figure 7.2 shows the original and the alternative distributions. At the core, it's about weighting the higher costs of the alternative deployments against the potentially gained advantages. The utility function concept from section 4.1.1 takes on the task of trade of these different aspects.

Only two non-functional requirements are suggested:

- Location: European Union
- Throughput: At least 15 Req./s.

$M0$ represents the first month of the first year, $M12$ the first month the second year and so on. The example suggests the time period from February 1, 2015 to April 31, 2016, thus, time period $T$ is defined by $T = [M1, M15]$.

### 7.2.1 Workload

IT experts identified three different workloads. This evaluation is orientated on [SALS14]. The work in [SALS14] is based on 1 GB representative database content and a set of tool-generated database queries. These 23 queries are categorized in three categories: (1) *compute low* (CL), (2) *compute medium* (CM) and (3) *compute high* (CH). The paper includes measurement results for the distribution and alternative distributions suggested in this chapter. These numbers are used to get the workload for the evaluation. Workload is observed and predicted for the time period from the 1. January to the 31. December of a representative year.

Workload $w_0$'s (normal utilization) usage profile includes a composition of the 23 database queries. CL queries are selected with probability 0.714, CM and CH queries with probability 0.143. Thus, the probability for CL queries is about five time higher then the probability for CM or CH queries. The calculated average throughput is:

- On-Premise: 18.2 Req./s.
- IaaS: 13.3 Req./s.
- DBaaS: 19.4 Req./s.

For workload $w_1$ (high utilization) CH queries are selected with probability 0.714, CL and CM queries with probability 0.143. Therefore, the calculated average throughput is:

- On-Premise: 4.2 Req./s.
- IaaS: 3.0 Req./s.
- DBaaS: 6.9 Req./s.

The *average number of transactions per user* is defined by:

$$\overline{TPU(w_j)} = \begin{cases} 5 & w_j = w_0 \\ 12 & w_j = w_1 \end{cases} \tag{7.1}$$

The Wikimedia Foundation, Inc. provides statistics about the Internet encyclopedia Wikipedia[1] There are records about *edits per month*[2]. The analysis of *edits per month* for German articles in the years 2011, 2012 and 2014 shows a repetitive pattern. Figure 7.3 presents that there is a unusually large deviation in January. February and March shows a smaller positive deviation. The other months lie between the average (approximately) and $-5\%$.

[1] https://stats.wikimedia.org/EN/Sitemap.htm
[2] https://stats.wikimedia.org/EN/TablesDatabaseEdits.htm

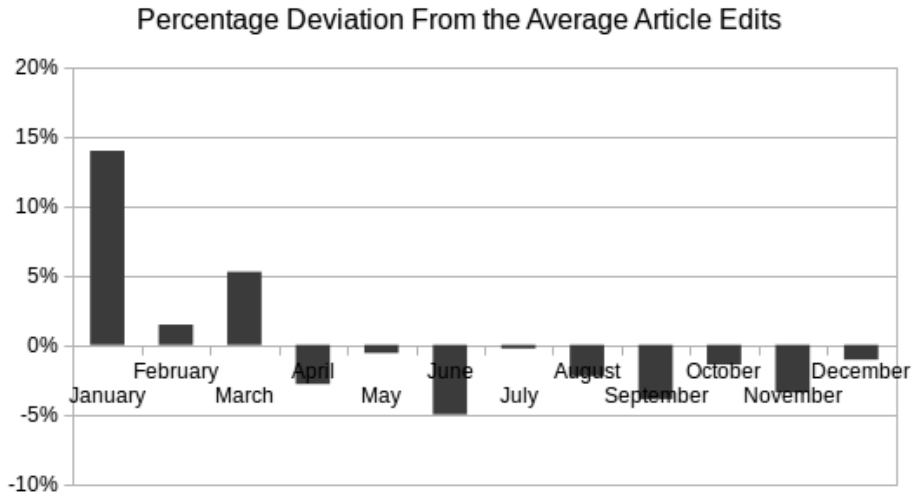


**Figure 7.3:** Wikipedia, Deviation from the Average Article Edits

Based on the records for German articles in 2011, 2012 and 2014.

Following this observation, we assume, that there is a high *average number of users* in January. The ratio is smaller in February and March and additional smaller in the rest of the year. We assume this pattern matches the user behaviour for the MediaWiki application.

Thus, *average number of users* is assumed by:

$$\overline{USER(t)} = \begin{cases} 5000 \frac{users}{month} & 0 \leq (t \bmod 12) < 1 \\ 3200 \frac{users}{month} & 1 \leq (t \bmod 12) < 3 \\ 2000 \frac{users}{month} & 3 \leq (t \bmod 12) < 12 \end{cases} \tag{7.2}$$

Workload $w_0$'s average probability of occurrence is 70%, $w_1$'s 30%.

$$\overline{p(w_j)} = \begin{cases} 70\% & w_j = w_0 \\ 30\% & w_j = w_1 \end{cases} \tag{7.3}$$

