Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Automatic interpretation of a declarative cloud service description

Vadim Raskin

Course of Study:	Infotech
Examiner:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Supervisors:	Dipl.-Inf. Tim Waizenegger, Dipl. -Phys. Cataldo Mega, IBM
Commenced:	November 1, 2014
Completed:	May 3, 2015

Abstract

The accelerated move from on-premise enterprise software to a cloud service model introduces certain challenges to service providers. The difference between enterprise customer's workloads and the complexity of software products create the need for a formal service description. It increases abstraction level, encompasses business requirements and eliminates misunderstanding between a service provider and its customers. However, the declarative nature of the description does not allow to determine precise implementation components. Furthermore, each provider is interested in customization and portability of its services to meet requirements of several customers, at the same time allowing automatic selection of service topology. The major objective of this work is to show how declarative cloud service description can be automatically processed, analyzed and mapped to the service topology matching customer's workload and original business requirements. In order to decrease service delivery time and to eliminate the manual selection of application components, a method of automatic interpretation of a declarative cloud services is proposed.

In this work, several solution concepts of interpretation of a declarative service description were discussed. As a result, a method of automatic identification of service components matching to a given business requirements was elaborated. It is based on the gradual reduction of the list of possible service components considering their compatibility and deployment sequence. Moreover, it includes an optimization algorithm that selects exact service topology for the case of several matching options. Additionally, a prototype of an interpreter that transforms declarative cloud service description into the exact topology of components was implemented and discussed.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Recently traditional computing is more and more being replaced by the cloud service model. The new approach is providing a win-win situation for both users and service providers. From the user's perspective, he has to pay only for the consumed resources, which allows cutting partially its *Total Cost of Ownership* (TCO), including capital as well as operational costs [Cha09]. At the same time service provider enjoys the effect of economies of scale by delivering its services to many customers. In the case of B2B model switch from traditional software deployment to cloud services does not seem to be straightforward due to several reasons.

Software intended for enterprises has to reflect their internal business processes and cope with unique workloads, meaning that solution appropriate for one customer is not acceptable for the another. Enterprise software differs from end-user desktop applications in its customizability and involvement of dozens of hardware and software components that are required to be deployed. As a consequence, each B2B software solution has to be manually configured and should represent a unique combination of known components. Software solutions delivered in such a fashion are costly due to the involvement of subject matter experts in requirement analysis, employment of operation teams that manually adapt the software to customer needs. Consequently, it takes several months to deliver enterprise software. In order to further clarify the problems of switching from traditional computing to the cloud model, *Enterprise Content Management* (ECM) system is chosen as a reference. After a short introduction to ECM, aspects of the process of traditional software delivery of such system are considered, its drawbacks are highlighted and a possible solution is proposed.

ECM systems deal with information governance in big enterprises. In the case of small and medium companies, information governance could be handled manually without losses, or by keeping data in applications from different vendors scattered around the company. But in bigger enterprises lack of control on information and content can lead to lost revenue or non-compliance with the law. Therefore, the need for centralized management of scattered content arises.

At the very first step, business customers indicate their problems and ask an ECM system provider for a possible solution. Later, a list of system requirements is provided. Further clarification with subject matter experts is needed in order to eliminate ambiguous requirements and identify exact needs. Thus it is considered to be a long-term process due to misunderstanding based on term difference between the field of business problems and solution domain. Usually, customers are faced with the same problems regardless of their operation field but express

them in "different languages" specific to their operation domain. Hence, here arises the need for a common vocabulary with strict semantics that will allow users explicitly define their needs and increase service delivery time.

After processing of user requirements, solution architect manually determines the exact system topology. This step is taken over by operation team that orchestrates and deploys selected components. Customer specific nature of ECM systems requires fine-grain configuration and manual adaptation of orchestration scripts in the case of specific requirements. In other words, it leads to individual solutions that are built from scratch. Typically orchestration scripts are extremely fragile, which means that changed parameters in one component may lead to failures of the other ones. Hence, this process is error-prone, requires appropriate testing and increases the time of service delivery. In the case of several orders of the same software product with limited human resources in operation team, it even more increases waiting time for the customers in the queue. Taking into account contradiction of individual solution approach with cloud principles it becomes obvious that the next step towards cloud acceptance would be to have the list of generalized *services templates* with customizable parameters. The template represents a set of service components aggregated from best practices of already delivered solutions, including software components, network infrastructure, and virtual machine appliances. Each entity exposes the list of parameters that can be configured externally. Hence, there is a need for an application that interprets user requirements, determines appropriate service topology and initializes its deployment with user specific parameters.

Orchestration scripts are usually devoted to one specific environment and cannot be easily adapted to changed customer needs. If the system was delivered to the customer on premises, there is no future maintenance provided that guarantees portability of the system to another environment or conduction of software updates. Even if the software was deployed in the cloud, and customer was first time satisfied with it, once the enterprise grows it might be reasonable to install the software in own private cloud infrastructure or arrange a hybrid topology[1]. For these purposes, enterprise software deployment format must be portable between different environments.

The move from on-premise to cloud service delivery model has to cope with the following challenges: manual requirements engineering for each new customer, human involvement in selection of service components, lack of customizable deployment automation, portability of services between several cloud environments. In order to eliminate manual requirement engineering, a *Domain Specific Language* (DSL) must be defined. Its aim is to reflect user needs as well as ECM service provider capabilities. This language enables the user to express his requirements in a declarative form eliminating implementation's specification. Human action involvement in component selection is to be reduced by a *DSL interpreter*. The goal of this application is to encompass the knowledge of best practices of already delivered services. It is responsible for parsing of user defined DSL templates, analyzing its content, suggesting

[1] According to "State of the Cloud Survey 2015" conducted by Right Scale

the most appropriate service components and their mapping to the exact service templates. Predefined customizable service templates can solve the last two issues: lack of customizable deployment automation and portability of services. Depending on the exact cloud provider format, these templates could be portable to several environments with the customization features.

1.1 Scope of the work

The major objective of this work is to show how declarative cloud service description can be automatically processed, analyzed and mapped to the service topology matching customer's workload and original business requirements. A method of automatic interpretation of user requirements was elaborated. It is based on gradual reduction of the map of possible service component that leads to the selection of precise service topology. Determination of exact components is guaranteed by optimization algorithm that calculates the most appropriate service topology based on compatibility, deployment sequence and optimization metric. In order to prove the elaborated concept, a prototype of an interpreter was elaborated. It consumes DSL templates defined by the users and maps user input to one of the predefined service templates. At the end the prototype implementation was discussed followed by conclusions and highlighted future research directions.

Further content of this thesis is structured as follows:

- *Related work and Context.* These subsections follow up the introduction section of this thesis. Related work subsection gives more insight into the previously utilized approach of service delivery, as well as research on the transformation of requirements into service components, TOSCA based interpretation and automatic configuration of cloud infrastructures. Context subsection clarifies the main terms used in this work: cloud computing, portability of services and enterprise content management domain.
- *Solution concept.* This section proposes the concept of interpretation and introduces sample application architecture. User input is considered to be in the form of DSL. The result of the interpretation is defined as one of the predefined service templates. Several ways of handling of user input are considered, and the one based on a gradual reduction of possible service components is selected and details of this method are discussed. As part of the approach, an algorithm of optimized component topology selection is introduced. It is based on graph abstraction and requires the calculation of path's metrics. Furthermore, it introduces the concept of partial match of user requirements.
- *Prototype implementation.* The section introduces the proof of concept that was implemented in a Java-based application with web GUI. Application consumes TOSCA based ECM domain specific language templates as an input defined in XML format. Two sample DSL service templates are defined. They are interpreted by the application that maps

them to registered Heat Orchestration Templates (HOTs) and triggered its deployment with configured parameters. In order to emulate cloud provider environment, OpenStack infrastructure was installed. Furthermore, used technologies, details of implementation and application lifecycle are covered in this section. At the end of the section, the contribution of the prototype is discussed in the context of the switch to a cloud service model.

- *Summary.* This section provides conclusions that were drawn from elaborated work. Moreover, future research directions are emphasized.

1.2 Related work

In the following section, the traditional approach to service delivery is to be presented as well as consideration of work on the transformation of user requirements, portability of cloud services and automatic identification of cloud infrastructure configuration.

In the following an already existing approach of service delivery is to be presented, it was elaborated by ECM experts at IBM. The problem of interpretation of business requirements of ECM customers was previously carried out manually by means of questionaries filled out individually by each new customer. In this checklist, potential customer answers the questions regarding current issues in content management domain and shares its expectations from the system. From the architectural perspective, the customer is asked to mention technologies he is already using. In order to assess the capacity of the future system, possible workload characteristics were also included in the user requirements definition. For example, number document load operations per minute, average document size, number of searches per time unit, etc. After retrieving all the above mentioned data it is analyzed manually by domain experts, and appropriate service topology is selected. However, this approach requires the repeat of similar work items each time when a new customer comes, the lack of automation in the selection of the template increases the time of service delivery. If the number of new customers exceeds the actual number of domain experts, then delivery will be done in a sequential fashion, processing customer requests individually.

In their work Chang et al. [CLY+14] represent a model-driven development approach to transform from high abstraction level system model to the diagrams describing system architecture. Original system requirements are gathered by means of interviews, and afterward requirement diagram is elaborated. With the help of ATLAS modeling language, this diagram further transferred into Use Case Diagram and Activity Diagram according UML. The given method describes the principle of transformation based on predefined rules that map elements between models. However, it does not touch the process of elicitation of actual software components that will be contained in the service.

Either Binz et al. 2013[BBH+13] and Katsaros et al. [KML+14] present independent prototypes of applications that process services defined in the language following TOSCA specification [TOS13a]. Bitz et al. depict the architecture of the OpenTOSCA application, with an example of deployment of a cloud service. The application consists of the whole stack of tools needed to circumscribe the cloud service: modeling tool, runtime environment, and user interface for administrators and customers. From the application architecture point of view, it was underscored that the Cloud Service Archive (CSAR) must contain predefined set of plans to deploy and manage the service. Both solutions prove the concept that TOSCA described applications can be run independently of IaaS provider, but with a condition that provider specific extension is either enabled in the container or directly on the cloud provider side. In both works an imperative method of service deployment was utilized, e.g., the user is responsible for the definition of the components, their orchestration scripts and managing plans. However, they did not touch the topic of declarative service description that is utilized in this work.

In Wettinger et al. [WBB+14] authors make an overview of TOSCA capabilities to define a cloud service and suggestions on decoupling between service description and implementation of its deployment. The improvement they proposed for script invocation uses inversion of control pattern [PL12] to lighten management plan description and provide loose coupling between plans and orchestration scripts. In other words plan declared once can be maintained and changed over time, on the other hand, orchestration scripts (Shell, Python or another scripting language) can be reused in different cloud provider environments. The proposed solution is a good attempt to guarantee portability of cloud applications between several environments. However, the approach concentrates more on the management of cloud application, rather that transformation of business requirements into the exact implementation.

In their work Uchiumi et al. [UKKM13] present a method of automatic configuration of cloud infrastructures. The system is trained on a wide set of existing infrastructures. Based on this knowledge their algorithm can predict the configuration of a new data center. However, the focus of this work differs in a sense that application layer must be configured based on user input. Furthermore, deterministic analysis is utilized which does not leave space for training of the system.

1.3 Context

Cloud Computing

In order to be widely adopted a new technology has to add significant value, which could bring improved efficiency, reduced operational costs, increase profit, etc. Cloud computing is one of such technologies that allow to decrease IT infrastructure operational costs in more than 1.5 times.[Shr10] It is an evolving and relatively new concept; therefore there exist many

definitions of it. The term will be referred throughout this thesis. Hence it is important to define it and related technologies explicitly. In their work, Dillon et al[DWC10] relate on the key expectations from that field and refer to the denotation provided by U.S. NIST (National Institute of Standards and Technology):

> *"Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*

According to[ISO14]main properties of cloud computing are:

- *Broad network access*: this feature of cloud computing is addressed to geographical availability of the service from any point with network connectivity, also including support of various devices, e.g. workstation, tablet or cell phone.
- *Measured service:* guarantees metering, billing and charging of service user. Important is the added value for the customer - lower operational costs - payment is performed only for the resources that were used.
- *Multi-tenancy:* the share of computing resources between several customers. Tenant's are not aware of each other, their data is safely stored and can not be accessed by other customers.
- *On-demand self-service:* if the client assigned to the service, there are no other necessary steps preventing from its usage. Service are used only when they are needed. However, they are constantly available.
- *Rapid elasticity and scalability*: resources must be scaled up according to the current workload or forecast information. Statistics gathered during the usage of the application shows the pick hours when the workload is maximal and triggers the allocation of additional resources. The system is also taking the workload drop into account and the resources are given back to the pool.
- *Resource pooling*: serving multiple customers on a shared hardware that is 'hidden' and the only visible interface is a pool of virtual machines, application components or designated cloud service itself.

An important characteristic of the cloud is its *deployment model.* It could be defined by user groups that could access a cloud and level of tenant separation. In the following paragraphs, a brief explanation of cloud service models is provided.

A public cloud is available for both enterprise customers and individual users by means of multi-tenancy. As a result of user diversity and possible high number of customers, cloud provider is able to handle peak workloads of individual customers thanks to the fact that high workloads at one tenant are not necessary correlating with the other one. In this case economy

of scale is easy to leverage, every customer sees public cloud as a black box with unlimited resources, however resources are limited and spreaded on demand [FLR+14] .

Private cloud is limiting the allocation of resources to one enterprise environment and tries to ensure maximal isolation from the public Internet. Allocation of the cloud could be done either on company's data center or delegated to the outsourced cloud provider. Often public cloud providers offer so-called *virtual private clouds,* which may guarantee a higher degree of isolation from other service consumers in compare to the public cloud, e.g. each tenant could use only designated hardware. In this case connection to the cloud is established by means of a secured tunnel, e.g. VPN [FLR+14].

A community cloud is a cooperation initiative supported by different companies and organizations with a mutual contribution to its infrastructure. It takes the place when companies share their data to achieve synergy, public cloud may be not appropriate according to security issues, private cloud is not meant for multi-tenant access.

A hybrid cloud deployment model represents a mixture of above-mentioned models optionally including the internal non-cloud infrastructure of the enterprise. As a motivation to use such a cloud following aspects have to be considered: elasticity, accessibility, and trust. To ensure elasticity property it would be economically reasonable to store cloud components according to their workload on separate cloud environment, e.g. components with predictable workload could be stored in internal environment while components with unpredictable or periodically changing workload could be stored in a more scalable environment. Access to applications is needed by different groups of users, some of them are employees connecting within intranet, whereas others are business partners interacting over the public network. Trust aspect is related to the willingness of cloud customer to store its highly valuable or confidential data on provider side [FLR+14].

Cloud Computing is a very wide topic that demands an analysis from different angles. Above introduced properties play a role of a foundation for service models and deployment models that sets up a list of requirements for cloud providers. Service model touch on functional part of a cloud service, answering the question, what exactly is being delivered. However deployment model is considering a question of security and trust in a cloud, answering the question, where does customer wants to store the data?

Portability and interoperability of cloud services

Portability and interoperability show up among other aspects of cloud computing, as well as correlating with the context of this work. Portability relates to a possibility of reuse of cloud service components, meaning minimization of costs when migrating services between clouds or creating a heterogeneous service compound of relationships to more than one service provider. Interoperability relates to the ability of different cloud service components to communicate in a way with either a little or no knowledge about internal characteristics

of their components [ABC13]. The possibility of non-cloud applications being involved in the picture also has to be considered. For instance, enterprises are required to keep personal data in an internal environment. Thus, cloud services have to be coupled with already existing legacy systems. Both terms closely relate to the hybrid cloud deployment model, which requires loose coupling between components and their alignment in fine granular groups [FLR+14]. Hence, portability and interoperability involve private, public cloud services and non-cloud applications binding together both roles: customer and service provider. Interaction between them can be carried through diverse abstraction levels, perfectly fitting into the frame of service models. Depending on the interaction layer between customer and service provider, following entities are considered: Data, Application, Platform and Infrastructure. Discussion of portability and operability perspective of these entities carried through the following paragraphs.

Interoperability of data is done via interfaces of application component rather then directly[opea]. These interfaces are based on unified OSI model standards, i.e. a stack of protocols that provide among others virtual networking. Apart from other standards it is necessary to emphasise VLAN and VPN. VLAN enable multi-tenancy and guarantee that data can not be transferred to other customer's network. VPN ensures interoperation between the corporate network and public cloud provider or between different clouds. In order to provide message exchange between applications *message-oriented middleware* is being used [FLR+14].It encapsulates complexity and heterogeneity of cloud services, addressing the problem of broadcast and multicast communication. Among possible implementation patterns are *message queue* and *pipes-and-filters* architecture.

Portability of data between applications has found successful implementation by means of widely acceptable markup languages and object notations, e.g. XML [Lea99]. Among others are JSON, YAML, traditional SQL schemas, etc. Some languages are utilized to represent data while communicating between different tiers of one application or across different applications. Others, like SQL schemas and JSON, are also used to store the data in *relat*ional and *non-relational* databases accordingly.

