FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Supporting Architecture Evolution in Industrial Cloud-Edge Ecosystems

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Stefan Gschiel, BSc
Matrikelnummer 1226352

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Mag. Mag.rer.soc.oec. Stefan Biffl
DI Dr. Daniel Schall

Wien, 18. Februar 2018

Stefan Gschiel	Stefan Biffl

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Supporting Architecture Evolution in Industrial Cloud-Edge Ecosystems

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Internet Computing

by

Stefan Gschiel, BSc
Registration Number 1226352

to the Faculty of Informatics

at the TU Wien

Advisors: Ao.Univ.Prof. Dipl.-Ing. Mag. Mag.rer.soc.oec. Stefan Biffl
DI Dr. Daniel Schall

Vienna, 18th February, 2018

Stefan Gschiel

Stefan Biffl

Erklärung zur Verfassung der Arbeit

Stefan Gschiel, BSc
Gutkeledweg 32, 7000 Eisenstadt

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. Februar 2018

Stefan Gschiel

Danksagung

Hiermit möchte ich mich bei allen herzlich bedanken, die mich während der Erstellung meiner Diplomarbeit unterstützt haben. Besonders bedanken möchte ich mich bei meinen Betreuern Stefan Biffl und Daniel Schall.

Weiters möchte ich mich bei Daniel Schall und der Firma Siemens dafür bedanken, dass sie mir ermöglicht haben in einem Unternehmensumfeld meine Diplomarbeit zu machen und dabei gleich berufliche Erfahrung zu sammeln. Von meinen Kollegen bei Siemens möchte ich mich noch neben Daniel Schall bei Sebastian Meixner bedanken für die erfolgreiche Zusammenarbeit im Projekt aus dem meine Diplomarbeit hervorging sowie für die Implementierung des Frontends des dabei entwickelten Tools. Natürlich möchte ich mich noch bei allen anderen Kollegen bei Siemens bedanken, die mich bei meiner Diplomarbeit unterstützt haben, besonders bei allen Teilnehmern an der Evaluierung meiner Arbeit.

Acknowledgements

I would like to thank everybody who supported me while creating my Master's thesis, especially my advisors Stefan Bifflˆ and Daniel Schall.

Furthermore, I would like to thank Daniel Schall and Siemens that they made it possible for me to do my Master's thesis in a cooperate environment and to gain experience through that. Regarding my colleagues at Siemens I want to thank besides Daniel Schall also Sebastian Meixner for the successful collaboration in the project of my Master's thesis as well as for the implementation of the frontend of the developed tool. Of course I also want to thank all my other colleagues at Siemens who supported me with my Master's thesis, especially all participants of the evaluation work shop of my thesis.

Kurzfassung

Die Veränderung und Verbesserung der Software Architektur von Applikationen in Cloud-Edge Umgebungen ist eine entscheidungsintensive Aufgabe bei der viele Constraints beachtet werden müssen. Auch bei der Migration von sogenannten legacy Applikationen in die Cloud müssen eine Vielzahl an technischen und nicht-technischen Constraints beachtet werden. Darüber hinaus unterscheiden sich Cloud Umgebungen von Edge Umgebungen anhand ihrer Qualitätseigenschaften (quality attributes). Eine Edge Plattform wird beispielsweise nicht das Maß an Skalierbarkeit unterstützen wie eine Cloud Umgebung. Daher ist Architecture Decision Support hilfreich um Entscheidungen zu verbessern.

Der Zweck dieser Arbeit ist es Software Architektur Entscheidungen, die insbesondere relevant für Software Architekten in Cloud-Edge Umgebungen sind, zu identifizieren. Basierend auf diesen Entscheidungen soll ein Workflow entwickelt werden, der dabei helfen soll die Komplexität der Entscheidungen zu reduzieren sowie Entscheidungen zu verbessern. Der Workflow wird unterstützt durch einen Algorithmus um Cloud Offerings anhand von Qualitätsanforderungen zu reihen. Auf diese Weise soll es einem Architekten ermöglicht werden das beste Offering auszuwählen anhand von Qualitätsanforderungen, beispielsweise als Ersatz für eine bereits existierende Komponente. Schließlich wird in dieser Arbeit noch die Software Architektur für ein Decision Support System präsentiert, das sowohl den Workflow als auch den Algorithmus zur Reihung von Cloud Offerings verwendet. Diese Architektur ist die Basis für die prototypische Implementierung, die in der experimentellen Evaluierung mit professionellen Software Architekten verwendet wird.

Abstract

Architecture refactoring of applications deployed into cloud-edge environments is a decision intensive task since multiple constraints must be considered. Similar technical and non-technical constraints must also be respected when migrating a legacy application into the cloud. Furthermore, cloud as well as edge environments differ in the quality attributes (QAs) they support. For example, an edge platform will not provide that scalability a cloud platform provides. So architecture decision support will help to improve decisions.

The purpose of this work is to identify architectural decisions relevant in cloud-edge environments from the point of view of a software architect and develop a taxonomy of decisions. Based on the taxonomy of decisions a decision support workflow will be created as main contribution of this work to reduce complexity and improved decisions. The workflow is supported by an algorithm for cloud offering ranking based on their QAs. This assists the architect in selecting the best cloud offering for a given component depending on its QA requirements. Finally, the architecture for a decision support system utilizing the workflow and the algorithm for offering ranking is presented. This is the basis for the prototypical implementation which is used for the experimental evaluation with professional software architects in industry.