The *Amazon EC2 Service Level Agreement*[3] contains a service commitment to a monthly availability of at least 99.95%. This ratio is used for both, the DBaaS and the IaaS Cloud service. IT estimates one hour per month downtime for updates and maintenance of the physical server. As experience shows the physical server is also unavailable due to failures for half an hour per month in average. It can therefore be assumed that the physical server's availability is

[3] As at June 1, 2013

99,80%. The *average availability at time t of* $T_\mu^i$ is defined by:

$$\overline{AV(T_\mu^i, t)} = \begin{cases} 99.80\% & T_\mu^i = T_\mu^0 \\ 99.80\% * 99.95\% \approx 99.75\% & T_\mu^i = T_\mu^1 \\ 99.80\% * 99.95\% \approx 99.75\% & T_\mu^i = T_\mu^2 \end{cases} \tag{7.4}$$

We assume that the users' satisfaction is lower in January. The pattern in figure 7.3 displays more edits in January. The assumption is that this will lead to lower satisfaction. This ratio corresponds with the percentage of aborted transactions due to unfulfilled requirements. The satisfaction under $T_\mu^0$ is significant lower then under $T_\mu^1$ and $T_\mu^2$. The reason is to be seen in the lack of possibilities to react to rapid load variations. Both Cloud services provide the advantage of vertical and horizontal scaling on demand.

$$\overline{SAT(T_\mu^0, t)} = \begin{cases} 71.0\% & 0 \leq (t \bmod 12) < 1 \\ 82.0\% & 1 \leq (t \bmod 12) < 12 \end{cases} \tag{7.5}$$

$$\overline{SAT(T_\mu^1, t)} = \begin{cases} 96.0\% & 0 \leq (t \bmod 12) < 1 \\ 97.0\% & 1 \leq (t \bmod 12) < 12 \end{cases} \tag{7.6}$$

$$\overline{SAT(T_\mu^2, t)} = \begin{cases} 98.0\% & 0 \leq (t \bmod 12) < 1 \\ 99.9\% & 1 \leq (t \bmod 12) < 12 \end{cases} \tag{7.7}$$

**adaptation costs** April till December requires less adaptations than the rest of the year (following figure 7.3). We assume Mondays, Tuesdays, Wednesdays and Thursdays requires vertical scaling of $T_\mu^1$ for five hours per day in this time span. The adaptation costs are 0.50$ per hour. The resulting costs sum up to 42.00$ per month.

We assume more adaptations are necessary for January till March. $T_\mu^1$ requires vertical scaling for three hours per day at weekends (0.50$ per hour). The fulfilment of requirements requires vertical scaling for six hours per workday (0.50$ per hour) and a replica inside the persistence tier for 2 hours (1.20$ per hour). The resulting costs sum up to 125.00$ per month.
$T_\mu^2$ requires a replica of the database in another location inside the European Union for 2 hours on Tuesdays and Wednesdays (3.50$ per hour). The resulting costs sum up to 120.00$ per month.

$$\overline{adap_{T_\mu^0}(t)} = 0 \tag{7.8}$$

$$\overline{adap_{T_\mu^1}(t)} = \begin{cases} 125.00 \frac{\$}{month} & (t \bmod 12) < 3 \\ 42.00 \frac{\$}{month} & 3 \leq (t \bmod 12) < 12 \end{cases} \tag{7.9}$$

$$\overline{adap_{T_\mu^2}(t)} = \begin{cases} 120.00 \frac{\$}{month} & (t \bmod 12) < 3 \\ 0.00 \frac{\$}{month} & 6 \leq (t \bmod 12) < 12 \end{cases} \tag{7.10}$$

### 7.2.2 Revenue

With equations 4.2 and 4.3 the expected revenue is:

$$rev_{exp}(T_\mu^i, W, T) = \int_{t_{min}}^{t_{max}} \sum_{j=0}^{j=m} p(w_j, t) * \overline{USER(t)} * \overline{TPU(w_j)} * \overline{RPT(t)} * \overline{sat(T_\mu^i, t)} * \overline{AV(T_\mu^i, t)} \tag{7.11}$$

In order to enable the calculation of utility, the subfunctions contained in 7.11 have to be defined. $p(w_j)$, $\overline{USER(t)}$, $\overline{TPU(w_j)}$, $\overline{AV(T_\mu^i, t)}$ and $\overline{SAT(T_\mu^0, t)}$ are already defined (see section 7.2.1).

The considered MediaWiki application is funded exclusively by donations. Without influences, we assume there is a constant willingness to donate. The company is running a campaign from November till December to increase donations. The pattern repeats every year. The *average revenue per transaction* is assumed by:

$$\overline{RPT(t)} = \begin{cases} 0.18\$ & (t \bmod 12) < 10 \\ 0.35\$ & 10 \leq (t \bmod 12) < 12 \end{cases} \tag{7.12}$$

### 7.2.3 Resource Modeling

In order to enable Kereta's decision-support, resources have to be created. The following sections cover the creation of the resources which are necessary to use Kereta's decision support.