Application interoperability could be realized via Service Oriented Architecture (SOA), particularly in Web Services concept, that was discussed from a business processes perspective in the work of Leymann et. al. [LRS02]. They solve the problem of flexibility in choosing a business partner offering a service. The main idea is if the service interfaces are unified and information about them is stored in a centralized place then they can be automatically discoverable by each other. As a mechanism to communicate with web services SOAP standard is used, which defines semantics to exchange messages between services. As an alternative REST architecture could be used. It is not protocol specific, however, is mostly used over HTTP, thus providing high application interoperability [Bur10]. One could also define predeceasing technologies like CORBA and Java RMI that are used less due to vendor-specific dependencies.

Application portability. Cloud applications differed from traditional monolithic one primarily by its distributed nature and increased complexity resulting from its properties. The first steps in cross-platform portability were done by Java, Python, and Ruby, where program code

is executed or interpreted in OS add-on, e.g. Java Virtual Machine (JVM). Reusability of relatively simple applications could not be that tedious task, e.g. deliver JAR files to another JVM. However looking into the future, applications are getting more complex with growing number of interdependencies to other platforms[2]. If an enterprise building an application in PaaS provider decides for whatever reason to switch to another provider, it is not guaranteed that the application will be easily adapted. Another possible scenario of application portability would be application migration from development environment to the operation environment. It is not always economically reasonable to build development environment, which will not be used after application release. It is vital that application could be transferred to the operation environment without changes. Thus, companies utilize services of one PaaS provider for both development and operations purposes (*devops*). Cloud applications are also including programs addressed to deployment, configuration, provisioning and orchestration of cloud resources. It is important for SaaS provider to be flexible in a choice of a PaaS or IaaS provider. For that purpose connection to different cloud providers interfaces must be unified, this property is named *management interoperability*. There are several solutions for automation of operations that fulfilling that property, e.g. Chef, Puppet [opea].

Interoperability of platforms is achieved by protocols for information exchange with standardized or vendor defined interfaces. The platform includes both operating systems and *middleware*, e.g. application servers, database management systems [FLR+14]. Among standard interfaces CDMI could be distinguished, which is specifying how to access cloud data storage and how it must be governed [CDM14]. That is allowing access to diverse storage platforms via a standard interface. From the other hand communication between different platform architectures is often done by *point-to-point interfaces*, for that purpose development of a converter between every two platforms is required [ABC13]. For instance interoperability between following platform pairs could be a problem without an interface specification: JVM and Python, .NET and JVM. Interfaces allow execution of Java code from Python and C# code from Java.

Portability of platforms is done by means of virtual images, which could be copied across private users or cloud service providers. Traditionally portability of platforms is realized via copying the whole virtual machine (VM) with its guest OS, which is not always necessary, e.g. only one middleware component is needed to be copied to the other environment, not the whole VM. However more fine-granular way of portability is conveyed through Linux Containers (LXC)[CS14]. LXC provides operating system level virtualization that creates application sandboxes isolated from each other. This technology was adopted in a software product called Docker, it eases deployment and portability of platforms and their components. Fine granularity in platform images contributes to the overall portability of cloud services, meaning that the transfer of cloud services could be done not only on IaaS, but on PaaS level. However according to the LXC nature of Docker it is not possible to execute containers in Windows

[2]According to Financial Times https://ibm.biz/BdE9FT

environment. It has high adoption rate with positive dynamics among web service providers, i.e. on January 2014 33% of websites were using Windows web servers [Net14]. Hence, it is still leaving shortcomings on OS portability.

Portability and interoperability of infrastructure entities are related to virtualization techniques and hardware components, which utilize physical interfaces applied in traditional computing. Hence, they are less relevant to the context of this work and will not be further discussed here.

Based on the portability mentioned above and interoperability aspects of different layers of cloud computing, one can conclude that overall portability and interoperability of cloud service descriptions depends on components of all specified layers. Nowadays portability of cloud service descriptions can not be fully guaranteed due to several challenges.

- Different application standards across cloud providers. Cloud service defined in one environment can not be migrated to another provider without significant refactoring. Nevertheless, individual steps in this directions are already done by *TOSCA*. However, it is a relatively new specification that is not yet a standard that can be adopted by major cloud providers.
- The separation between layers of a cloud service. In order to describe a service on the infrastructure layer, OpenStack Heat template could be utilized. It provides compatibility with AWS Cloud Foundation template format, but it experiences the lack of support of deployment scripts above VM provisioning layer. If it is required to configure and deploy software, Chef recipes could be used to perform it on a centralized basis. However, infrastructure layer is out of its responsibilities. Docker containers cause the same problem by defining only application components. Hence, there is a need for a standard that connects all layers of a cloud service and allows transparency in establishing of cloud services.

Enterprise Content Management

The amount of information produced, stored and consumed by companies, is growing year by year. This information is closely related to the content, organization is dealing with. It can include text documents, emails, spreadsheets and other digital assets. There are several driving forces motivating to manage this content. Some of them are a competitive advantage by means of valuable knowledge mined from unstructured content, compliance with law and regulations, provisioning of evidence in a lawsuit managed by eDiscovery. Knowledge are gathered from the refinement of enterprise content to the business valuable information [MNS14]. Enterprise knowledge could be split on know-how characterized by learning-by-going principle, know-what characterized by learning-by-studying principle and know-why related to learning-by-using [Gar97]. Afterward, this knowledge is used by the interested user groups in the enterprise. Another encouragement to pay attention for the content, is the

requirements put by the legislation. In some countries companies are obliged to store their *business records* for a certain retention period, e.g. according to Sarbanes-Oxley Act, electronic records of public companies have to be saved for the period of minimum 5 years [EHW04].

Above mentioned use cases are governed by Enterprise Content Management (ECM) domain. Definition of ECM field varies depending on the chosen perspective, e.g. content, function, technology, enterprise [GHH+12]. *Content perspective* is concerned with the semantics of the content and its relationship to the user. From the *functional perspective*, ECM is seen as a list of functional requirements to the system, while *technology perspective* regarded as an exact combination of software products used by an individual provider. *Enterprise perspective* shows the business requirements or problems that will be solved by these components. Taking into consideration above mentioned perspectives, ECM can be considered as a lifecycle of information from its creation and capturing to retention and deletion. In order to be precise the notation provided in the review of ECM research is used [GHH+12]:

„Enterprise Content Management comprises the strategies, processes, methods, systems, and technologies that are necessary for capturing, creating, managing, using, publishing, storing, preserving, and disposing content within and between organizations.„

Further content of this subsection is meant to clarify above mentioned perspectives on ECM. It is structured as following: general functional perspective follows content perspective. Enterprise and technology perspectives are tightly coupled to the particular customer domain. Thus, they are omitted here.

There are several forms of content that are produced and managed by enterprises: documents, records, email, web content, digital assets [AII13b].

Document management. Documents can be paper and electronic based, here only electronic forms are considered. Document management deals with retrieving, tracking, storing and controlling of the documents [AII13a]. It covers subtasks of ECM and comes with the following key properties [AII13a]:

- Thread-safe operations on documents, with distributed locking and consistent editing. Changes made by one user should not override simultaneous changes committed by the other one.
- Version control. The whole history of changes on the document is saved with the possibility to roll-back to a previous version.
- Non-repudiation during the audit. Ensure mapping between actions on the document and users, e.g. see who did what.

This list gives only an overview of possible properties since there are many more which are document specific. In order to illustrate it, let us consider workflow property of a contract in an enterprise. Big companies usually operate with significant number of contracts. Every contract includes vast number of intermediate steps from draft to completion. Typical lifecycle of a

contract might be: draft, review, revision, sign, approval, completion and possible termination [CNMK07]. In order to successfully manage such a document its current state might be considered, and also possible interoperation with external systems must be provided. As one can see every type of a document might have its properties, thus document management systems are usually targeted on a certain document category [NCK+09].

Web Content management. Nowadays enterprises often share their information and communicate with business partners by means of Web. Web Content usually includes HTML documents, images and videos. The primary goal of Web Content Management is to facilitate non-technical users to process the lifecycle of the content, e.g. provide basic CRUD operations [LY10]. In other words, it allows maintenance of presentation layer of *three-tier architecture* application.

Email management. Emails encompass the bulk of the communication within an enterprise, what makes them valuable for legal compliance and internal analytics. As a matter of fact, email systems is a source of history of decisions of the company, which makes them an object of eDiscovery. From the other hand, they could be used to mine data for knowledge-discovery [DHGS13].

Records management. All above mentioned content types could be included into record, which is a type of content related to some state and serves as a prove of executed activity. Usually record corresponds to the following [ZAB+09]:

- Its content can not be changed.
- Plays role of evidence of action or transaction brings essential value for business or legal compliance.
- Identified by retention policies, governing questions what information is to be stored and when it will be deleted.

Hallmark of record from other types of content, for example, document, that it cannot be modified. If the document is modified several times, it will correspond to different records. Records management involves: capturing of the information that was identified as valuable, categorising of records, keep records for a certain period of time related to retention policy, destroy records when they are no longer needed, guarantee consistency of the record during audit trail.

2 Solution concept

The declarative nature of service description provided by enterprise customers has a variety of possible interpretations and does not allow to conclude directly how it will influence the resulting service runtime. Thus, it is required to provide clarifications and assumptions addressing possible input to the system. Once the system input is specified, the scope of the influence on implementation components is to be elaborated. In order to avoid misunderstanding in used terminology, it is necessary to state that both words $user$ and $customer$ are used as synonyms. They define an abstract enterprise customer who experiences the need in the interpretation of a service description. Specification of exact actors within enterprises is not appropriate due to the difference in their hierarchy. Thus, the generalized notation is utilized.

Several possibilities of interpretation of user input are considered: direct mapping, model transformation and component map reduction. In the following chapters selection of one these concepts is made and justified. It is followed by the discussion of elaborated details of chosen approach.

2.1 User input considerations

In the following two forms of observed user input are considered:

- *Ambiguous form.* In the general case, a user describes the system that is needed in the term of its problem domain. In the most basic case they could be expressed in the form of natural language. The problem definition reflects functional requirements to the system. For example, the statement "I need to store my data in the web archive" could be interpreted as requirements on a repository, database and web user interface from solution domain. However, the meaning of this sentence may vary from customer to customer, at the same time there could be synonyms interpreted differently depending on the field. Priority of service delivery can be conveyed in the sentence "I need the solution with lowest price", this could trigger the search for open source components with the appropriate license. Both statements relate to functional and non-functional requirements to the system. However, the number of system components to be deployed and the final topology remain unclear.
- *Precise form.* The user is aware not only of the problems but also of the service entities that possibly solve them, as well as the workload that its company produces. Hence, one can

specify abstract components with their dependencies and quality of service requirements. The common vocabulary in the form of a *Domain Specific Language* (DSL) is used for the purpose of precise identification of customer requirements. Its *semantics* defines the mapping to components of the system; while its *syntax* determines the service topology and its interpretation sequence. Language semantics covers functional and non-functional requirements to the system. Functional requirements can be presented in the form of a graph where nodes are abstract components of the system and edges are relationships between them. Each such a node could be mapped to more than one software component from implementation domain while edges reflect orchestration sequence of components by simple "depends_on" relationship. Non-functional requirements specify the quality of service or capabilities the system should obtain, e.g., average number of transactions per day.

In both cases, the cloud provider needs to determine the exact service components considering the required quality of service. Mapping to the correct components is concluded from the semantics of user input. In order to specify this mapping, one needs to consider implementation components of the system and how they can be influenced. First of all I split the components of a generic cloud service into three layers: Infrastructure, Platform, and Solution. Infrastructure layer includes components defined by IaaS provider, which encompasses virtual machines, network topology, load balancers. Platform level includes software components that are common to several services that could be delivered based on the user input. Solution layer contains software components that differentiate services from each other. Entities on each layer typically have configuration options that can be customized according to customer requirements. The higher the number of configurable components, the more flexibility in provided services exists. Thus in this work configuration of components of all layers is to be considered and user influence on the system is restricted to:

- *Inclusion of components into the system.* It is found that user input includes functional specification of abstract system components. They are mapped to the exact components based on interpretation logic.
- *Fine grain tuning of the resulting service.* The quality of service parameters defined by the user is translated into component parameters.

Following restrictions are identified regarding the relationship between user input and service implementation: (1) functional requirements could be mapped to implementation components only, (2) non-functional requirements are mapped to either implementation components or their individual parameters. This separation of mapped service elements addresses the issue of the definition of processing sequence, e.g., after selecting the components fulfilling the required functionality it is possible to proceed with their configuration. Selection of system components is followed by their configuration and deployment of a final service. Details of this process are described in the following subsections.

In the rest of this work, it is considered that user input has the form of domain specific language with known vocabulary, syntax and semantics.

Customer expectations

Enterprise customers do not always need a service with 100% matching of specified requirements since there is a degree of uncertainty in the necessity of each component. Furthermore, their requirements usually change over time [Ber14]. At the same time, it is not always possible to totally eliminate redundant requirements or to persuade the customers continuously to provide exact requirements. However, it is possible to differentiate them based on their necessity and finally let the user decide whether the service will be deployed or not.

It is proposed to allow the user to define requirements based on two categories: "strict" and "soft". The first type represents critical system properties that must be definitely provided, the second one defines desired properties that can be omitted in service deployment. "Soft" requirements could be either not specified at all or missed during the selection of final service template. In the first case, default system values are chosen, while in the second case the most similar service is selected. Furthermore, when user input is analyzed and compared to the available components the result of comparison should be displayed for the user that could make the final decision if such a service is acceptable or not. Another possibility would be to let the user define acceptance rate, which determines the percentage of soft requirements that may be missed.

Exact matching of user needs would require the generation of service templates from scratch for each user input, e.g., attempt to make individual solutions. However, this option might not find support of service providers based on the following reasons:

- Service providers typically have a finite number of offered services. If they are known in advance, then it is obvious that their configuration could be also predefined.
- Reliability of resulting service. Each generated service must be adequately tested and debugged. After that quality assurance team has to ensure stability of each solution. In the context of limited resources, this might be unrealistic to perform for all possible combination of components and their parameters.
- Deployment test. The requirement in a centralized orchestration system which includes individual properties of each component, not only their compatibility. Following requires the prior generation of each possible combination of service templates in order to prove that the final solution is deployed correctly.

Aforementioned user expectations do not always need to match the delivered service ideally. At the same time list of predefined service templates could be generated due to finite number of offered services. It is also not always economically reasonable to generate and test each possible combination of components. Based on the facts as mentioned earlier, it is decided to have a

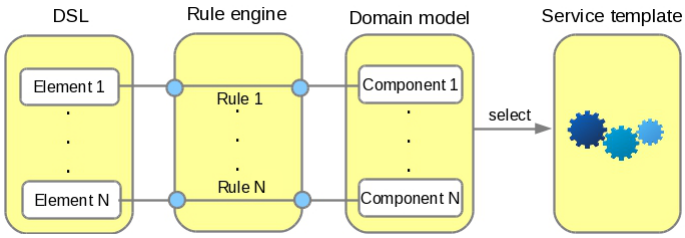

Figure 2.1: One to one mapping concept

list of already predefined templates with configurable parameters. Each template contains bundles of components based on best practices of service delivery and considers the majority of customer expectations. Hence, the service is delivered according to the following sequence. Parsed user input is associated with exact components. Their combination is compared with each predefined service template. The one with maximal similarity is selected. After performing of fine-grain tuning, the template is sent to deployment.

2.2 Concept selection

Based on the above-mentioned considerations regarding possible DSL input, three promising solution concepts are identified. Their aim is to answer the question of how to interpret customer requirements to the system into implementation components.

- *One to one mapping* between the elements of DSL and application components. The significant point here is the assumption that interpretation of the DSL is realized based on the decision table [SP10], where every DSL component relates to one single element of a domain model. After element's decision was made, service template is to be generated and deployed in a cloud provider environment.

  The concept is schematically depicted on the Figure 2.1. As soon as the customer defines a service based on DSL, he uploads it into the interpreter. It parses the input, and the Rule engine defines a strict mapping between each input element and a component of the domain model. On the next step service template is selected and submitted to the cloud provider. The provider instantiates the deployment and sends back the access data with the status of deployed application to the customer.

  The approach would work fine for the case of static components defined in each service. However, when it comes to the situation of non-trivial mapping,e.g., ambiguous one to many relationship, the choice of the exact components has to follow complex rules considering non-functional requirements which is not addressed in this approach.

- *The transformation of models.* The approach is depicted in the Figure 2.2; it was utilized and discussed in the work of Chang et al. [CLY+14]. Here initial data model is going

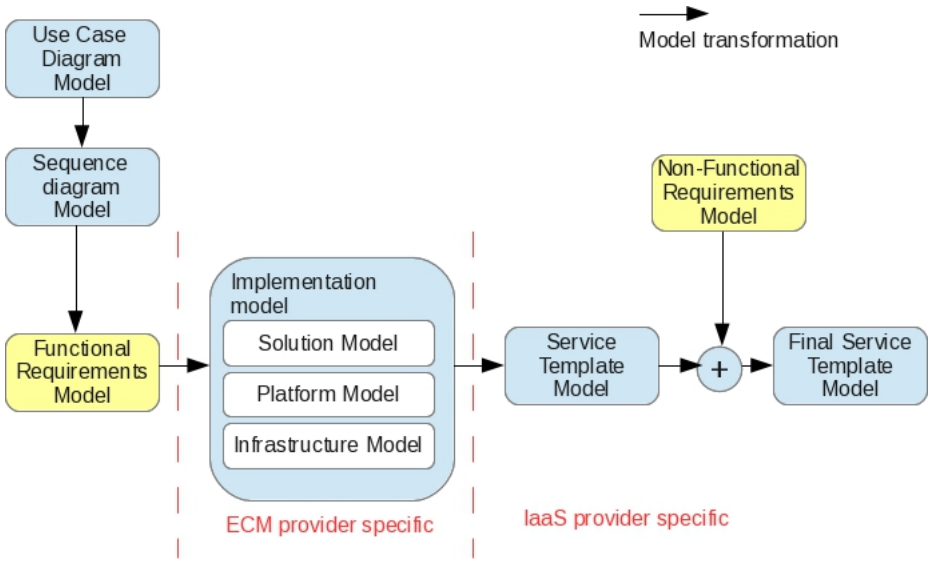

Figure 2.2: Transformation of Models

through series of transformations $(arrows)$, where each step represents a new model $(rectangles)$. Every model is associated with its schema, or metamodel, which describes the elements and their relationship. One can distinguish two types of transformation: between models belonging to the different schema and between models within the same schema. Transforming into different schema is considered as a *standard transformation* process. Transformation to a new model is done by a set of rules which identifies the relationship between elements of both models, typical finding one-to-one mapped components. Interpretation in the scope of one schema is noted as *sibling transformation*; it serves for refining purpose. This transformation can only change the already defined parameters, or substitute its elements constituting to its metamodel. Mapping is defined as *Sum* as depicted on the Figure 2.2.

The key characteristic of this approach is the separation of functional and non-functional requirements to the system. The initial model is represented by use case diagrams from different stakeholders with similar use cases related to the same functional components. The first transformation includes the generation of sequence diagram with eliminated duplicates. During the next step, each element of sequence model is to be transformed into a functional requirement. The next transition interprets requirements as implementation components. Here every requirement is mapped to one of the components of either solution, platform or infrastructure model. Combination implementation elements are transformed into a service template. The final step is to include non-functional requirements to the system. This approach provides a powerful to tool to transform user requirements into implementation components. However, it would require significant

number of transformation steps as well as created metamodels and rules in the case of high difference in the abstraction levels between the models.

- *Domain model reduction.* This approach combines ideas of the first two methods; it is schematically represented in the Figure 2.3. Certain similarities are followed from feature space reduction in content classification problem introduced in [Alp10]. While classifying sets of data one is given with series of random observations and a list of categories, they could belong to. In this work, observations are represented by the DSL input and categories are groups that contain service components. However, in the case of the declarative language input with strict vocabulary and finite number of combinations, one does not obtain random observations. Hence, only a deterministic approach, where the same user input always generates the same output, can be applied. At the same time, the concept of gradual reduction of component space fits well to the problem. It allows to decrease the number of service components under consideration.

  The DSL input is to be split into two parts, functional and non-functional requirements to the system. Instead of transforming into different models. Only the one model is utilized; it is defined by the map of components. Initially it contains the whole range of components, after interpretation of each part of user requirements it shrinks until it contains only precise elements of the service. At the first step, a rule engine identifies the categories of components according to processed functional requirements. After that non-functional requirements related to implementation components further reduce the number of elements in the model. Finally, property-oriented non-functional requirements influence the model. Based on the observed entities, final components obtain their configuration parameters or use the default ones. Determined components are looked up in the list of predefined service templates. If the matching template exists, it is configured based on the above-mentioned configuration parameters. The result of this step is represented by the template ready for deployment.

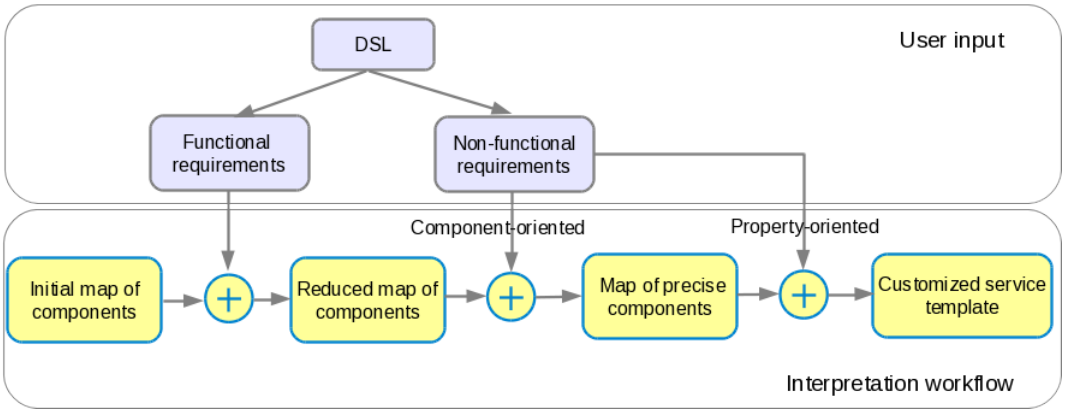

Figure 2.3: Concept of domain model reduction

This method reduces shortcomings of the first two approaches: flexibility on mapping to several possible components after processing of functional requirements, only one model that stores all the components of the system reduces the complexity of interpretation. The concept of gradual reduction of components of the model is chosen as a basis of this work.

2.3 Interpretation workflow

Once knowing input types and their relationship to the implementation entities, one can define the process of interpretation of a declarative service description. In the following this process is introduced, in the next subsections details of the process are discussed.

Initially, a map of all implementation entities is available. It includes service components with their possible configurations. The overall aim is to reduce this map to the list of components that suits to the service requirements defined in the DSL input. Service components are classified according to their functional and non-functional characteristics. The first step is to select only those that match to provided functional requirements. This process is hardly dependent on the syntax of user input. The difference in abstraction levels makes an impact on the further steps of interpretation. From one hand side, the syntax can have fine-grain detailization and its vocabulary is strongly related to entities of implementation domain. In this case direct one-to-one mapping could be defined. For example, DSL represents repository component, which could be mapped to one of the available object stores. However, deployment of only the one component does not result in fully functional service, to solve this issue dependencies to other entities need to be taken into account. From the other side import by means of coarse grain DSL could be provided, in this case, direct mapping to implementation entities seems to be unrealistic. If DSL entity is too abstract, then various interpretation options could match to the user input, which requires an additional step to associate atomic implementation components to the input. Considering the worse case with coarse grain components of DSL, initial analysis of input values associates observed functional requirements with various component categories. Thus, after initial extraction and analyzing of functional requirements, matching components reduce the original map. Now it includes only the entities reflecting functional requirements of the service. It stands to mention that the map contains redundant elements, for example a need in a database could lead to several software components that provide such a capability. Hence, the next step of interpretation further reduces the map by th selection of only those entities that are being deployed. Here non-functional requirements related to implementation components come into play. Each of them highlights a component required in the final service. An example of such a non-functional requirement provided by the user may include the definition of the vendor. In this case, only components from the certain vendor will be selected. In order to reduce the map to the condition with no ambiguous components, the user may be obliged to provide one of the priority dimensions. Otherwise, a default value is set. In a short priority dimension is a generic non-functional requirement to the system, such can be cost, vendor,

availability. Internally each system component has a metric associated with each priority, in the case of ambiguity in the selection process, components are chosen randomly or according to default priority.

On the next steps, the map of implementation components with their default parameters becomes available for further processing. Only non-functional requirements related to fine grain tuning of the system are not reflected in the components yet. They are processed by the *Rule Engine* that maps them to the exact component's characteristics. At this stage, one has to consider that the components are still required to be deployed. A naive approach would be to process left virgin DSL input and change parameters of implementation components. Afterward, to find a service template and deploy it. However, this method has a significant drawback. The probability to match to one of already existing service template drops significantly if configuration parameters are considered in the comparison. Hence, it is more rational to match service template without consideration of its parameters first, and then configure it and finally initialize the deployment process. This approach leaves space for customization of service templates and their reuse for several customers. Originally there is a list of already predefined service templates in the format compatible with one the cloud environments. They reflect highly-demanded services only. Each of them is configured based on default parameters and exposes an interface to change them on-demand. The last step of the interpretation is to initialize deployment of the template. It is realized by sending it to one of the available cloud environments with parameters based on non-functional requirements targeted to the configuration. Additionally service access details must be provided to the customer.

Determination of component categories

As mentioned earlier specification of implementation components is a sequential process whose first step is to analyze functional requirements and to determine a $category$ of potential implementation components. This process is considered in the context of the whole interpretation sequence as illustrated in the Figure 2.5. The following subsection describes the details of category identification.

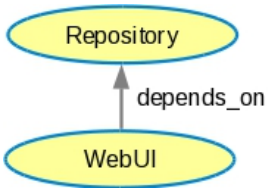

Figure 2.4: Graph of functional requirements provided by DSL

First of all user input is divided into two groups: functional and non-functional requirements. Initial analysis is done based on functional requirements only. They are internally represented

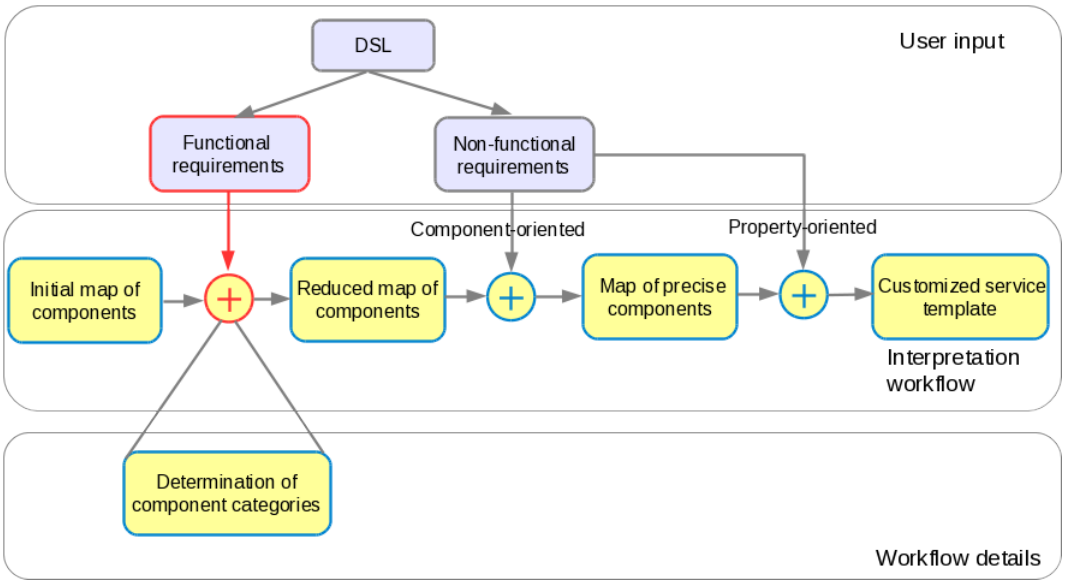

Figure 2.5: Processing of functional requirements

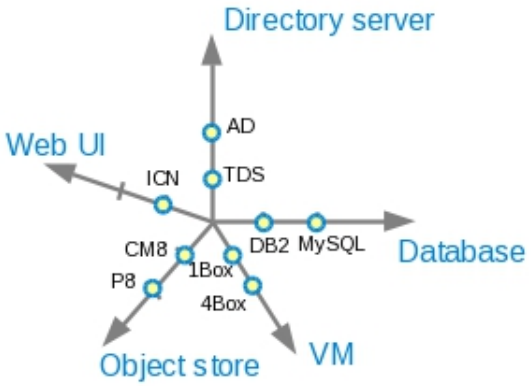

Figure 2.6: Component space example

in the form of a graph[1]. This notation is illustrated in the Figure 2.4 and defines a system that consists of the repository with the web user interface. The key interpretation element that maps implementation components to each functional requirement is the *Rule Engine*. Each graph node is checked against matching to individual rules. Each rule determines the semantics of functional requirements of DSL input. In other words, it maps user input to component categories including atomic implementation components. The property of atomicity defines that component can not be further cracked into smaller entities and is unambiguously mapped to one of the points in a component's space. This space is represented by n form-

[1]graph representation is discussed in the section 2.1

vectors, where n is the number of component categories of the system. An example of such a space is depicted on the Figure 2.6. Each form vector is defined by discrete values that are represented by components of one of the implementation layers. In object oriented notation, each dimension could be considered as an interface, and the values on its axis are defined by classes implementing this interface. For example, database dimension may include various implementations differentiating by its vendors or other unique properties.

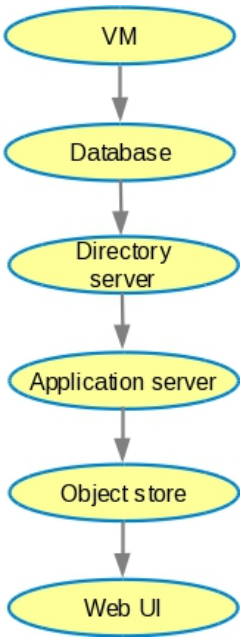

Figure 2.7: Graph of rough components after rule engine processing

Identified atomic components with their relationship substitute the nodes of the original graph of functional requirements. It results in a new graph of service components that is depicted in the Figure 2.7. It represents deployment workflow of components. Each node relates to one dimension in the component space. This association identifies one to many relationship between a node and implementation components. Each outgoing edge represents the next component that is being deployed. Hence, all nodes of the graph are mapped to the component space. For the purpose of expressiveness two representations of functional requirements are necessary: vector and graph representation. The first one is showing that reduced map contains components with several possible implementations. The second one stores the information about deployment workflow; it is used in one of the further steps for determination of compatibility between components.

On this stage processing of functional requirements results to the map of possible service components.

Reduction of component space

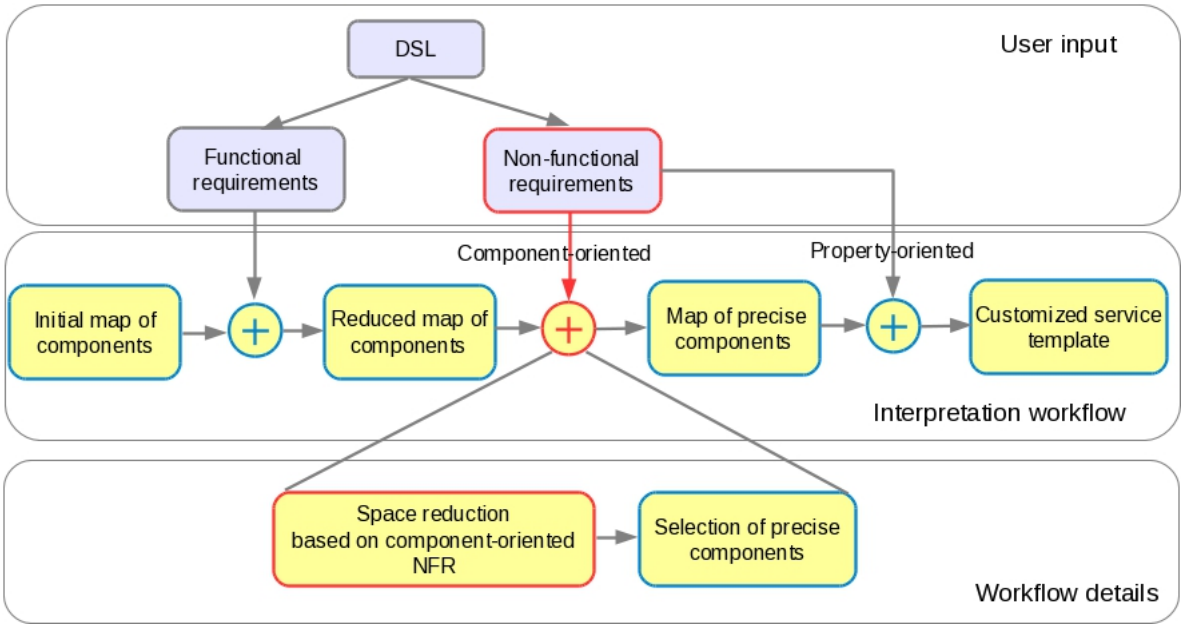

Figure 2.8: Processing of component-oriented non-functional requirements

On this step, *non-functional requirements* defined in user input are being processed as depicted in the Figure 2.8. The following example demonstrates the difference between them. One has to consider two non-functional requirements "Vendor of components" and "Number of transactions per day". The first one relates to components of the system while the second one in the common case determines their parameters, e.g., the number of CPUs of virtual machines. In the following only non-functional requirements related to components are discussed.

The requirements further reduce component space. Observed user input is mapped to an individual component belonging to one of the directions. This eliminates ambiguity at least on one dimension. Hence, all the other components from that directions that were not mapped can be excluded from the space.This step is necessary to eliminate unnecessary operations on the components that are not being used in the final service.

Selection of precise components

Once a component space is determined, further processing needs to be done in order to identify a unique vector from this space that will be associated with exact components of the system.