#### Revenue Function

In order to simplify the revenue expression six *Function* resources are created . These resources are reused by the *Function* resource that represents $rev_{exp}(T_\mu^i, W, T)$. The resources for $p(w_j, t)$, $USER(t)$, $TPU(w_j)$, $RPT(t)$, $SAT(T_\mu^i, t)$ and $AV(t)$ are outlined below. Ratios are hard-coded. It is possible to replace values (e.g. $AV(t)$: 99.80%, 99.75% and 99.75%) with parameters. That increases reusability but leads to more complex evaluations (e.g. more parameter assignments).

**workload w's probability of occurrence**
alias: *pt_pro*
parameter: *w* (identifier of workload, data type: *number*)
expression:

```
1 IFF(EQU(w,0), 0.7) + IFF(EQU(w,1), 0.3)
```

**Listing 7.1:** Expression: Workload Probability

**average number of users**
alias: *pt_users*
parameter: *t* (time period, data type: *number*)
expression:

```
IFE(
  <(MOD(t,12), 1), 5000,
    IFE(<(MOD(t,12),3), 3200, 2000)
)
```

**Listing 7.2:** Expression: Average Number of Users

**average number of transactions per user**
alias: *pt_tpu*
parameter: *w* (identifier of workload, data type: *number*)
expression:

```
IFF(EQU(w,0),5) + IFF(EQU(w,1),12)
```

**Listing 7.3:** Expression: Average Number of Transactions

**average revenue per transaction**
alias: *pt_rpt*
parameter: *t* (time period, data type: *number*)
expression:

```
IFE(<(MOD(t,12), 10),0.10, 0.25)
```

**Listing 7.4:** Expression: Average Revenue per Transaction

**average user satisfaction**
alias: *pt_sat*
parameter: *t* (time period, data type: *number*)
parameter: *T* (identifier of the topology, data type: *number*)
expression:

```
IFF(
  EQU(T, 0),
  IFE(<(MOD(t,12),1), 0.71, 0.82) ) +
IFF(
  EQU(T, 1),
  IFE(<(MOD(t,12),1), 0.96, 0.97) ) +
IFF(
  EQU(T, 2),
```

```
9   IFE(<(MOD(t,12),1), 0.98, 0.999) )
```

**Listing 7.5:** Expression: Average User Satisfaction

**average availability**
alias: *pt_av*
parameter: *t* (point in time, data type: *number*)
expression:

```
1 IFF(EQU(T,0), 0.998) +
2 IFF(EQU(T,1), 0.9975) +
3 IFF(EQU(T,2), 0.9975)
```

**Listing 7.6:** Expression: Average Availability

It is now easy to create the *Function* resource which represents formula 7.11. The XML representation is:

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <function>
3    <id>4c5f5308-c1f4-49e7-b50f-1c36edd14ac4</id>
4    <alias>pt_rev</alias>
5    <formula>
6      IGR_t(
7          SUM_w(
8              FCT(pt_pro, w:w) *
9            FCT(pt_users, t:t) *
10             FCT(pt_tpu, w:w) *
11             FCT(pt_rpt, t:t) *
12           FCT(pt_sat, t:t$T:To) *
13             FCT(pt_av, T:To)
14         )
15     )
16   </formula>
17   <description>
18     Prototype, revenue function;
19     t: time interval,
20     To: topology numbering,
21     w: workload numbering
22   </description>
23   <functionType>revenue</functionType>
24   <author>mackfn</author>
25   <create>2016-01-08T17:10:21+01:00</create>
26   <links>
```

```
    <link rel="parameters">
      /Function/4c5f5308-c1f4-49e7-b50f-1c36edd14ac4/Parameter
    </link>
    <link rel="this">
      /Function/4c5f5308-c1f4-49e7-b50f-1c36edd14ac4
    </link>
  </links>
</function>
```

**Listing 7.7:** XML Representation: Revenue Function Resource

*Parameter* resources allow for the definition of default values, descriptions and data types. These resources increase the usability.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<Parameters>
  <parameter>
    <name>t</name>
    <dataType>array of numbers</dataType>
    <defaultValue>[0,0]</defaultValue>
    <description>length: 2; [t_min, t_max]</description>
    ...
  </parameter>
  <parameter>
    <name>To</name>
    <dataType>number</dataType>
    <defaultValue>0</defaultValue>
    <description>Topology numbering</description>
    ...
  </parameter>
  <parameter>
    <name>w</name>
    <dataType>array of numbers</dataType>
    <defaultValue>[0,0]</defaultValue>
    <description>
      length: 2;
      [w_0, w_n]
      numbering of the first and last workload
    </description>
    ...
  </parameter>
</Parameters>
```

**Listing 7.8:** XML Representation: Parameter Resources

#### Cost Function

Equation 4.4 defines function $cost(T_\mu^i, W, R, T)$. The first term $cost_{fixed}(T_\mu^i, T)$ doesn't require a *Function* resource. The Nefolog framework is utilized to calculate the costs of viable topology $T_\mu^i$ over time interval $T$. The second term $\sum_{k=1}^{o} cost_{adaptation}(T_\mu^{i,k}, W, R, T_k)$ requires the observation and/or prediction of the execution of workload. Section 7.2.1 includes the results of the workload analysis. The concept in section 4.1.1 calculates the sum over all adaptation costs in the suggested time interval. In this example the sum is calculated by evaluating the following function:

$$\int_{t_{min}}^{t_{max}} \overline{adap_{T_\mu^i}(t)} \tag{7.13}$$

$\overline{adap_{T_\mu^i}(t)}$ is defined by formula 7.9 and 7.10. The corresponding *Function* resource's representation (in part) is:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<function>
  <id>2929801b-edd0-427b-bffe-5c6470af492b</id>
  <alias>pt_adap</alias>
  <formula>
    IGR_t(
    IFF(
        EQU(To,1),
        IFE( <(MOD(t,12),3), 125, 42 )
    ) +
    IFF(
        EQU(To,2),
        IFE( <(MOD(t,12),3), 120, 0 )
    ))
  </formula>
  <description>
    Prototype, cost function;
    t: time interval,
    To: topology numbering
  </description>
  <functionType>cost</functionType>
  ...
</function>
```

**Listing 7.9:** XML Representation: Cost Function Resource

Nested *Parameter* resources looks like the *Parameter* resources of the revenue *Function* resource (without the resource for parameter $w$).

#### Application and Topology Resources

Next, a *Application* resource and nested *Tier* and *Requirement* resources has to be created. Furthermore *Distribution*, *Offering* and *Performance* resource has to be created for each alternative distribution.

The *Application* resources alias is *pt_app*, the *Distribution* resources' *pt_Prms*, *pt_IaaS* and *pt_-DBaaS*.

#### Utility Function Resource

It is possible to create the *Utility Function* resource and nested *SubFunction* resources for one *Distribution* resource and then clone them for the other *Distribution* resources.

The costs for the physical server are ignored. Each distribution contains this matter of expense. Thus, it will not have an impact on the decision. *Distribution pt_Prms*'s *Utility Function* resource's alias is *pt_uf-0*. *SubFunction* resources represent links to the already created *cost* and *revenue Function* resources.

The *Utility Function* and *SubFunction* resources' representations (in part) looks like:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<utilityFunction>
  <id>282b087f-7939-4a13-bfcd-87b8a684b98b</id>
  <alias>pt_uf-0</alias>
  <description>
    Prototyp Utility Function for pt_Prms
  </description>
  <distributionId>
    9a2194ff-eaa4-4d0e-890a-8f898de73e53
  </distributionId>
  ...
</utilityFunction>
```

**Listing 7.10:** XML Representation: Utility Function Resource for $T_{\mu}^{0}$

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<subfunctions>
  <subFunction>
    <functionId>4c5f5308-c1f4-49e7-b50f-1c36edd14ac4</functionId>
    <number>1</number>
    <utilityFunctionId>
      282b087f-7939-4a13-bfcd-87b8a684b98b
    </utilityFunctionId>
```

```
    ...
  </subFunction>
  <subFunction>
    <functionId>2929801b-edd0-427b-bffe-5c6470af492b</functionId>
    <number>2</number>
    <utilityFunctionId>
      282b087f-7939-4a13-bfcd-87b8a684b98b
    </utilityFunctionId>
    ...
  </subFunction>
```

**Listing 7.11:** XML Representation: Sub-Function Resources for $T_{\mu}^{0}$

This resources are cloned for the other *Distribution* resources, but both *Distribution* resources (*pt_IaaS* and *pt_DBaaS*) requires a third *SubFunction* resource. The additional resource enables the cost calculation by the Nefolog framework. The Nefolog configuration-IDs are 289 (IaaS) and 187 (DBaaS). The resource's XML representations are (in part):

*pt_IaaS*:

```
<subFunction
  <functionId>nefolog$289</functionId>
  <number>3</number>
  <utilityFunctionId>
    687b5a5f-74e1-41ac-a6c8-e6f350c14274
  </utilityFunctionId>
  <links>
    <link rel="parameters">
      /UtilityFunction/687b5a5f-74e1-41ac-a6c8-e6f350c14274/
         SubFunction/3/NefologParameter
    </link>
  </links>
</subFunction>
```

**Listing 7.12:** XML Representation: Nefolog Sub-Function Resource for $T_{\mu}^{1}$

*pt_DBaaS*:

```
<subFunction>
  <functionId>nefolog$187</functionId>
  <number>3</number>
  <utilityFunctionId>
    a00bc6d5-6b77-4ce3-9807-c162e41be21a
  </utilityFunctionId>
  <links>
```