In order to select exact components, the graph abstraction is selected. The decision is motivated by the fact that final service represents a unique combination of components deployed in a certain sequence. Each deployment step is designated to a component category with several

possible components. Their non-functional characteristics could be mapped to countable values and used as metrics for transitions between vertexes. Finally, it is required to find the optimal path in the graph based on the selected metric. The Vertexes are represented by possible service components and directed edges are defined by the deployment sequence. One only needs to find all paths from initial to the latest component of deployment workflow. Selection of one of them highlights the required service components. In the following content of this subsection I introduce the concept of graph creation and path selection with possible optimization.

Component	A	B	C
A	-	1	0
B	1	-	1
C	0	1	-

Figure 2.9: An example of compatibility matrix

Creation of a new graph involves substitution of vertexes and assignment of new edges between them. Injection of new vertexes is realized by joining of graph of rough components with component space. Here each node is to be substituted by new nodes corresponding to values from space dimensions. However, old edges from the predecessor graph do not reflect compatibility between individual components from each dimension. It might be the case that components from different vendors are not compatible with each other, or even the difference in the versions of the same software component within one vendor does not lead to their interoperability. Since user does not know internal representation of the system and can not define it in the form of requirements, it must be performed internally in the system. For this purpose edges between nodes are created based on their compatibility. To fulfill these needs the concept of compatibility matrix was elaborated, it is depicted in the Figure 2.9. Its columns and rows are represented by components of the system, their intersections contain the value showing if the components pair is compatible or not. Three possible values are chosen: "1" - compatible, "0" - not compatible, "-" - there is no relation between the components including the case of loop back to itself. At the end the graph contains interconnected components, where relationships reflect the deployment sequence and consider their compatibility. The resulting graph is depicted in the Figure 2.10.

In a simple case when transition between nodes is not weighted, random selection of the path could be utilized. This is appropriate when user did not specify a service priority. However, more optimal topology can be selected in the case of weighted transitions. The transition between each pair of nodes has a weight selected from one of the predefined priorities: "cost", "availability". Hence, service topology is optimized in one of predefined dimension. For example, a topology with cost optimization contains components from the shortest path between start and end vertexes of the graph.

In order to be able to define weights of transitions, the concept of priority space is introduced. This space represents a system of coordinates where every implementation component is allocated to a unique point, an example of such a space is depicted in the Figure 2.11. In the example I consider a three-dimensional system of coordinates with "Scalability", "Availability" and "Price" axises. Two implementation components of database set "MySQL" and "DB2" obtain their unique points according to the values in each dimension. Depending on the implementation, both discrete and continuous values could be used. Needless to say that performance indicators for each component along each dimension have to be identified and statically defined in order to place components into such a space. Points in priority space could be also explained from the object-oriented perspective. Each implementation component is represented by an instance of the class. Hence its metrics related to each dimension is stored in attributes of the class instance. In order to lighten future retrieval, all components are sorted according to each dimension. By knowing the weight of each component, one can calculate the path from one of the first nodes to the latest nodes of the graph. The exact path is chosen based on one of the metrics.

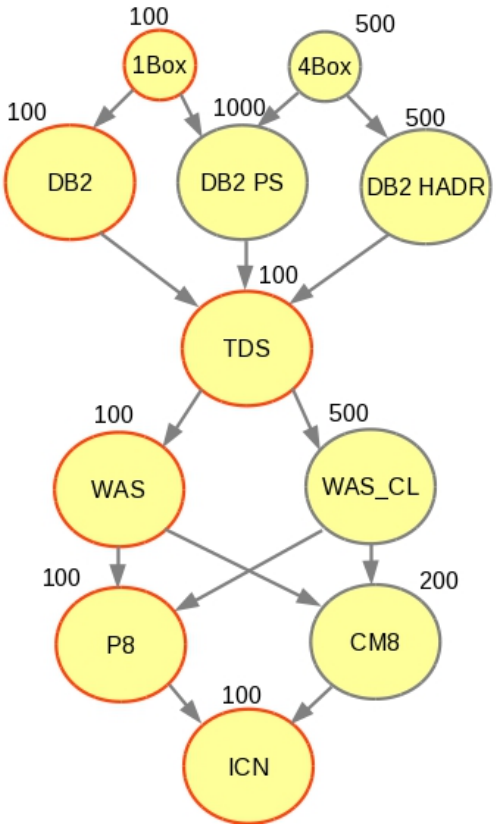

Figure 2.10: The graph of precise components

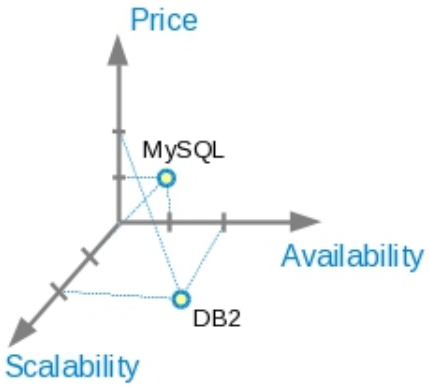

Figure 2.11: Priority space example

The following orthogonal cases of user input defining priority dimensions have to be treated differently: (1) user did not specify them at all, (2) only the one dimension is present, (3) several contradicting dimensions were chosen. If the user does not specify the dimension itself, then default priority should be defined that will be utilized for topology optimization. In the second case, the user input precisely identifies the dimension. In the third case, no trivial selection is possible, thus either the user has to specify priorities of non-functional requirements he provides or it may be implemented in the system. Consider an example of the following DSL input by means of non-functional requirements related to the components: "vendor=IBM", "availability=99.99", "cost=low". This combination means that the user defines the system based on IBM components with high availability and the lowest cost. In this case, all three requirements may conflict with each other since the components with the lowest cost most probably do not guarantee high availability and might have an open source license that conflicts with the vendor requirement. Hence, non-functional requirements have to be treated according to their rank; the components matching the requirements with highest priority are chosen first, under otherwise equal conditions other non-functional requirements with lower priorities are considered.

Path selection

Calculation of paths between any two nodes could be solved by well-known algorithms. However, weights for each transition must be known. Priority dimension is to be defined prior to calculation. This condition guarantees the selection of appropriate metric for the transition. There are two options for path selection: each dimension has its rule, one rule for all dimensions with unified metrics. According to the first option, the rule determines the selection of either one of the extreme values of the paths, e.g., shortest,longest or the path with the exact value. For the second option, one may determine metrics of different priority in such a way that for each path selection a single action is to be matched. For example, "availability"

might be valued in descending order, the lower the number, the higher the availability of the component. Hence, in the case of only two priorities - "cost" and "availability", calculation of the shortest path is enough to determine the exact components.

The first option with rules for each dimension is more flexible and allows among others to define a service plan as a dimension. A typical cloud provider could have such a plans containing components differentiated by their QoS characteristics. Each plan is to be associated with a particular metric that is saved in the properties of each component. Hence, in order to choose the components matching the designated plan, one need to select the path with the weight associated with the plan.

Configuration of components

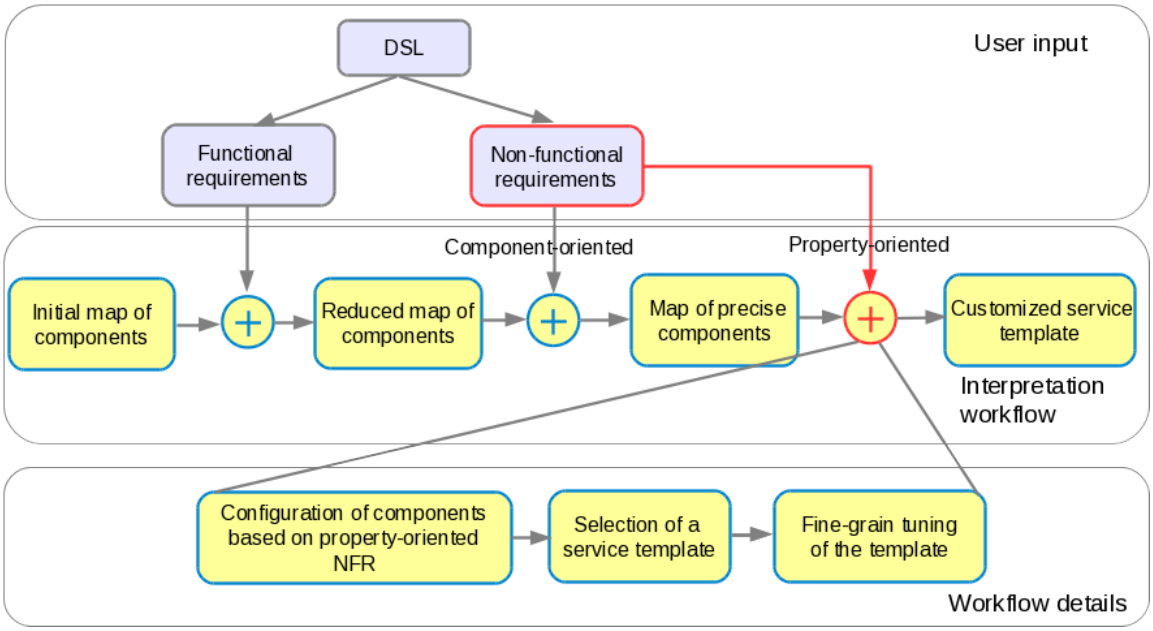

Figure 2.12: Processing of property-oriented non-functional requirements

After selection of exact service components, the non-functional requirements related to component properties are considered as depicted in the Figure 2.12. Finally, the second part of non-functional requirements related to properties of the components is being processed. The main purpose of this step is to translate properties of DSL nodes into the characteristics of system components. It is realized by the rule engine that maps properties of abstract DSL entities to configurable parameters of service components.

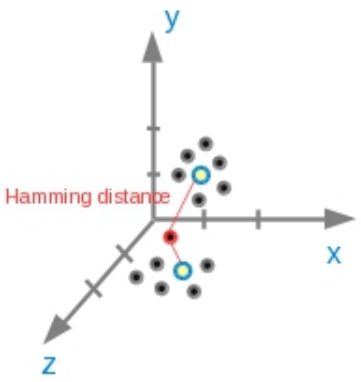

Figure 2.13: Component space with possible service configurations

Selection of a matching service template

On the previous step, properties were assigned to exact service components. However, until this moment they existed only in the form of objects without specification of their deployment scripts and consideration of cloud environment they will be executed in. Each component has the associated binary files, and its deployment scripts, the combination of selected components should be defined in the service template. Service templates could be either generated from scratch or belong to the list of predefined templates containing most demanded combinations of components. The first case is more flexible in a sense that a new template can be possibly generated for each user input. However, the orchestration sequence has to be known to the system. The second approach allows to configure the template by means of input parameters, however, without guarantee of 100% matching to input requirements.

Finally, service provider has a finite number of verified templates. Each of them includes software entities from component space. Hence, the template is associated with a unique point in this space as illustrated in the Figure 2.13. Black dots show possible component sets that could be observed. Yellow dots are service templates with defined components. Notable is the fact that the number of service templates might be significantly lower than the number of possible combinations of components. For this purpose similarity between the identified service and the existing templates is calculated. For these means Hamming distance could be used to identify the most similar template [GKK93]. The closest point in component space will be selected. Moreover, certain acceptability threshold has to be defined, e.g., maximal Hamming distance that is allowed for the template to be matched. Each cloud service template is to be given a unique identifier that depends on components presented in the template. At the same time, each of the selected service components is also provided an identifier. The service template with the maximal number of congruent components will be chosen for deployment.

Fine-grain tuning of the template

The only part of user input that was not discussed is non-functional requirements related to parameters of the implementation components. They are processed on the final step when a matching service template is already selected. Injection of parameters can be done either by modification of default template parameters or by passing them at the moment of deployment. In the first case modification of the template is done on the side of the interpreter. Default parameters of the software components are substituted based on non-functional requirements provided by the user. In the second case, service templates are already stored at the cloud provider. Initialization of one of the templates is accompanied by parameters passed from the interpreter. These parameters relate to configurable settings of the template, e.g., number of CPU cores, RAM, hard drive size, required platform as well as the configuration of software components. This functionality should be supported by the rule engine of the interpreter, it passes values to the REST client of cloud provided.

In both cases only template parameters could be changed, not the combination of the components. In contrast to instant modification of the service template, it takes the less computational effort to pass parameters at the moment of deployment. The interpreter makes an API call passing the parameters instead of parsing the template, finding default entries and replacing them. In this case, the values are replaced automatically by the cloud provider. According to the interpreter requirements it is assumed that ECM customer is not aware of the internal configuration of the service. Hence it is impossible to download the final implementation template. As long as this requirement is valid, it is not necessary to prepare the template on the interpreter side. Based on these assumptions, passing parameters to cloud provider API is chosen as fine-grain tuning of the service components.

2.4 Application architecture

In this chapter, the architecture of a generic interpreter is introduced, which provides automatic interpretation and deployment of service templates defined in a domain specific language.

From the high abstraction level perspective the application interacts by means of parsing a service template defined in the DSL. It is followed by matching of the service templates, its refinement and deployment of one of the predefined service templates. As can be seen from the Figure 2.14, interpreter consists of the following tiers: presentation, logic and data. Parts of the first two tiers fall into Model-View-Controller (MVC) design pattern [MJ99]. While data tier incorporates application data backend, and the cloud provider.

The presentation tier is represented by View in terms of MVC design pattern. It includes a web user interface that enables the user to create, retrieve, update, delete service templates in the database. The main goal of this level is to provide transparent user experience encapsulating interpretation logic, and service orchestration.

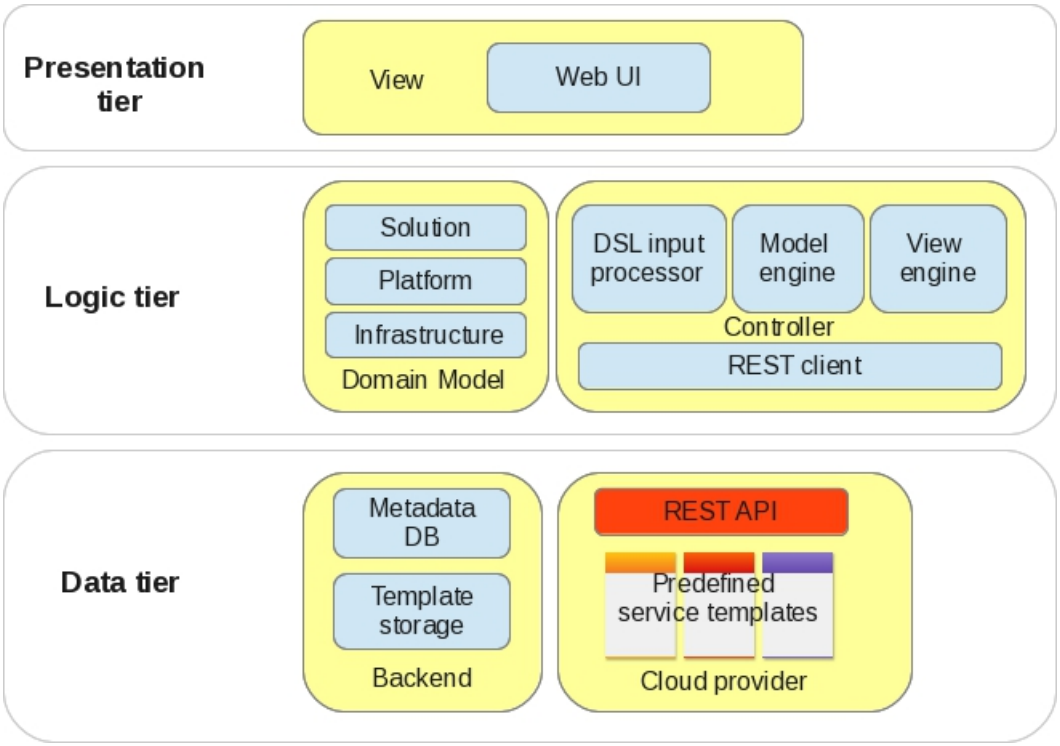

Figure 2.14: Architecture of the interpreter

Logic tier represents a core component of the application. It is made up of *Domain Model* and *Controller*. Domain Model reflects an operational view of the final system, describing domain-specific components. The components belong to one of these three groups: Infrastructure, Platform, and Solution. The *controller* encapsulates core logic of interpretation, it is responsible for execution of the following operations: process DSL input, change the model according to that input, update the view depending on user actions, select matching template, pass parameters based on the updated model, communicate with the cloud provider API.

Decoupling of *data tier* from the application logic guarantees interchangeability of service templates and domain model components as well as data consistency in the case of system crash. Data tier encompasses application backend and cloud service provider. Backend represents the metadata of user defined DSL services as well as cloud provider service templates; it includes creation date, name, its current status. Template storage preserves service descriptions uploaded by the user. Cloud provider represents runtime environment where services are deployed. It encloses several predefined service templates that can be selected, parametrized and deployed by the application. Manipulation of the templates is done via REST API of the provider.

Domain Model

Domain Model represents a map that stores information about all possible service components that can be included into the final service. It is defined by object representation of component and their properties. This map changes during the time of interpretation. Properties are defined as customizable parameters that are adapted to the DSL input. The final model is reduced to the components that are deployed. An example of such a model for ECM applications is depicted in the Table 2.1. However, it does not depict the initial state of the model. Thus, not all the components with their parameters are covered.