```
8       <link rel="parameters">
9         /UtilityFunction/a00bc6d5-6b77-4ce3-9807-c162e41be21a/
              SubFunction/3/NefologParameter
10      </link>
11    </links>
12  </subFunction>
```

**Listing 7.13:** XML Representation: Nefolog Sub-Function Resource for $T^2_\mu$

### 7.2.4 Decision Support

All necessary resources have been created, but *SubFunction* resources still missing parameter assignments. After the assignments the decision support is operational.

The application should run for 15 month from February 1, 2015 to April 31, 2016, thus, $T = [1, 16]$. Two workloads are suggested ($w = [0, 1]$) and *To* corresponds with the viable topology's numbering (0, 1, or 2).

#### Parameter Assignments

The *UtiltyFunction* resource for *pt_Prms* has two nested *SubFunction* resources, the *Utilty-Function* resources for *pt_IaaS* and *pt_DBaaS* have the additional resource for the Nefolog cost calculation.

*Revenue*

- /UtilityFunction/pt_uf-0/SubFunction/1/assign?key=pt&w=[0,1]&t=[1,16]&To=0
- /UtilityFunction/pt_uf-1/SubFunction/1/assign?key=pt&w=[0,1]&t=[1,16]&To=1
- /UtilityFunction/pt_uf-2/SubFunction/1/assign?key=pt&w=[0,1]&t=[1,16]&To=2

*Cost*

- /UtilityFunction/pt_uf-1/SubFunction/2/assign?key=pt&w=[0,1]&t=[1,16]&To=0
- /UtilityFunction/pt_uf-2/SubFunction/2/assign?key=pt&w=[0,1]&t=[1,16]&To=1
- /UtilityFunction/pt_uf-3/SubFunction/2/assign?key=pt&w=[0,1]&t=[1,16]&To=2

*Nefolog*
URIs

- */UtilityFunction/pt_uf-1/SubFunction/3/NefologParameter*
- */UtilityFunction/pt_uf-2/SubFunction/3/NefologParameter*

provides XML documents which contains parameters for the cost calculation.

```
<parameters>
  <parameter>Hour</parameter>
  <parameter>Month</parameter>
  <parameter>GBExternalNetworkEgress</parameter>
  <parameter>location_zone</parameter>
  <parameter>usage_pattern</parameter>
</parameters>
```

**Listing 7.14:** XML Representation: Nefolog Parameters for $T_{\mu}^{1}$

```
<parameters>
  <parameter>Hour</parameter>
  <parameter>i/oOperation</parameter>
  <parameter>GBStorage</parameter>
  <parameter>Month</parameter>
  <parameter>GBExternalNetworkEgress</parameter>
  <parameter>location_zone</parameter>
  <parameter>usage_pattern</parameter>
</parameters>
```

**Listing 7.15:** XML Representation: Nefolog Parameters for $T_{\mu}^{2}$

These Cloud services should run 24/7 for 15 month. Location must be inside the European Union. Parameter *GBStorage* has a strong influence on the result. Missing the assignment for *GBStorage* results in excessive costs for the DBaaS service. Therefore, the parameter is set to 1000 (1 terrabyte).

- /UtilityFunction/pt_uf-2/SubFunction/3/assign?key=pt&Hour=10800&Month=15&location_zone=EU
- /UtilityFunction/pt_uf-3/SubFunction/3/assign?key=pt&Hour=10800&Month=15&location_zone=EU&GBStorage=1000

#### Evaluation

Two URIs provide decision support. Kereta can create a list with the evaluation of each suggested viable topology or just select the fittest topology.

- /Application/pt_app/compare?pt_Prms=pt_uf-0:pt&pt_IaaS=pt_uf-1:pt&pt_DBaaS=pt_uf-2:pt
- /Application/pt_app/select?pt_Prms=pt_uf-0:pt&pt_IaaS=pt_uf-1:pt&pt_DBaaS=pt_uf-2:pt

The first URI returns the following XML representation:

```
<listing>
  <calculation class="utilityFunction">
    <result>34824.82</result>
    <distributionId>
      6617d158-da36-4913-a60e-cb986d9461a2
    </distributionId>
    <subCalculations>
      <calculation class="subfunction" number="1" type="revenue">
        <result>52741.32</result>
        ...
      </calculation>
      <calculation class="subfunction" number="2" type="cost">
        <result>594.00</result>
        ...
      </calculation>
      <calculation class="subfunction" number="3" type="cost">
        <result>17322.50</result>
        ...
      </calculation>
    <subCalculations>
  </calculation>

  <calculation class="utilityFunction">
    <result>44531.99</result>
    <distributionId>
      5831e0d0-3a8b-453c-a963-fc1ec3bfd681
    </distributionId>
    <subCalculations>
      <calculation class="subfunction" number="1" type="revenue">
        <result>51266.84</result>
        ...
      </calculation>
      <calculation class="subfunction" number="2" type="cost">
        <result>1040.85</result>
        ...
      </calculation>
      <calculation class="subfunction" number="3" type="cost">
        <result>5694.00</result>
        ...
      </calculation>
    </subCalculations>
  </calculation>

  <calculation class="utilityFunction">
```

```
45      <result>42680.74</result>
46      <distributionId>
47        9a2194ff-eaa4-4d0e-890a-8f898de73e53
48      </distributionId>
49      <subCalculations>
50        <calculation class="subfunction" number="1" type="revenue">
51          <result>42680.74</result>
52          ...
53        </calculation>
54        <calculation class="subfunction" number="2" type="cost">
55          <result>0.00</result>
56          ...
57        </calculation>
58      </subCalculations>
59    </calculation>
60  </listing>
```

**Listing 7.16:** XML Representation: Compare Alternative Distributions

| Rank | Alias | Distribution | Utility | Revenue | Cost |
|---|---|---|---|---|---|
| 1 | pt_IaaS | $T_{\mu}^{1}$ | 44,531.99$ | 51,266.84$ | 6,734.85$ |
| 2 | pt_Prms | $T_{\mu}^{0}$ | 42,680.74$ | 42,680.74$ | 0.00$ |
| 3 | pt_DBaaS | $T_{\mu}^{2}$ | 34,824.82$ | 52,741.32$ | 17,916.50$ |

**Table 7.1:** Ranking based on the Utility

The ranking is: $T_{\mu}^{1} > T_{\mu}^{0} > T_{\mu}^{2}$. The summarized results (table 7.1) shows that *Distribution pt_IaaS* provides the highest utility (44,531.99$). Furthermore, users can read, that *pt_DBaaS* generates the highest revenue (52,741.32$) but also comes with the highest costs (17.916,50$). The main cost driver is the DBaaS offering. This distributed deployment promises the best end-user experience (measured on the basis of revenue), it is nevertheless not the best choice. Utility breaks different aspects into one axis [STFG08]. Higher end-user experience could not equal higher costs in this particular case. Between *pt_IaaS* and *pt_Prms* the reverse is true. Thus, *pt_IaaS* is the best choice.

## 7.3 Discussion

The example in section 7.1 shows that the concept presented in section 4.1.1 can be used to calculate the expected utility of different viable topologies. The connection to Nefolog enables a convenient calculation of distributions' fixed costs. The task of keeping data up-to-date can be left to the Nefolog vendor. The process of parameter assignments is a little bit complicated and surly violated the REST constraint stateless. On the other hand, usability is improved and error-prone, extensive long string queries are prevented.

Decision support not only delivers the topology which promises the highest utility, but also provides numbers that support a analysis of the cause. In this way, it is possible to evaluate which changes will alter the decision. The implementation of the parser and the expression evaluator fulfil their purpose. There occurs no limitations for the descriptions of formulas in section 7.2.

# 8 Outcome and Future Work

Chapter 2 presents concepts and technologies which are necessary as a basis for this thesis. The chapter also introduces the *Winery Modeling Tool* and the *Nefolog* system. Both tools are utilized in the following concept for the *Utility Calculation Framework*. The framework follows the also mentioned architectural style REST. Chapter 3 presents existing approaches for using utility theory in software architectures. The comparison shows that there are different ways to handle the trade-off between e.g. cost and performance. The two main identified concepts are (1) focussing on the financial impact and value each aspect with an amount of money and (2) defining for each aspect a mapping that transfers a quality (e.g. availability, costs, CPUs) into a uniform scale (e.g. $U(q) \rightarrow [0, 1]$). The utility function concept in Chapter 4 is focussing on the profitability of business applications, and therefore is based on concept (1). The analysis of existing approach carried out a way to handle uncertainties with respect to the actual perceived QoS.

The utility function concept determines a viable distribution's utility by calculating the expected revenue, the distribution's fixed costs and the adaptation costs. Chapter 4 contains the exemplary application of the utility function concept. Furthermore, requirements, use cases and the system overview provides the foundations towards a framework which allows for the utility-based evaluation of the different deployment alternatives for cloud applications. Chapter 5 is based on this work and describes the design of the *Utility Calculation Framework*. Central components are a repository, a module that enables the evaluation of functions and a RESTful Interface. The repository manages resources like *Application Descriptions*, *Distributions*, *Utility Functions* and reusable *Functions*. A major focus lies on providing reusable *Functions* that allows for the simplify development of customized utility functions. Users also have the possibility to create their own *Function* resources. Therefore, Chapter 5 defines the mandatory syntax and semantic for mathematical expressions. The RESTful API should provide a uniform interface to the system's functionality. The design follows the REST constraints outlined in Chapter 2.

Chapter 6 presents the actual implementation of the framework designed in Chapter 5. Key points are the overview of resources URIs (table 6.1), resources' XML representation and the implementation's functionality, e.g. calculation and decision support. The name of the prototypical implementation is *Kereta*. *Kereta* implements the repository, the module that evaluates functions and the RESTful Interface. Finally, *Kereta* and the utility function concept is evaluated by performing a decision process from scratch in Chapter 7.