Model			
Group	Component	Properties	Component Status
Infrastructure	VM	OS CPU RAM	true
	Database	Vendor Transactoins per day	true
Platform	Object store Application server Web UI	Number of object stores Number of domains Number of workspaces	true true true
Solution	Content collector	Type of collected content	false

Table 2.1: Model of ECM Components

Components are grouped according to their deployment similarities:

- Infrastructure. Typically includes back-end components that are critical for system performance, could scaled-up or scaled-out depending on customer needs. Includes virtualized hardware components as well as software components that are not domain-specific.
- Platform. Involves components that are typical for every solution. They could be also deployed once and then adapted to the user by further migration and configuration.
- Solution. Describes optional component that are unique for each user and can be deployed only after interpretation of requirements.

3 Prototype implementation

In order to prove the concept elaborated in the previous section, a prototype of an interpretation application with a web user interface is implemented. For demonstration purposes two service templates simulating user requirements are defined. The translation process identifies the most appropriate service topology optimized on cost or availability parameter. Selection of exact service components is associated with one of the HOT templates of OpenStack. As a result their deployment in a cloud environment is to be triggered.

3.1 Bird's-eye view

In order to demonstrate general view on implemented prototype, the whole picture with actors involved in interpretation process is presented. It leads to clarification of core components of the prototype as well as a justification of their selection. Thus, generalized view on the implementation in the context of ECM user input and cloud provider environment is presented as depicted on the Figure 3.1. It represents actors of the system, operational entities, their implementation and justification of their necessity. Actors define participating parties in ECM service delivery process. Operational entities represent artifacts they cope with. The Implementation column identifies technologies that relate to the artifacts, e.g., domain-specific language foundation, Java classes and cloud orchestration template. Justification column gives a short introduction to each artifact implementation. The prototype implemented in this work will represent the Interpreter operational entity belonging to ECM Service Provider.

Core logic of the application is trapped between DSL input and runtime environment. Thus, three principal actors are represented: ECM User, ECM Service Provider, and Cloud Provider. The user specifies his requirements in the syntax of the domain-specific language. Notable is the fact the Metametamodel and Metamodel are defined by the service provider. So ECM user is not able to change them, the only creation of its Model based on already existing language syntax and vocabulary is possible. ECM Service Provider identifies the meaning of DSL vocabulary and initiates deployment of appropriate service. Cloud provider arranges the runtime environment for future services.

ECM user describes the needs by means of domain-specific language. Its syntax is specified in TOSCA standard, which plays the role of metametamodel. It defines following notations: *Node, Policy*. This knowledge is essential for interpretation of the language, e.g., parsing of user

Actor	Operational entity	Implementation	Justification
ECM User	Metametamodel	TOSCA v1.0	Identify syntax of DSL
	Metamodel	ECM DSL	Define domain vocabulary
	Model	Nodes, Policies	Specify user requirements
ECM Service Provider	Interpreter	DSL Parser	Parse user input and store it locally
		Rule Engine	Assign component categories to each observed Node
		Topology Calculator	Identify exact service components
		Template Selector	Select service template from the catalogue
Cloud provider	Service template	OpenStack HOT template	Encapsulate service deployment

Figure 3.1: General view on DSL interpretation

defined templates. Nodes define component groups, policies themselves influence on the exact components selection and definition of their properties. ECM DSL fills in the vocabulary of the language, specifying what components can be used by the users to describe their needs. Thus, it is named metamodel of the service. It includes extensions of standard TOSCA node types by ECM specific types, as well as the definition of policies and relationships. Knowledge of metamodel is required to identify the semantics of the vocabulary, e.g., exact meaning of each word in the implementation domain. The model itself encloses the needs of the customer. In other words, it represents the vocabulary defined in the metamodel that follows the semantics of metametamodel. Final user selection will consist of system requirements.

Processing of the input is done by the interpreter, which was developed in Java. It serves to assign semantics for each word of the language that is observed. Following core components can be identified: DSL parser, Rule Engine, Topology Calculator and Template Selector. DSL parser is responsible for parsing of the input template that constitutes to TOSCA semantics. It creates an internal representation of observed user input separating functional and non-functional requirements to the system. Rule engine processes each functional requirement assigns them to implementation components and reduces the initial model of all possible components. Topology calculator creates the graph of selected components considering their compatibility and optimization priority. After that, it calculates the path that identifies the exact components that reflect user needs. Finally, template selector determines the cloud service

template with minimal Hamming distance and initiates its deployment in a designated cloud environment.

A cloud provider plays the role of the execution environment for services identified by the interpreter. It must provide an API that allow to instantiate services. In this case, OpenStack IaaS provider was chosen. Motivation to adopt IaaS model was followed by flexibility in the configuration of the service including network parameters and software components. Furthermore, OpenStack provides Heat orchestration engine that allows to define services in the form of templates. Theoretically it is possible to define the service on PaaS level. However, it is considered as out of the scope of this work. Hereafter only IaaS cloud provider with already configured network topology is considered as backend of the interpretation.

3.2 Core considerations

In order to simulate the user input, a prototype of ECM domain specific language provided by IBM was utilized [Kuk]. It is defined in XML and follows TOSCA v1.0 standard. According to previously highlighted need in separation of user requirements following core decisions were made: nodes are associated with functional requirements related to components, their properties define non-functional requirements targeted to their configurable parameters, and policies applied to the nodes play the role of non-functional requirements.

In order to observe the case with high semantic difference between input and output of interpretation, it was decided to use high abstraction level components describing user service requirements. The need for archiving service is expressed in two components: "Repository" and "Web Client". This notation represents the necessity to store its data in an archive and access them via web GUI. The number of users of this application could be modified and influence on parameters of the virtual machine instances by means of *flavor* in OpenStack notation. Rule engine maps each TOSCA node to a list of component categories. In the case of "Repository", it is assigned to the following categories: virtual machine, database, directory service, object store and application server. In the case of "Web Client" direct mapping to web UI component is realized. Notable is that category does not specify exact components, only a group of possible implementations.

For the demonstration of topology selection based on different priority dimensions, it was decided to use cost and availability metrics. They reflect non-functional requirements to the service. Selection of these metrics was driven by the fact that opposite values of both metrics are demanded by the users, e.g., highest availability and lowest price. Components aligned on one priority dimension do not intersect with the ones selected according to the other priority. This fact leads to two opposed cases in topology determination: lowest price solution constitutes to the shortest path in the graph while solution with high availability leads to the longest path in the graph. A typical factor that differentiates priority dimensions is the target environment of the application. It can be designated for either development or production,

where the first one requires a minimal functional system with lowest cost and the second one demands a highly available service. Hence, two TOSCA policies can be applied to the components: "Development environment" and "Production environment".

In order to address the concept of partial matching of user needs as well as the non-functional requirements related to components, the policy "Continuous availability" is defined. Its application to one of the components will be interpreted as a need to deploy the database with foreseen scalability and high available cluster. In the prototype, it is mapped to the *DB2 Pure Scale* component.

Two service templates (*stacks*) describing orchestration sequence of ECM components were predefined. They are associated with a unique ID that reflects their content. The decision of what template is to be deployed is made based on the Hamming distance between the ID of selected components and the ID of one of the mentioned templates. Enterprise content management services represent highly complex systems, and their deployment requires numerous orchestration scripts. Thus, in order to concentrate more on interpretation logic it was decided to use pattern templates that simulate the deployment of ECM components. Since the focus of this work is the interpretation process, this decision does not make significant impact on the results of the thesis and the implemented prototype.

3.3 Interpretation process

The resulting prototype represents the system that interprets DSL input and selects one of the predefined service templates to be deployed in a cloud environment. In order to demonstrate that service topology can be calculated on the fly and optimized based on the input, two types of user templates are considered. The user either needs a service with minimal cost or highest availability. The need for the service with minimal cost is required for development purposes while the solution with high availability is used in a production system. The template with the lowest cost components is named "OneBox", while the template with high availability is defined as "FourBox". Their content, as well as, domain meaning assigned by the interpreter, are depicted on the Figure 3.2. This conventional naming is required for future references in this work. In the following section, the process of interpretation of both types of input is considered. Furthermore, comments are given on the difference in topology treatment and possible extension of the method.

The user input is packed in zip files containing description of nodes, relationships, and policies in TOSCA notation[TOS13b]. For conventional purposes, the term $node$ is used as a synonym of a $NodeType$ element in TOSCA notation. The original purpose of relationships is defined as prediction of orchestration sequence, since non-atomic components are observed in the input, they can not provide the whole orchestration sequence of exact implementation entities[1].

[1] atomicity property of components is discussed in the section 2.3

Listing 3.1 Example of TOSCA policy type and node template

```
<NodeTemplate name="Repository" id="ECMRepositoryNode" type="ns1:ECMRepositoryNode">
</NodeTemplate>
<NodeTemplate name="WebClient" id="WebClient" type="ns1:WebClient">
</NodeTemplate>

<PolicyType name="DevEnv">
<DerivedFrom typeRef="RootPolicyType"/>
<PropertiesDefinition element="pp:HAProperties"/>
<AppliesTo>
<NodeTypeReference typeRef = "ns1:ECMNode"/>
</AppliesTo>
</PolicyType>
```

Based on this consideration, the relationships are not utilized for further interpretation, only nodes and applied policies are considered. They are represented in XML notation, an example of nodes and policies is depicted in Listing 3.1. The original DSL input consists of several XML files containing nodes, their capabilities, and requirements. However, only node names, their inheritance and applied policies are processed. Thus, other parts of DSL as well as TOSCA implementation details are not discussed here. The content of both "OneBox" and "FourBox" user input contains the same node types: "Repository" and "Web Client". They differentiate only on the applied policies. "OneBox" contains only "DevEnv" policy while "FourBox" has "ProdEnv" and "ContiniousAvailability" policies. This differentiation leads to the selection of non-similar components from each category.

Processing of nodes

As soon as the user template is uploaded, the interpreter parses the XML document, identifies nodes and policies and preserves them internally. Important thing here is that interpreter knows that node types are mapped to functional requirements only, in other words to categories of service components; policy types relate to both functional and non-functional requirements, e.g., exact service components; and properties of the nodes have direct mapping to the options of exact service components. This knowledge defines the processing order of DSL elements according to defined concept. Hence, the sequence of interpretation is as follows: node types, policies, properties of the node types. In the first step, each node is consumed by the rule engine. In the prototype, it is implemented by a list of "$if(NodeType) \quad then \quad action$" statements that check each observed TOSCA $NodeType$ on matching to one of the known nodes. If such a node is identified, the action on assigning atomic component categories is executed. The categories are taken from the map of all possible domain components reducing the initial map.[2] It is realized by the inclusion of the results provided by the rule engine.

[2]Initial map reduction was discussed in rough component determination in the section 2.3

DSL template name	Included TOSCA elements	Domain meaning assigned by interpreter
One Box	Repository node	Database, Directory Server, Object store, Application Server categories
	Web Client node	Web UI category
	Development Environment policy	Install the system on one VM
Four Box	Repository node	Database, Directory Server, Object store, Application Server categories
	Web Client node	Web UI category
	Production Environment policy	Install the system on four Vms with high availability property
	Continuous Availability policy	Database with active-active replication

Figure 3.2: Components of utilized DSL templates

Following component categories are present in the model: virtual machine, database, directory server, object store, application server and a web user interface. The list of categories is presented on the Figure 3.3. Internally each category is represented by Enum class as depicted in the Listing 3.2, which includes implementation components of such a category. Each object is initialized with its index, price metric, availability metric. Set of all possible categories are stored in a sequence that reflects their deployment order, e.g., the first element should be deployed first. However, in case of production application data about categories could be stored in a database. For conventional purposes it is directly implemented in POJO based on the following reasons: (1) category components are static, (2) internal presentation eliminate the need to connect to the database to retrieve component information.

Processing of policies

On the next step policies applied to nodes are processed. Until this moment, interpretation of both "OneBox" and "FourBox" templates does not differ due to identical node types. However, from this point interpretation of each case faces certain differences. In the processing of non-functional requirements, policies that are applied to exact components are processed first.

Category	Component	Availability	Price	ID
Virtual machine	1VM	1	100	01
	4VMs	2	1000	10
Database	DB2	1	100	01
	DB2 with HADR	2	500	10
	DB2 Pure Scale	3	1000	11
Directory Server	Tivoli Directory Server	1	100	01
Application Server	IBM Web Sphere Application Server	1	100	01
	IBM Web Sphere Application Server cluster	2	500	10
Object Store	CM8	1	100	01
	FileNet P8	2	500	10
Web UI	IBM Content Navigator	1	100	01

Figure 3.3: Categories of service components

It leads to the selection of only the one component from designated category. In the case of "OneBox" there is no such a policy. Hence, all components mentioned in Enum category are considered in the process of topology determination. However, "FourBox" template contains "Continuous Availability" policy that is interpreted as the requirement to install database with active-active replication. Interpretation engine searches for Database that constitutes to that policy. In our case, it forces to deploy DB2 Pure Scale component in the final service. Thus, the list of components in Database category is reduced to only the one above mentioned component.

The next step of interpretation includes processing of policies designated to optimization priority. As mentioned earlier, it is defined by availability and cost. In order to calculate the path of components that are being deployed, each component of the system is assigned with a metric. These metrics are depicted in the Listing 3.2 and defined in $price$ and $availability$ properties of class objects. "DevEnv" policy will result in the selection of price property that will be used as a weight for transitions between nodes, while "ProdEnv" will force to choose availability field. In this prototype, assignment of both metrics was based on the following principle. The component without high availability support was given the smallest metric (DB2 with availability 1), component with active-passive replication was given a bigger value (DB2 with High availability disaster recovery has a value of 2), component with active-active replication has the highest value (DB2 Pure Scale has availability property has a value of 3). In the case of other components, scalability criterion was used in order to differentiate availability

Listing 3.2 Internal representation of component categories

```
public enum DBTypes{
DB2("01","100","1"), DB2_PS("10","1000","3"), DB2_HADR("11","500","2");
private final String index;
private final String price;
private final String availability;

DBTypes(String index, String price, String availability){
this.index = index;
this.price = price;
this.availability = availability;
}
public String getIndex(){
return this.index;
}
public String getPrice(){
return this.price;
}
public String getAvailability(){
return this.availability;
}
}
```

of components. The ability to create a cluster of components increases the chances of the system to stay alive during pick workloads. Thus, IBM Web Sphere Application server cluster has more availability coefficient that the same server without clusterization property. The price of components was specified proportionally to their availability index, the higher the availability, the more it costs.

Graph traversal

According to the elaborated concept, components are stored in the graph that allows to calculate the exact path from the first to the last element of the orchestration order. After processing of user requirements mapped to service components, each component category is substituted by exact implementation components generated from its Enum class. Each exact component is represented by a class that implements $IServiceComponent$ interface. The final graph of components is represented in a $HashMap$ of vertexes with initiated weights of transitions between them. Connections between nodes based on both orchestration order of components categories and compatibility between the components defined in the compatibility matrix. It is represented by two constants, first encapsulates all components of the system, second stores the compatibility vector for each row.

In order to prove the method of calculation of optimized service topology, modified version of Dijkstra shortest path finding algorithm is implemented [Cor09]. The modification was

made based on the requirement to calculate the longest distance between nodes. It was achieved by saving not only the connection from the previous node with smallest weight, but by keeping the information about all connections preserved in a $HashTree$. Capturing the last element of the tree will give the component with the highest metric. Hence, iteration from the latest node based on the highest metric leads through the path with the longest distance. Implementation of Dijkstra consumes a graph with initialized edges and vertexes and calculates all possible paths from the given first node. For two given user input formats, interpreter either asks for the shortest path between given nodes in the case of "DevEnv" policy or the longest path in the case of "ProdEnv" policy. Depending on the applied policy, different weights are calculated during initialization of the graph. Important is that the graph of components is not statically stored, but it is generated based on user input. The weight for each transition between edges is calculated following identified optimization priority. These features provide further extendability of the given method to select paths with exact metrics or from a given interval.

Stack selection

Once a service topology is identified, it is mapped to one of the existing stacks. OpenStack environment was selected as an application runtime for several reasons. It provides its *Heat orchestration engine* out of the box that allows to specify VM provisioning parameters, as well as to start the scripts that instantiate the deployment of desired services. Another feature of this environment is the ability to deploy a hybrid cloud environment based on diverse hypervisors [opeb]. Mapping of the template is based on the calculation of Hamming distance between ID of selected topology and one of the stored template. The ID distinguishes each component in its category, the concatenation of IDs of all entities that were selected on the previous step forms a unique binary number. At the same time, the Heat template is given with an ID that was formed, in the same way, e.g., depending on included components. Metadata of each registered template is stored in SQLite database. The decision to select this database was motivated by its ability to persist the data locally without the need to run a database server. It persists the information about user DSL templates, its status and assigned OpenStack template. The number of not matched symbols after comparison of ID of selected topology and one of the registered templates is the desired Hamming distance. In the prototype, it is defined that more than 80% matching template leads to the automatic service deployment. However, in the case of future extension it can by either configured by service provider or asked from the user if she accepts such a service. In the case of "Continuous Availability" policy, it forces to use DB2 Pure Scale as a database. However, if there no such registered template that includes this particular version of the database, then the most relevant service template is selected.