The utility function concept in Chapter 4 strongly addresses applications in B2B and B2C scenarios. A future research could be focussed on other scenarios and it may be necessary to think about other concepts then suggesting the profitability of applications. Furthermore,

the concept does not adapt the above mentioned handling of uncertainties. This could improve the decision support in cases where decisions rely on probabilistic forecasts. One additional, interesting challenge is the seamless integration of the *Utility Calculation Framework* in the value chain from specifying an application till the application's deployment and dynamically redeployment. Forecasts could influence earlier decisions and the back coupling of business reports and the results of repeated workload analysis could improve further decision.

# Bibliography


[ABLt13] V. Andrikopoulos, T. Binz, F. Leymann, and S. trauch. How to Adapt Applications for the Cloud Environment. *Computing*, 95:493–535, 2013.

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Commun. ACM*, 53(4):50–58, April 2010.

[AGSLW14] V. Andrikopoulos, S. Gómez Sáez, F. Leymann, and J. Wettinger. Optimal Distribution of Applications in the Cloud. In M. Jarke, J. Mylopoulos, C. Quix, C. Rolland, Y. Manolopoulos, H. Mouratidis, and J. Horkoff, editors, *Advanced Information Systems Engineering*, volume 8484 of *Lecture Notes in Computer Science*, pages 75–90. Springer International Publishing, 2014.

[ARSL14] V. Andrikopoulos, A. Reuter, S. G. Sáez, and F. Leymann. A GENTL Approach for Cloud Application Topologies. In *Proceedings of the Third European Conference on Service-Oriented and Cloud Computing (ESOCC 2014)*, Lecture Notes in Computer Science (LNCS), pages 1–11. Springer, September 2014.

[ARXL14] V. Andrikopoulos, A. Reuter, M. Xiu, and F. Leymann. Design Support for Cost-Efficient Application Distribution in the Cloud. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 697–704, June 2014.

[AS13] W. Al Shehri. CLOUD DATABASE DATABASE AS A Service. *International Journal of Database Management Systems*, 5(2):1–12, 2013.

[ASL13] V. Andrikopoulos, Z. Song, and F. Leymann. Supporting the Migration of Applications to the Cloud through a Decision Support System. In *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD 2013), June 27-July 2, 2013, Santa Clara Marriott, CA, USA*, pages 565–572. IEEE Computer Society, July 2013.

[BBKL14] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. *TOSCA: Portable Automated Deployment and Management of Cloud Applications*, chapter TOSCA: Portable Automated Deployment and Management of Cloud Applications, pages 527–549. Springer, New York, January 2014.

[BEDL+03] R. Bardohl, H. Ehrig, J. De Lara, O. Runge, G. Taentzer, and I. Weinhold. *Node type inheritance concept for typed graph transformation*. Technische Universität Berlin, Fakultät IV-Elektrotechnik und Informatik, 2003.

[BSW14] A. Brogi, J. Soldani, and P. Wang. TOSCA in a Nutshell: Promises and Perspectives. In M. Villari, W. Zimmermann, and K.-K. Lau, editors, *Service-Oriented and Cloud Computing*, volume 8745 of *Lecture Notes in Computer Science*, pages 171–186. Springer Berlin Heidelberg, 2014.

[CSM14] CSMIC. Service Measurement Index Framework 2.1. `http://csmic.org/downloads/SMI_Overview_TwoPointOne.pdf`, July 2014.

[EAA+04] M. Endrei, J. Ang, A. Arsanjani, S. Chua, P. Comte, P. Krogdahl, D. M. Luo, and T. Newling. *Patterns: Service-Oriented Architecture and Web Services*. IBM Redbooks, 2004.

[FGM+99] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, Internet Engineering Task Force, June 1999.

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, 2000. AAI9980887.

[Fis70] P. Fishburn. *Utility theory for decision making*. Publications in operations research. Wiley, 1970.

[GB08] M. Galster and E. Bucherer. A Taxonomy for Identifying and Specifying Non-Functional Requirements in Service-Oriented Development. In *Services - Part I, 2008. IEEE Congress on*, pages 345–352, July 2008.

[GGS10] J. Gutierrez-Garcia and K.-M. Sim. Self-Organizing Agents for Service Composition in Cloud Computing. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 59–66, Nov 2010.

[GSAGL14] S. Gómez Sáez, V. Andrikopoulos, K. Ganguly, and F. Leymann. Enriching Cloud Application Topologies with Evolving Performance and Workload Models. In *TODO*, page TODO, 2014.

[GSALS14] S. Gomez Saez, V. Andrikopoulos, F. Leymann, and S. Strauch. Towards Dynamic Application Distribution Support for Performance Optimization in the Cloud. In *Cloud Computing (CLOUD), 2014 IEEE 7th International Conference on*, pages 248–255, June 2014.

[GVB11] S. Garg, S. Versteeg, and R. Buyya. SMICloud: A Framework for Comparing and Ranking Cloud Services. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 210–218, Dec 2011.

[HS10] C.-W. Hang and M. Singh. From Quality to Utility: Adaptive Service Selection Framework. In P. P. Maglio, M. Weske, J. Yang, and M. Fantinato, editors, *Service-Oriented Computing*, volume 6470 of *Lecture Notes in Computer Science*, pages 456–470. Springer Berlin Heidelberg, 2010.

[JM12] Y. Jadeja and K. Modi. Cloud computing - concepts, architecture and challenges. In *Computing, Electronics and Electrical Technologies (ICCEET), 2012 International Conference on*, pages 877–880, March 2012.

[Joh07] T. Johnson. Utility Theory. *C2922 Economics*, 2007.

[JR01] G. Jehle and P. Reny. *Advanced Microeconomic Theory*. Addison-Wesley series in economics. Addison-Wesley, 2001.

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann. Winery – Modeling Tool for TOSCA-based Cloud Applications. In *11th International Conference on Service-Oriented Computing*, LNCS. Springer, 2013.

[KDM13] R. Karim, C. Ding, and A. Miri. An End-to-End QoS Mapping Approach for Cloud Service Selection. In *Services (SERVICES), 2013 IEEE Ninth World Congress on*, pages 341–348, June 2013.

[KM14] S. Kheradmand and M. Meybodi. Price and QoS competition in cloud market by using cellular learning automata. In *Computer and Knowledge Engineering (ICCKE), 2014 4th International eConference on*, pages 340–345, Oct 2014.

[LRBK10] S. Leimeister, C. Riedl, M. Böhm, and H. Krcmar. The Business Perspective of Cloud Computing: Actors, Roles, and Value Networks. In *Proceedings of 18th European Conference on Information Systems (ECIS 2010)*, Pretoria, South Africa, 2010.

[LS10] W. Lehner and K.-U. Sattler. Database as a service (DBaaS). In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 1216–1217, March 2010.

[LWW+10] M. Litoiu, M. Woodside, J. Wong, J. Ng, and G. Iszlai. A business driven cloud optimization architecture. In *in SAC, 2010*, pages 380–385, 2010.

[Max05] E. M. Maximilien. Agent-based trust model involving multiple qualities. In *In Proc. of the 4th Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, pages 519–526. AAMAS, ACM, 2005.

[MF11] D. Minarolli and B. Freisleben. Utility-based resource allocation for virtual machines in Cloud computing. In *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 410–417. IEEE, 2011.

[MG11] P. M. Mell and T. Grance. SP 800-145. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States, 2011.

[Nor99] J. Norstad. An introduction to utility theory. *Unpublished manuscript at http://homepage. mac. com/j. norstad*, 1999.

[OMN+04] D. Orchard, F. McCabe, E. Newcomer, H. Haas, C. Ferris, D. Booth, and M. Champion. Web Services Architecture. W3C note, W3C, February 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

[Org06] Organization for the Advancement of Structured Information Standards. *Reference Model for Service Oriented Architecture 1.0*. OASIS, July 2006.

[otSS11] E. A. D. of the Systems and N. A. C. (SNAC). Guidelines for Implementation of REST. Report I73-015R-2011, National Security Agency, 9800 Savage Rd. Suite 6704 Ft. Meade, MD 20755-6704, mar 11.

[PG03] M. Papazoglou and D. Georgakopoulos. Service-oriented computing. *Commun. ACM*, 46(10):24–28, 2003.

[PTDL07] M. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-Oriented Computing: State of the Art and Research Challenges. *Computer*, 40(11):38–45, Nov 2007.

[PvdH11] M. Papazoglou and W. van den Heuvel. Blueprinting the Cloud. *Internet Computing, IEEE*, 15(6):74–79, Nov 2011.

[PZJ14] J. Park, A. Zomaya, and H. Jeong. *Frontier and Innovation in Future Computing and Communications*. Springer, 2014.

[RCL09] B. Rimal, E. Choi, and I. Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 44–51, Aug 2009.

[RR07] L. Richardson and S. Ruby. *Restful Web Services*. O'Reilly, first edition, 2007.

[Sáe14] S. G. Sáez. Design Support for Performance-aware Cloud Application (Re-)Distribution. In *Proceedings of the PhD Symposium at the 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC 2014)*, pages 6–11. Jenaer Schriften zur Mathematik und Informatik, September 2014.

[SALS14] S. G. Sáez, V. Andrikopoulos, F. Leymann, and S. Strauch. Design Support for Performance Aware Dynamic Application (Re-)Distribution in the Cloud. *IEEE Transactions on Service Computing*, pages 1–14, December 2014.

[Sin15] M. Singh. Study on cloud computing and cloud database. In *Computing, Communication Automation (ICCCA), 2015 International Conference on*, pages 708–713, May 2015.

[SP12] J. Siegel and J. Perdue. Cloud Services Measures for Global Use: The Service Measurement Index (SMI). In *SRII Global Conference (SRII), 2012 Annual*, pages 411–415, July 2012.

[STFG08] J. D. Strunk, E. Thereska, C. Faloutsos, and G. R. Ganger. Using Utility to Provision Storage Systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 21:1–21:16, Berkeley, CA, USA, 2008. USENIX Association.

[UK12] G. Ucoluk and S. Kalkan. *Introduction to Programming Concepts with Case Studies in Python*. SpringerLink : Bücher. Springer Vienna, 2012.

[VRMCL08a] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A break in the clouds: towards a cloud definition. *ACM SIGCOMM Computer Communication Review*, 39(1):50–55, 2008.

[VRMCL08b] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

[XA13] M. Xiu and V. Andrikopoulos. The Nefolog & MiDSuS Systems for Cloud Migration Support. Technischer Bericht Informatik 2013/08, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, Universität Stuttgart, Institut für Architektur von Anwendungssystemen, November 2013.

[YMBD14] Z. Ye, S. Mistry, A. Bouguettaya, and H. Dong. Long-term QoS-aware Cloud Service Composition using Multivariate Time Series Analysis. *Services Computing, IEEE Transactions on*, PP(99):1–1, 2014.

[YZL07] T. Yu, Y. Zhang, and K.-J. Lin. Efficient Algorithms for Web Services Selection with End-to-end QoS Constraints. *ACM Trans. Web*, 1(1), May 2007.

All links were last followed on January 18, 2016

## Declaration

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

Stuttgart, January 18, 2016 _____

(Name)