At the final step, properties of the node are processed and added to the internal component's model. Each observed property is checked matching to the rules. In our case, more than ten users of web UI is mapped to a mid-size flavor of a virtual machine that is used in service

deployment. This task is done by passing parameters to the Heat API at the moment of deployment of the template.

3.4 Application architecture

Implemented prototype belongs to typical multi-tier application architecture [ZLZW02]. Since this well-known design pattern can be achieved in a variety of technology stacks, the discussion of pros and cons of each third-party software component is left out of the scope of this work. Application layers include web user interface, Java-based server part, SQLite database and OpenStack cloud backend. In the following section, light is shed on the technologies that were used in order to implement such a system, as well as insights into implemented Java packages and selected classes.

Utilized software and hardware

In order to implement the prototype certain already existing technologies were utilized. The application was developed with Java Standard Edition in Eclipse IDE and is compatible with Java Virtual Machine version 1.7. DSL interpreter is built with Maven and deployed in Apache Tomcat v7 application server. Its architecture is presented in the Figure 3.4. According to the requested URL, each HTTP request to the server is passed to one of the registered Java Servlets. For every GET request to URL associated with the index page of the application, JSP page with injected values is passed to JVM, then HTML page is generated and returned in HTTP response back to the user. This page includes among others JavaScript code that plays the role of a client application. It is responsible for capturing the user behavior and processing HTTP responses from the server. In order to render HTML page following JavaScript libraries were used: JQuery, Bootstrap.js. JQuery provides an API to work with DOM elements, as well as encapsulates functionality to communicate via HTTP protocol. In order to connect to the database, the JDBC driver was used. During its life cycle the application operates with JSON files. That is the data format used in communication between web client, server, and OpenStack backend. In order to manage these files, Jersey Java library was utilized. The metadata of uploaded service templates is stored in SQLite database.

For the purpose of simulation of the customer environment, private cloud infrastructure is installed. The setup was done using IBM Cloud Manager (ICM) with OpenStack [IBM15]. ICM is a software product meant for deployment of both public and private cloud infrastructures. A private cloud instance is installed on IBM x3650 M2 server with RedHat version 6.6; the setup includes OpenStack with following components: Heat, Compute, Identity, Network. Every component provides an API, in this work only two components were called directly from the application: Identity API v2.0 and Orchestration API v1.0 [opeb].

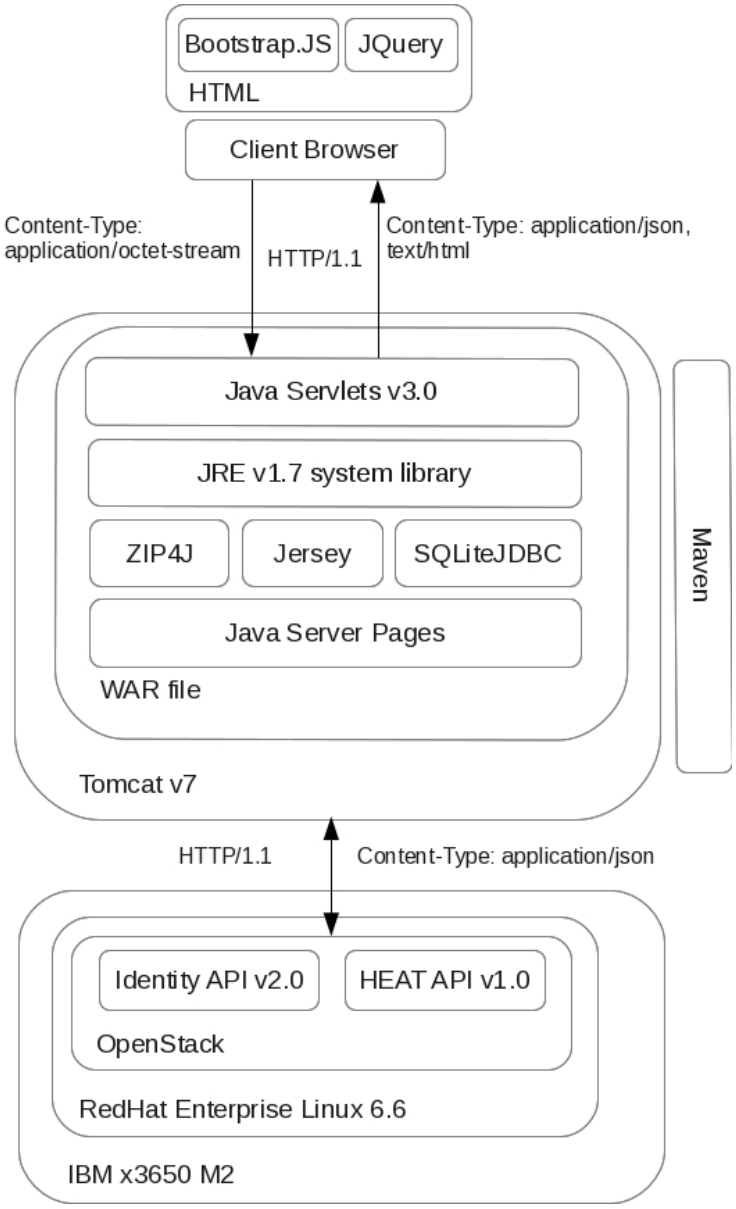

Figure 3.4: Technology overview

Class diagram

On the Figure 3.5 reduced class diagram of server part of the application is represented, it shows the main components excluding Enum classes of ECM model for expressiveness. Incoming HTTP user requests are processed by one of the class instances extending Java Servlet class. These classes are located in the $PageHandlers$ package. Mapping between each pair page handler and URL is done directly in the class by means of Java annotations using the

following syntax $@WebServlet("/URL")$. Following classes represent such a handlers that dispatch user requests:

- *Default.* Mapped to the main page of the application, shows template list with their status.
- *Delete.* Processes user request, deletes the template and its metadata.
- *Deploy.* Initiates deployment of the template by means of initialization of a new cloud stack with most probable components.
- *Download.* Returns requested template in ZIP format.
- *Status.* Responsible for the retrieving status of deployed stack associated with ECM DSL template.
- *Upload.* Process input form data, unpack and allocate uploaded template on the disk.

Every page handler operates with the instance of class *Controller*. That is the key entry point to the functionality of the application. It provides following public methods: *parseTemplate()*, *deleteTemplateAndStack()*, *uploadTemplate()*, *deployTemplate()*, *getTemplateStatus()*. It returns the list of templates with their metadata, interacts with OpenStack API for updated stack status that is reflected in the UI. At the same time, it encapsulates core logic of parsing the input, management of metadata, choosing of model components, and deploying a stack. The functionality of the Controller class is dependent on the following classes:

- *The DSLParser* class is responsible for resolving of dependencies between the DSL nodes and creation of its internal representation. ServiceTemplateResolver class initiates an instance of TopologyGraph class that stores the uploaded template; it depends on classes Node and Enge - abstractions of graph elements. Node class implements IMatrixEntity interface in order to be assigned to observation matrix, which is to be used in the calculation of ECM service components after this. Class PolicyResolver applies global policies defined in the template to all nodes and changes their structure.
- *The RuleEngine* class is responsible for the initial interpretation of DSL template model. It assigns component categories to observed TOSCA NodeTypes.
- *The TopologyCalculator* class encapsulates methods that implement Dijkstra shortest path finding algorithm that allows to identify exact service topology.
- *The MetadataDB* class is an adapter for metadata database, which provides create, retrieve, update and delete methods on the entries. It encapsulates low-level SQL queries so that the other application components operate with object representations of table rows implemented by classes DAOHeatOrchestrationTemplate and DAOTemplate.
- *The IServiceComponent* interface represents elements of ECM model. In other words, the classes implementing this interface are the service components from ECM domain.

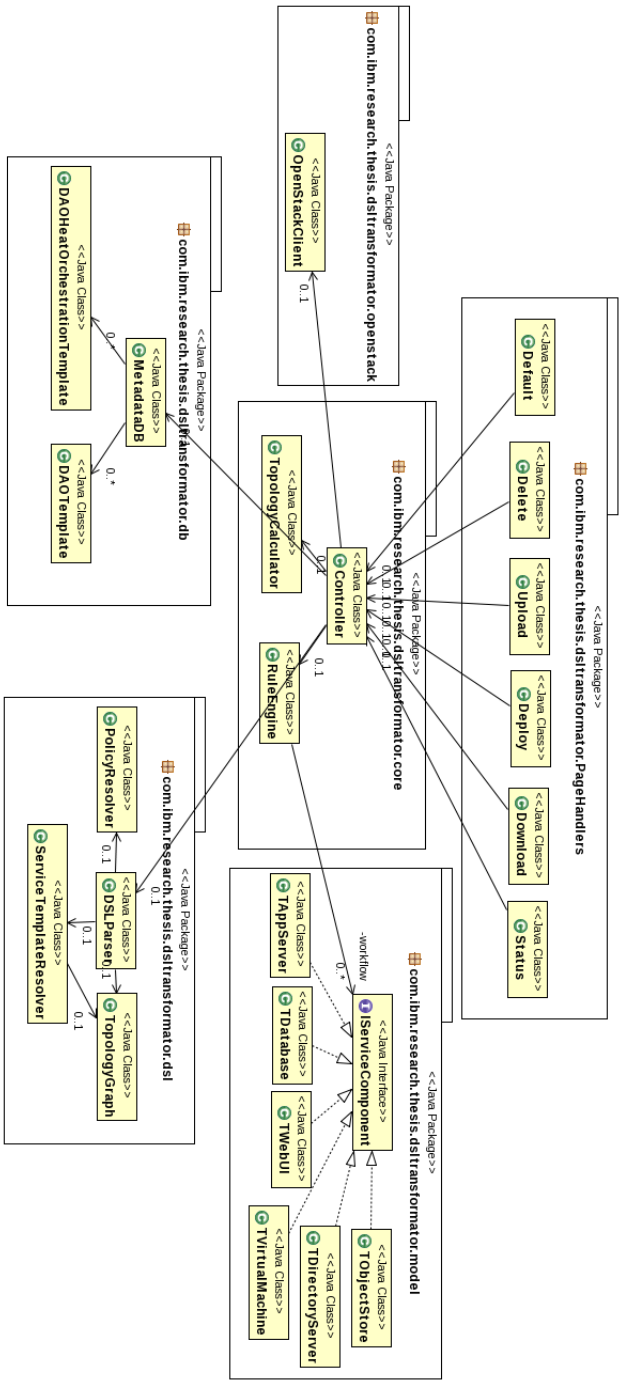

Figure 3.5: Class diagram

- *The OpenStackClient* class represents REST client that provides operations for authentication, stacks manipulation in OpenStack environment.

3.4.1 User interface

In order to interactively demonstrate the management of DSL templates, a web GUI was implemented. It provides upload, deploy, download and delete operations on the templates.

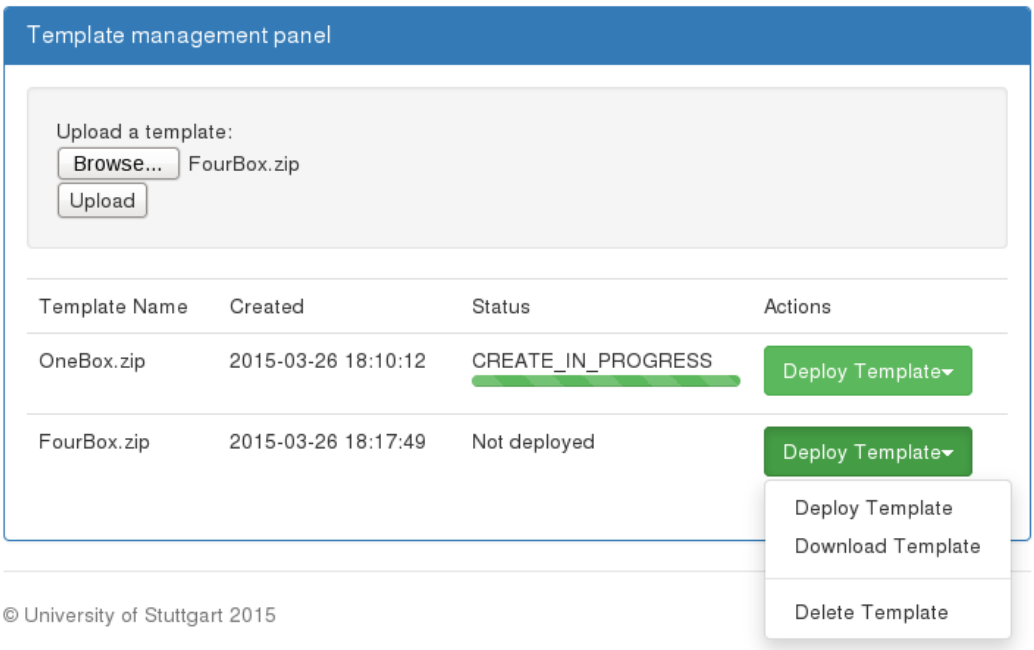

Figure 3.6: Application user interface

After sending a GET request to the application URL, an HTML page with the list of deployed templates is returned. From this point, the user interactions with the application are done asynchronously. It is necessary to track template status because it excludes the need to update the page when a new status is required. After the click on the $Deploy$ button, in the case of successful deployment, application returns stack id of the template. At that point JavaScript application initializes automatic tracking of the status of the template. It is realized by sending an HTTP get request on the following URL: $/status/template-name/$, the sequence of request processing is depicted on Figure 3.7.

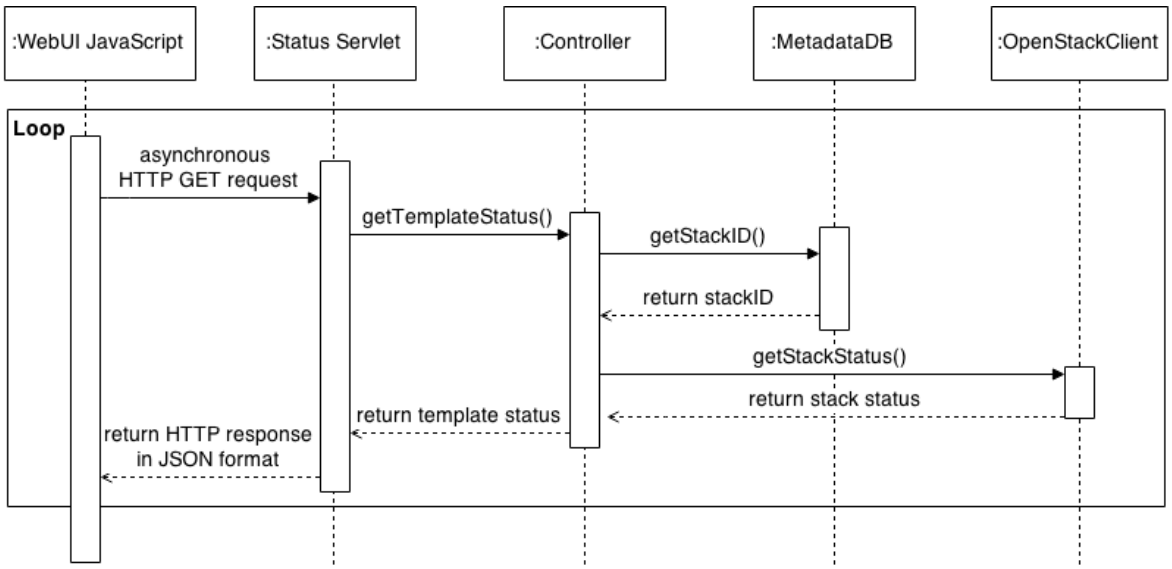

Figure 3.7: Sequence diagram of retrieving the template status

Listing 3.3 represents an asynchronous retrieval of template status. It keeps the user informed about the status of the template when deployment is initialized. The purpose of the updates is to help the user to resolve the errors that could arise during template processing, as well as to notify in the case of favorable result. It is equal to the current state of the stack that is associated with the template retrieved from Orchestration API of OpenStack. Once satisfied reply on template deployment request is received, the function is executed in an infinite loop until the page is reloaded. However, if the refreshed page contains templates with started deployment, then this function starts its execution again. It accepts two parameters $templateName$ - a template that status is required to check and $statusCell$ - TD element of HTML page which shows template's status. In the body of the function, AJAX method of JQuery library is called that performs GET request and accepts response in JSON format. Returned file has two possible formats: (1) "stack_status":"value" in the case of successful operation and (2) "error":"value" in the case if status is not available. In both cases, the response is extracted and inserted into p tag of the corresponding table cell. As soon as response processing is accomplished, function execution will halt for 5 seconds and then it continues by recursively calling itself. The key point here is the recursive call which should be executed only upon completion of the request. Otherwise, too many open connections may accumulate on the server and influencing its performance.

JavaScript on the main page is also used to check user input. Only zip files are allowed to be present on the server, the name of the file has to be unique and does not conflict with any of already uploaded.

Listing 3.3 Asynchronous retrieval of template status

```
function getTemplateStatus(templateName,statusCell){
var template = templateName;
var cell = statusCell;
$.ajax({
url: templateStatusURL+template,
dataType: "json",
success: function(json,status) {
console.log(json);
console.log(status);
if("stack_status" in json){
var templateStatus = json.stack_status;
cell.find($("p")).html(templateStatus);
cell.change();
}
else if("error" in json){
cell.find($("p")).html(json.error);
cell.change();
}
},
complete: function() {
setTimeout(function(){getTemplateStatus(template,cell);}, 5000);
}});
};
```

3.5 Application lifecycle

In order to give more insights into the functionality of the prototype, application lifecycle is considered. Based on the actions initiated by the user, sequence diagrams illustrate the interconnection between application components presented in the previous subsection. Moreover, details on exception handling in the prototype are provided.

Main page loading

Loading of the main page is the very first action that user performs in the process of interaction with the application. Sequence diagram of this process is depicted on the Figure 3.8. The user's browser initializes HTTP session sending GET request to the URL of the interpreter. The request is dispatched by the *Default Servlet*. It creates a new instance of class $Controller$, and executes a method $getTemplates()$. In its turn controller calls the method of $MetadataDB$ class instance to retrieve all registered templates. When the information about the templates is retrieved, it is injected into default.jsp file that is converted into an HTML file and sent out in the form of HTTP response.

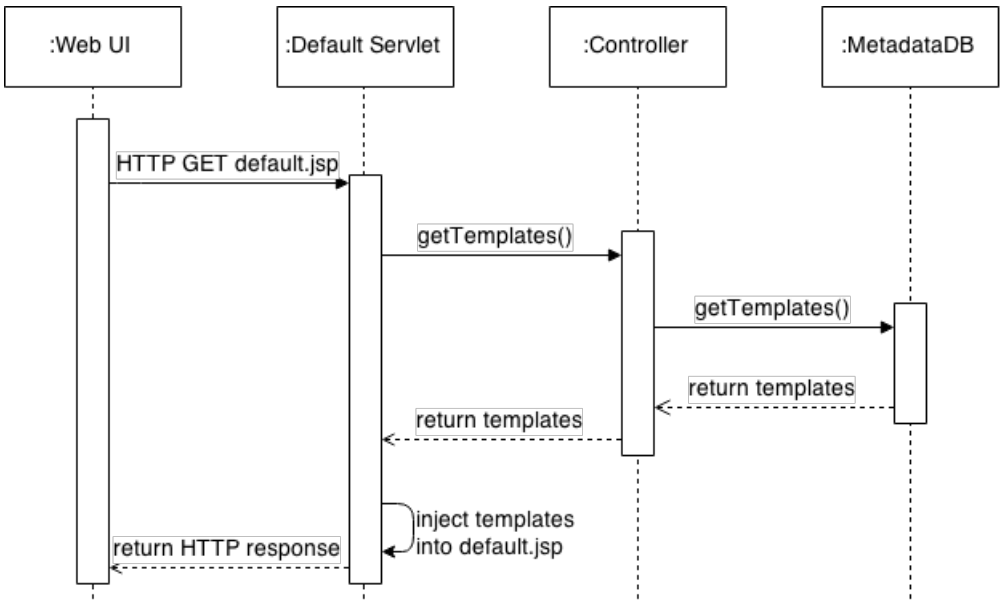

Figure 3.8: Sequence diagram of accessing the start page

Uploading of a template

Here the case when the user uploads a DSL template to the application is considered. After he specifies the template file and submits, the $uploadformHTTPPOSTrequest$ is sent to the server. *Upload Servlet* dispatches the request and extracts the part of it that contains the file - an archive in ZIP format. On the next step, the servlet calls $uploadTemplate()$ method of the controller, that in its turn performs the job of unpacking and registering in MetadataDB component. The archive is unpacked to the $uploads$ directory on the server, where it is available for future usage. Metadata of the uploaded template is stored persistently in the database, and it contains: template-id, name, creation data and associated stack ID. At the moment of upload, stack identifier is unknown. Thus, it contains a zero value. As soon as the deployment of this template is instantiated, and a matching stack is identified, the stack ID is assign to the template.

Deployment of a template

Once the template is uploaded to the server, the user can initiate its deployment. The deployment process is depicted in the Figure 3.10. Click on the $Deploytemplate$ button leads to sending out of HTTP POST request to the server with the name of the template in the URL. The request is received by $DeployServlet$ that initializes an instance of a class $Controller$ and calls the function $deployTemplate()$, passing template name as a parameter. Instances of the following classes will be initialized inside of the Controller:

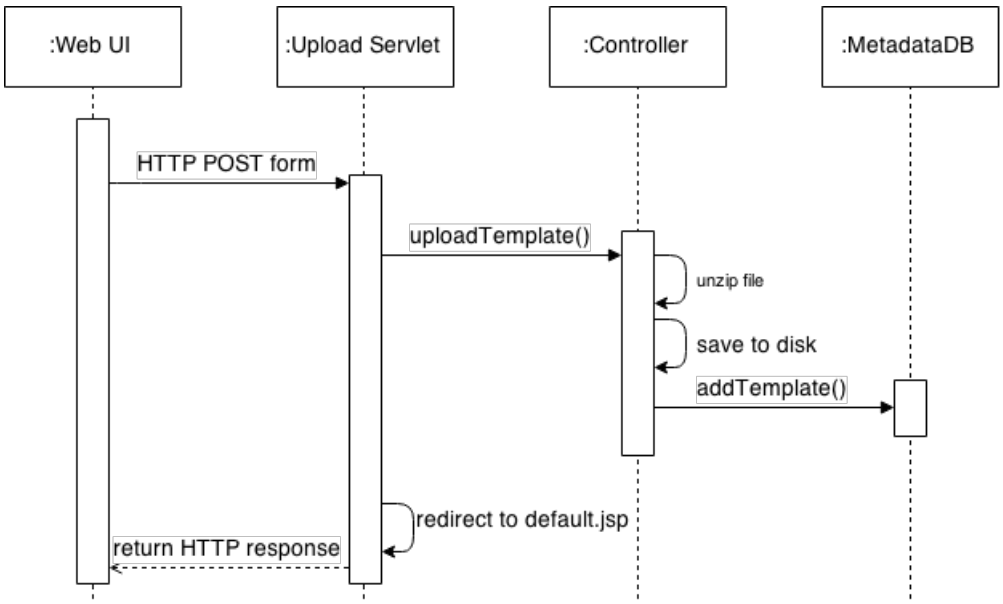

Figure 3.9: Sequence diagram of template uploading

$DSLParser, ModelManager, MetadataDB, OpenStackClient$. First parseTemplate function of DSLParser called. It looks up for the directory associated with the name of the template and search for ECM service files in the XML format. For each file, its POJO representation is created. Generation of Java classes mapped to the XML notation of DSL is done prior to parsing using Java Architecture for XML Binding $(JAXB)$. Retrieved elements of the service are not still ready for further processing and needed to be stored in a suitable data structure that reflects relations between DSL elements. For that purpose, graph abstraction is chosen to represent DSL service template. Its nodes relate to the abstract components, and its edges define the relationship between them. Thus, a bundle of parsed elements is transformed to topology graph and returned to the controller. The process of graph generation is not depicted on the figure. Once given the components from user template, the controller needs to find out what elements of its internal model are to be included into the final service. In order to accomplish this goal, the controller passes service template as a parameter to $getModelIndex()$ function of the $TopologyCalculator$. An instance of this class calculates either shortest or the longest path between the first and the last components of the model. The resulting path indicates exact components of the service. On the next step the index of resulting components is calculated. Every model element has its unique index from the group it belongs to, e.g. FileNet P8 component found in Object store group has an index 01. For the demonstration purpose index is defined as a two-digit binary number, causing maximum four elements in the group. The final index of the model is calculated as the concatenation of indexes of all the model components that were activated. After the calculation is done final index of the model is sent back to the controller, which calls MetadataDB component to return Heat template that is associated with this index. Based on the returned name of the Heat template, controller

triggers deployment of a new stack using OpenStackClient component. It returns stack ID in the case of successful initialization of the stack. At this moment controller is required to remember this ID for future usage and associate it with a user template. Hence it executes $setStackId()$ method of MetadataDB component. Once template status is successfully assigned, control is delegated to Deploy Servlet which sends JSON response containing stack ID to the client.

Deletion of a template

Deletion of the template from the user interface initiates the cascade of deletions of the data belonging to that template, its metadata, and the associated stack if the it is deployed. If the user presses the $Delete$ button in the UI that is associated to that template. Then JavaScript application sends HTTP POST request to the server that is dispatched by *Delete Servlet*, which calls deleteTemplate() method of the controller. It initiates deletion of template metadata in the MetadataDB component, which returns the stack ID of the template if it was previously deployed. Once given a stack identifier, the controller calls deleteStack() function of OpenStackClient component, which returns HTTP code 204 if the operation was successful. Finally, the controller returns the status of the template and the servlet redirects the original HTTP request to default.jsp page, it shows the updated list of templates.

Exception handling

Due to multi-tier architecture and the variety of integrated software components, thrown exception in one of the tiers is distributed back to the caller. Thus, it is important to handle thrown exceptions to prevent the user from observing the whole stack trace, and provide her with useful information instead. In order to guarantee exception handling, a class *ResponseMessage* is introduced. It represents a communication medium between application components. Instance of such a class has two parameters: status and message. The status is defined by *Enum* Java class and contains only two members: "SUCCESS" and "FAILURE". The message is represented by String type, since communication between components is done by either transferring of JSON or a simple sequence of chars, however, the message type could be substituted for generic Java Object class. It will allow to pass any objects and simply cast them to needed type upon receiving.

Exception handling is demonstrated on the example of getAPIToken() function of OpenStackClient class, depicted in Listing 3.4. It creates WebResource instance related to URL of OpenStack Identity API. Onwards it sends post request with access credentials and waits for the response. If the response status is not equal to 200, then we create a new instance of MessageResponse class with error status and explanation of the error. Otherwise, the authentication token is extracted from the JSON response and create a new instance of MessageResponse class with success status and the token in the message. During retrieval of the

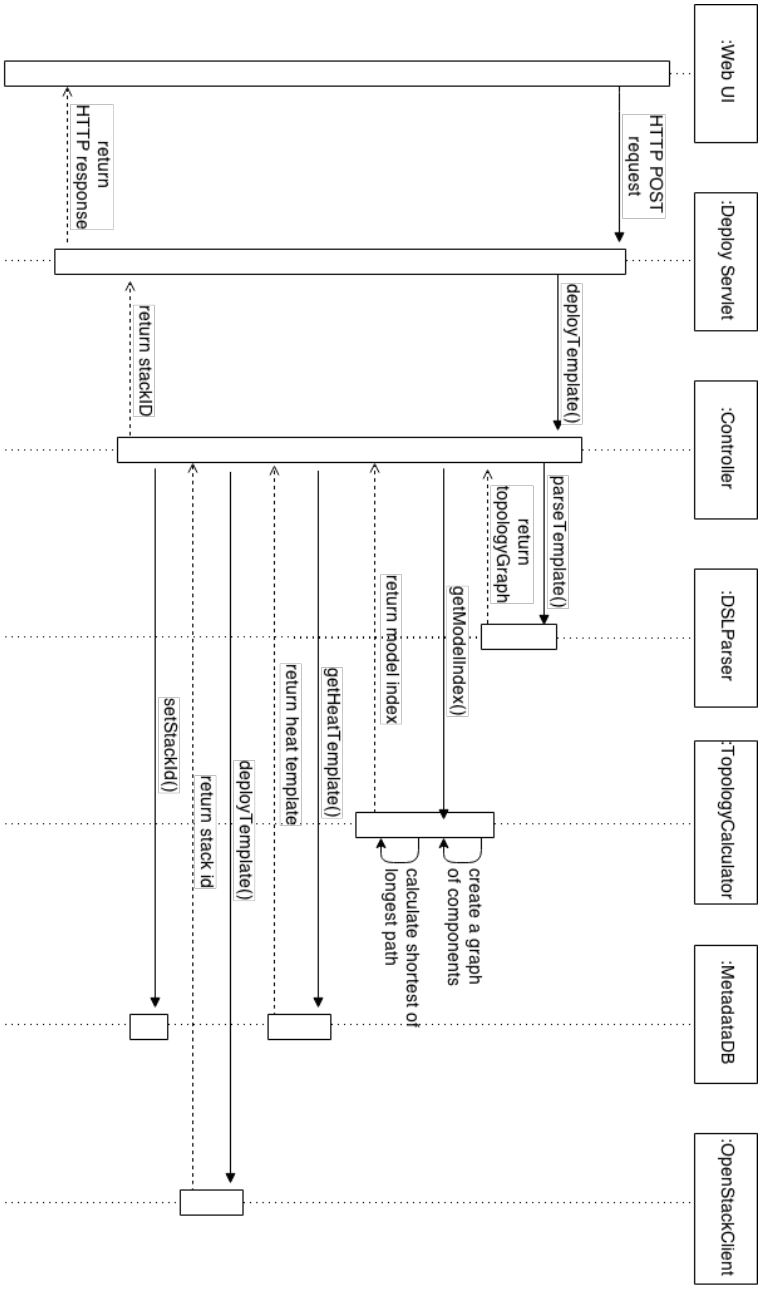

Figure 3.10: Sequence diagram of deploying a template

authentication token, several exceptions could be thrown, for example, web resource may be unavailable, the incorrect syntax of the request body in terms of JSON, wrong credentials, etc. The scope of this work represents a prototype and does not concentrate on the detailed investigation of all possible exception that may occur. Hence, all the exceptions which may

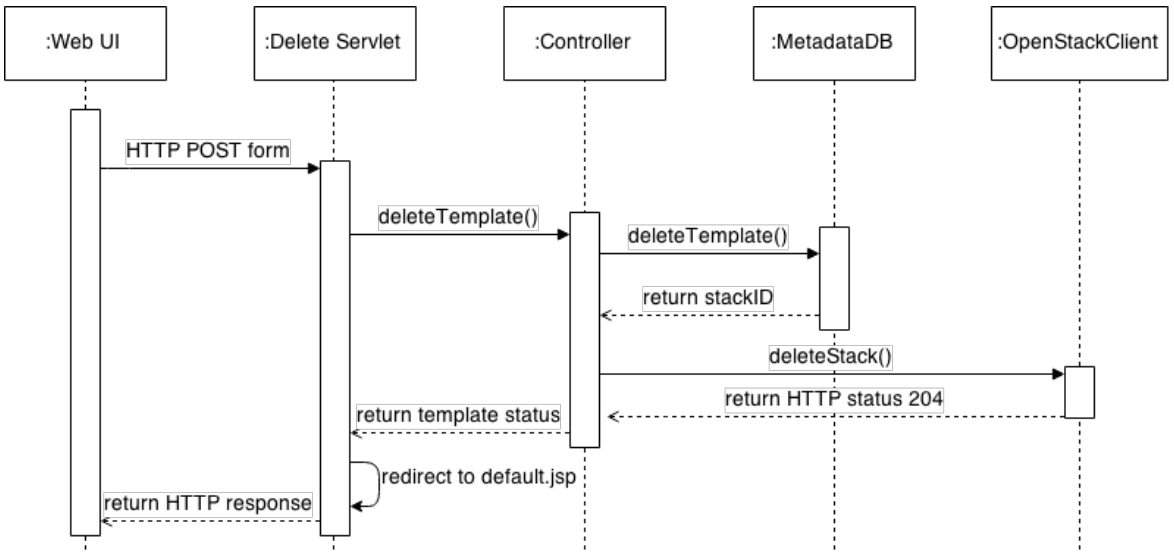

Figure 3.11: Sequence diagram of template deletion

be thrown in getAPIToken() method are handled by the creation of a response error message containing following text "Can not get token from OpenStack".

Based on the example mentioned above the following strategy for exception handling was elaborated. The $Caller$ method has two scenarios of execution: ResponseMessage with statuses "SUCCESS" and "FAILURE", it does not expect to handle exceptions but the response message. The callee method or function will take the whole responsibility for exception handling and generation of the explanation of what occurred if it was thrown. Eventually, the process of communication between application components comes down to interchanging of instances of the class ResponseMessage. This strict separation of responsibilities between components provides eliminates the problem of not handled exceptions that may occur during application runtime. At the same time, it guarantees error reasoning for the end user.

3.6 Discussion

The implemented prototype covers only several parts of the whole range of steps that are required to guarantee complete shift from traditional computing to the cloud model. Since non-considered steps like DSL development and creation of HOT templates are out of the scope of this work, evaluation of the prototype by performance test or subject matter expert surveys is difficult to realize. Therefore in order to evaluate the solution, one needs to analyze comparatively service delivery processes in order to observe the added value. In the following I compare previously utilized approach to the one addressed in this work.

Listing 3.4 Get OpenStack authentication token

```
public ResponseMessage getAPIToken(){
String token;
ResponseMessage getTokenResult = null;
File identityJson = new File(this.IdentityAPIConnectionFile);
// pass identity file to openstack identity API
try{
Client client = Client.create();
WebResource openstackAPI = client.resource(identityAPIAddress);
ClientResponse response =
    openstackAPI.type("application/json").post(ClientResponse.class,identityJson);
String entity = response.getEntity(String.class);
if(response.getStatus() != 200){
getTokenResult = new ResponseMessage("Can not get token from
    OpenStack",ResponseStatus.FAILURE);
}
else{
JSONObject jsonResponse = new JSONObject(entity);
token = jsonResponse.getJSONObject("access").getJSONObject("token").getString("id");
getTokenResult = new ResponseMessage(token,ResponseStatus.SUCCESS);
}
}
catch (Exception e)
{
getTokenResult = new ResponseMessage("Can not get token from
    OpenStack",ResponseStatus.FAILURE);
e.printStackTrace();
}
return getTokenResult;
}
```

Partially referring to the problem statement of this work I will reconstruct traditional process of ECM *on-premise* service delivery employed at IBM. Five essential steps are considered:

1. *Requirements engineering.* Retrieved by subject matter experts by means of queries and customer interviews.
2. *Selection of software components.* Manually performed by solution architects according to collected experience from previously delivered services and customer needs.
3. *Deployment of components in a test environment.* Done by operation team that is responsible for provisioning of virtual machines, network infrastructure, automation of software deployment.
4. *Testing.* Performed by quality assurance team that identifies test cases, reports found bugs either to the operation or directly to the development team.

5. *Service delivery.* The operation team executes the deployment of the system to designated production environment.

Last three steps could be done by the customer who already bought software products from ECM stack. However, I consider the case in which these tasks are included in the SLA and performed by the ECM service provider. Although certain steps for cloud archive components already have automation scripts, the key drawback of this method is that the routine is repeated the number of times proportional to the number of customers. In other words, as soon as a new customer with its requirements comes the process of individual service delivery iterates again. Needless to say that it reveals several directions for optimization that were elaborated in this work.

Modified service delivery model is reduced to these three steps:

1. *The user defines its requirements in DSL.* The usage of DSL directly substitutes the first step of the above-mentioned traditional process. The users can express their needs without subject matter experts. In order to reduce the ambiguity of user requirements, as well as the time of service delivery, it is assumed that they are clearly defined in the domain specific language. It is considered that the business user can declare his needs using GUI that persistently stores the result in the XML format acceptable by the interpreter.

2. *Automatic selection of service components.* Implemented prototype automates the selection of software components. It absolves of relying on the decision of solution architect since his expertise could be implemented in the interpretation application. The semantics of each DSL word is kept in the rule engine that assigns observed input to atomic service components. Knowledge of components interoperation is filed in the compatibility matrix. Non-functional requirements to the service are defined in priority direction of the system, and exact topology is selected from the graph of components. The method implemented in this work automates the optimization of service topology. Operation teams are required to create only the limited number of service templates with foreseen possibility of their configuration at the time of deployment. It is also considered, that user requirements do not always need to match the delivered service perfectly. Thus, finite number of service templates is required. If user defined requirements are partially met and their similarity coefficient is higher than the certain threshold, then the service is to be deployed.

3. *Automatic service deployment.* It eliminates last three steps of traditional service delivery. Deployment of the components in a test environment is not necessary because deployment scripts are already defined in service templates. Testing of the service is not performed before the actual delivery, but right after the moment of creation of a new template. As mentioned earlier, each service template has configurable parameters. Hence, all the variety of configurations is tested prior to publishing of the templates. The service delivery step previously executed by operation team is now automatically accomplished by the interpreter.

In order to provide a fully functional solution, each software component should have predefined scripts packed into a service template.

One can see that the problem addressed by this work provides a method of elimination of human interaction into the process of service delivery. It allows to reduce drawbacks of individual solutions and to offer customizable services for new customers. The key value of this approach is the ability to offer both public and private cloud services with possible extension to a hybrid model. In the case of public model the user obtains the service running in ECM provider environment with minimal onboarding costs. However, when the user finds it economically reasonable to have the service in its private cloud, the required service template may be purchased from ECM provider. The hybrid cloud model is still in the development, however, the results of various research allow to conclude that OpenStack environment is appropriate for this extension [DFCV14] [SPH+14].

4 Summary

This work addresses the problem of automatic interpretation of declarative service description in the context of the shift to a cloud environment. After discussion of related works conducted in the problem domain, three concepts of solution implementation were considered, and the one based on the gradual reduction of component map was chosen as a basis. The implementation of the concept allows to select exact service topology based on user input; it also introduces optimization criterion for the case of multiple matching components. As a proof of concept, the prototype of ECM DSL interpreter with web GUI was implemented. It allows the user to upload, download and delete service templates as well as to initialize deployment of matching topology. For demonstration purposes, two ECM service topologies were considered, and their pattern templates were elaborated. The templates address another accompanying problem - portability of service templates. OpenStack cloud environment was deployed and selected as a backend to guarantee the portability of services. One of the main benefits of it is the flexibility in the choice of hypervisors. While carrying out this work, several conclusions were made that will be reflected in this chapter.

ECM domain-specific language based on TOSCA v1.0 was consumed and interpreted by the prototype application. Its specification covers among others relationships between described components, management plans, and implementation artifacts. Relationships between nodes are not necessary when high abstraction level vocabulary is used since each observed node entity will be subdivided into atomic application components. It will lead to the creation of new connections between these components preserved in the rule engine that are not derived from observed user input. Management plans and implementation artifacts define deployment sequence and component installation scripts accordingly. Thus, they cannot be used in declarative service description based on the assumption that service implementation is hidden from the user. However, this functionality was not used during the interpretation. Hence DSL could have been defined without strict inheritance of TOSCA types. In the interpretation, only functional and non-functional requirements for the service were consumed.

The proposed interpretation method divides user input into three parts that are processed sequentially reducing the map of possible service components. First, functional requirements related to component categories are processed. They identify groups of components that belong to the service. Second, non-functional requirements related to exact service components are analyzed; this leads to further reduction of certain categories. On the last step, non-functional requirements related to component properties perform fine-grain tuning of the system. However, it was identified that after the second reduction step, several possible implementation

components could be present. In order to optimize the process of selection, graph abstraction was utilized. Its vertexes were defined by service components while edges reflected their compatibility and orchestration sequence. Based on the predefined transition weights between nodes one of the paths was chosen as a combination of final service components. Presented interpretation method gave impulse to the following conclusion:

- DSL syntax must allow to define types of user requirements. There is a direct dependency between the type and its processing order.
- Orchestration sequence of component categories is predefined. Otherwise, it must be concluded from user input.
- Optimization of topology selection requires predefined metrics. They are needed for the comparison of calculated weights of paths.

After determination of service components, they are mapped to one of the service templates from provider's catalog. They are predefined prior to interpretation. Generation of templates from scratch is not considered due to several objective reasons. Typically service providers offer only a limited number of services that can be predefined and customized for each user. Each template is tested before it becomes available, this guarantees more rapid service provisioning since there is no need for testing before actual delivery. Each template leaves space for configuration by passing the parameters at the moment of deployment. These parameters are derived from non-functional requirements related to component configuration from user input. During the research, it was identified that the number of possible combinations of service components is not always equal to the number of predefined templates. At the same time, usually users accept the service that is not fully matching their requirements. Based on the mentioned assumptions it was decided to find the template that has minimal Hamming distance that is above user acceptance threshold. Hence, this allows to provide the service with partially matched requirements acceptable by the user.

Future work

In order to guarantee full switch from traditional software delivery into cloud based services the following research directions were identified and highlighted. As mentioned earlier declarative input interpretation process is trapped between user input and underlying domain model with cloud runtime environment. This leads to the following future work directions:

- Clarification of DSL syntax and semantics. This could be realized by analysis of previously delivered software solutions and customer expectations. This will lead to selection of benchmark requirement bundles which will constitute to the service templates.

- Orchestration sequence preservation. Deployment workflow plays an important role in component selection, thus extension of the model by adding new components into the sequence must be investigated.
- Optimization of service topology based on metrics requires determination of priority dimensions. In this work only cost and availability metrics were selected, however their precise calculation as well as consideration of other possible metrics must be defined.
- Deployment automation. In order to guarantee cloud orchestration, deployment scripts must be provided for each component in the benchmark templates.
- Further evaluation of the concept. The concept must be evaluated in a larger set of service topologies in order to guarantee its robustness.

Bibliography

[ABC13] J. Alton Buddy Cleveland. Interoperability Platform. Technical report, A Bentley White Paper, 2013. (Cited on pages 16 and 17)

[AII13a] AIIM. Association for Information and Image Management. "What is Document Management (DMS)". http://www.aiim.org, 2013. Accessed: 2015-02-01. (Cited on page 19)

[AII13b] AIIM. Association for Information and Image Management. "What is Enterprise Content Management (ECM)?". http://www.aiim.org, 2013. Accessed: 2015-01-31. (Cited on page 19)

[Alp10] E. Alpaydin. *Introduction to machine learning*. The MIT press, 2010. (Cited on page 26)

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. OpenTOSCA – A Runtime for TOSCA-based Cloud Applications. In *11th International Conference on Service-Oriented Computing*, LNCS. Springer, 2013. (Cited on page 13)

[Ber14] J. Bergmann. *Requirements Engineering fuer die agile Softwareentwicklung*. dpunkt.verlag, 2014. (Cited on page 23)

[Bur10] B. Burke. *RESTful Java with JAX-RS*. O'REILLY, Sebastopol, USA, 2010. (Cited on page 16)

[CDM14] Cloud Data Management Interface (CDMI) Version 1.1.0. http://www.snia.org/sites/default/files/CDMI_Spec_v1.1.pdf, 2014. (Cited on page 17)

[Cha09] S. Chari. Confronting the Data Center Crisis: A Cost - Benefit Analysis of the IBM Computing on Demand (CoD) Cloud Offering. 2009. (Cited on page 9)

[CLY+14] C.-H. Chang, C.-W. Lu, W. P. Yang, W. C.-C. Chu, C.-T. Yang, C.-T. Tsai, P.-A. Hsiung. A SysML Based Requirement Modeling Automatic Transformation Approach. In *Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International*, volume 5, pp. 474 – 479. 2014. (Cited on pages 12 and 24)

[CNMK07] T. C. Chieu, T. Nguyen, S. Maradugu, T. Kwok. An Enterprise Electronic Contract Management System Based on Service-Oriented Architecture. In *Services Computing (SCC). IEEE International Conference on*, volume 8, pp. 613–620. 2007. (Cited on page 20)

[Cor09] T. S. Cormen. *Introduction to Algorithms*. MIT Press, 2009. (Cited on page 48)

[CS14] R. Chamberlain, J. Schommer. Using Docker to support reproducible research. Technical Report 1101910, figshare, 2014. (Cited on page 17)

[DFCV14] P. Donadio, G. B. Fioccola, R. Canonico, G. Ventre. Network Security for Hybrid Cloud. In *Euro Med Telco Conference (EMTC), 2014*, volume 6, pp. 1 – 6. 2014. (Cited on page 64)

[DHGS13] L. Dey, S. B. H., M. G., G. Shroff. Email Analytics for Activity Management and Insight Discovery. In *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), IEEE/WIC/ACM International Joint Conference on*, volume 8, pp. 557–564. 2013. (Cited on page 20)

[DWC10] T. Dillon, C. Wu, E. Chang. Cloud Computing:Issues and Challenges. In *Advanced Information Networking and Applications (AINA), 24th IEEE International Conference on*. 2010. (Cited on page 14)

[EHW04] E. Engel, R. M. Hayes, X. Wang. The Sarbanes-Oxley Act and Firms' Going-Private Decisions. *Journal of Accounting and Economics*, pp. 116–145, 2004. (Cited on page 19)

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns. Fundamentials to Design, Build and Manage Cloud Applications*. Springer, Wien, Austria, 2014. (Cited on pages 15, 16 and 17)

[Gar97] R. Garud. On the Distinction between Know-How, Know-What, and Know-Why. *Advances in Strategic Management*, pp. 81–201, 1997. (Cited on page 18)

[GHH+12] K. R. Grahlmann, R. W. Helms, C. Hilhorst, S. Brinkkemper, S. van Amerongen. Reviewing Enterprise Content Management: a functional framework. *European Journal of Information Systems*, 19:268–286, 2012. (Cited on page 19)

[GKK93] N. Gaitanis, G. Kapogianopoulos, D. A. Karras. Pattern Classification Using A Generalized Hamming Distance Metric. In *Neural Networks, 1993. IJCNN '93-Nagoya. Proceedings of 1993 International Joint Conference on*, volume 4, pp. 1293 – 1421. 1993. (Cited on page 36)

[IBM15] IBM. IBM Cloud Manager with OpenStack v4.2.0 documentation. http://http://www-01.ibm.com/support/knowledgecenter/SST55W/, 2015. Accessed: 2015-04-01. (Cited on page 50)

[ISO14] ISO. Information technology — Cloud computing — Overview and vocabulary. Technical Report ISO 17788, International Organization of Standardization, Geneva, Switzerland, 2014. (Cited on page 14)

[KML+14] G. Katsaros, M. Menzel, A. Lenk, J. Rake-Revelant, R. Skipp, J. Eberhardt. Cloud Application Portability with TOSCA, Chef and Openstack. In *Cloud Engineering (IC2E), IEEE International Conference on*, pp. 295–302. 2014. (Cited on page 13)

[Kuk] S. Kukhtichev. *Design and implementation of a Domain Specific Language for defining ECM workloads in elastic cloud environments using TOSCA*. Master's thesis. (Cited on page 43)

[Lea99] A. C. Lear. XML seen as an Integral to Application Integration, 1999. (Cited on page 16)

[LRS02] F. Leymann, D. Roller, M.-T. Schmidt. Web services and business process management. *IBM Systems Journal*, 14, 2002. (Cited on page 16)

[LY10] H. Liduo, C. Yan. Design and Implementation of Web Content Management System by J2EE-based Three-tier Architecture. In *Information Management and Engineering (ICIME), The 2nd IEEE International Conference on*, volume 5, pp. 513–517. 2010. (Cited on page 20)

[MJ99] M. J. Mahemoff, L. J. Johnston. Handling Multiple Domain Objects with Model-View-Controller. In *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 32. Proceedings*, volume 11, pp. 28 – 39. 1999. (Cited on page 37)

[MNS14] H. R. Motahari-Nezdhad, K. D. Swenson. Towards a Knowledge-Based Framework for Enterprise Content Management. In *System Sciencies (HICSS), 47th Hawaii International Conference on*, volume 10, pp. 3543–3552. 2014. (Cited on page 18)

[NCK+09] S. Nakamura, S. Chiba, H. Kaminaga, S. Yokoyama, Y. Miyadera. Development of a Topic-centered Adaptive Document Management System. In *Computer Sciences and Convergence Information Technology (ICCIT'09). Fourth International Conference on*, volume 7, pp. 109–115. 2009. (Cited on page 20)

[Net14] Web Server Survey. http://news.netcraft.com/archives/2014/05/07/may-2014-web-server-survey.html, 2014. Accessed: 2015-01-26. (Cited on page 18)

[opea] OpenGroup. http://www.opengroup.org/. Accessed: 2015-01-19. (Cited on pages 16 and 17)

[opeb] OpenStack - open source software for creating private and public clouds. http://www.openstack.org/. Accessed: 2015-04-21. (Cited on pages 49 and 50)

[PL12] G. Pabon, M. Leyton. Tackling Algorithmic Skeleton's Inversion of Control. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, volume 4, pp. 42 – 46. 2012. (Cited on page 13)

[Shr10] G. Shross. *Enterprise Cloud Computing. Technology, Architecture, Applications*. Cambridge University Press, Cambridge, United Kingdom, 2010. (Cited on page 13)

[SP10] P. Sutheebanjard, W. Premchaiswadi. Fast Convert OR-Decision Table to Decision Tree. In *Knowledge Engineering, 2010 8th International Conference on ICT and*, volume 4, pp. 37 – 40. 2010. (Cited on page 24)

[SPH+14] D. Sitaram, H. L. Phalachandra, S. Harwalkar, S. Murugesan, P. Sudheendra, R. Ananth, V. B, A. H. Kanji, S. C. Bhat, B. Kruti. Simple Cloud Federation. In *Modelling Symposium (AMS), 2014 8th Asia*, volume 7, pp. 83 – 89. 2014. (Cited on page 64)

[TOS13a] Topology and Orchestration Specification for Cloud Applications Primer Version 1.0. http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html, 2013. (Cited on page 13)

[TOS13b] Topology and Orchestration Specification for Cloud Applications Version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf, 2013. (Cited on page 44)

[UKKM13] T. Uchiumi, S. Kitajima, S. Kikuchi, Y. Matsumoto. Automatic parameter configuration for cloud infrastructures by design pattern extraction. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 8, pp. 224 – 231. 2013. (Cited on page 13)

[WBB+14] J. Wettinger, T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, M. Zimmermann. Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA. In *Proceedings of the 4th International Conference on Cloud Computing and Service Science, CLOSER 2014, 3-5 April 2014, Barcelona, Spain*, pp. 559–568. SciTePress, 2014. (Cited on page 13)

[ZAB+09] W.-D. Zhu, R. Aitchison, E. Bonner, H. C. Mendez, R. Rathgeber, A. Yadav, H. Yessayan. *Understanding IBM FileNet Records Manager*. IBM Redbooks, 11400 Burnet Road, Austin, TX 78758-3493, USA, 2009. (Cited on page 20)

[ZLZW02] S. Zhang, Q. Li, Y. Zheng, H. Wang. The Multi-Tier Architecture Based on Offline Component Agent. In *Computer Supported Cooperative Work in Design. The 7th International Conference on*, volume 4, pp. 241 – 244. 2002. (Cited on page 50)

All links were last followed on March 30, 2015.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than 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.

---

place, date, signature