Contents

CHAPTER 1

Introduction

Cloud computing has emerged from industry and it has essentially changed how software is developed and delivered. Most of its concepts as well as its benefits and risks are already well understood supported by academic and industrial publications [AFG+09, MG11, BGPCV12]. Major companies are exploiting the opportunities this paradigm provides to remain competitive and increase business agility. This also includes the migration of so-called legacy applications which are hosted on-premises into public cloud environments such as the Amazon Web Services (AWS). For that purpose the academic literature and cloud vendors proposed solutions to support this process [ABLS13]. Most of these solutions do not only focus on decisions relevant for software architects. They also consider decisions relevant for other stakeholders such as product managers deciding if a cloud migration should be done at all. In this decision context the benefits and risks of cloud migration or deciding who is responsible for which decision are relevant criteria for a decision maker. Such decisions are less relevant for software architects.

Besides cloud computing also *edge computing* has gained more and more importance in recent years [SCZ+16]. Since data is produced at the edge of the network, it would also be more efficient to process it there, and thus saving transfer overhead and costs. The popularity of the Internet of Things (IoT) also contributed positively to the importance of edge computing [SCZ+16]. With IoT a large quantity of data will be generated which cannot all be sent into the cloud for processing due to the huge network overhead this would produce. So applications must also be deployed at the edge of the network to consume this data. Migrating legacy applications into such cloud-edge environments would require *cloud-edge migration*.

Cloud and edge computing environments can be differentiated by the quality attributes (QAs) they fulfil. So an edge platform will not provide that scalability a cloud platform provides. This must, for example, be considered when migrating an application into a cloud-edge environment to distribute the components between cloud and edge according to their scalability requirements.

To support migrating applications into cloud environments *cloud migration patterns* were presented in academic literature (e.g. [JPCL15]) as well as in white papers and blog posts of cloud vendors [Mic17, Kel17, Ama15, Orb16]. Some of these patterns are only very high level and focus on the business aspects of cloud migration, and thus they are not sufficient for a software architect. Others are addressing software developers and development problems in more detail. Most of the works valuable for software architects have in common that mostly the same concepts are presented only with different names. As basis for this work regarding cloud migration patterns the work of Jamshidi et al. [JPCL15] was used since it is the only academic work found presenting vendor agnostic patterns focusing on software architecture design problems rather than on business aspects or development problems.

1.1 Problem Statement

Scope of this work is to consider the problem of cloud-edge migration from the point of view of a software architect. A cloud-edge migration is a decision intensive task since besides functional requirements also non functional requirements (NFRs) including technical and non-technical constraints must be considered. Therefore decision support is helpful to accelerate and improve the decisions made.

The literature review about software architecture decision making in industry by Dasanayake et al. [DMAO15] showed that there is a need for research and conceptual work in this field. Architecture decision making is a manual approach, however with a tool assisted workflow it can be done in a structured way, rather than an ad-hoc process. This helps to improve decisions by reducing the probability that decisions are overlooked. Furthermore, the number of errors made by a decision maker due to missing knowledge and ignored constraints can be reduced by using a knowledgebase. Such a knowledgebase, for example, includes information about the capabilities and constraints of a cloud platform. This knowledge can be used in the decision support workflow to improve decisions.

To summarize the current practices for software architecture decision making the following solutions could be identified when tacit knowledge is not enough or missing to solve a cloud-edge migration problem:

1. Ad-hoc web queries using a search engine such as Google to get the required information (e.g. documentation of AWS offerings).
2. A Q/A approach asking other people in a social network to support the architect during the migration process.
3. Consult experts familiar with technological foundations, best practices, limits, and constraints of a target platform.
4. Tool support which is mostly limited to Infrastructure as a Service (IaaS) migration such as database and virtual machine (VM) migration (e.g. migration to Microsoft Azure [Micb]) and does not cover Platform as a Service (PaaS) migration.

Furthermore, there is only informal knowledge sharing, for example by the means of meetings, but if people are distributed and do not know each other knowledge sharing becomes difficult. So other software architects having a similar problem must repeat the same ad-hoc process to get the required information. For example, other software architects will make similar queries using a search engine and dig through all the information returned to find the most suitable one. Moreover, they could ask similar questions in a social network again if they have overlooked that they were already answered.

An example for a architecture decision support problem in the context of cloud-edge computing would be the refactoring of an existing cloud-edge application already deployed to AWS. This application supports data collection, processing and storage capabilities for temperature data. On the edge side both data processing capabilities are already separated. However, on the cloud side this is not the case and must be improved. As it will be discussed in more detail in Section 6.2.2 there are multiple possibilities to fix this issue. Since there are many offerings which AWS provides and multiple of them would be possible solutions, it is not that easy to find the right one also when considering requirements regarding QA constraints.

1.2 Research Objectives

An architecture decision support process is developed in this work which supports a software architect in solving architecture decision support problems in cloud-edge environments such as the motivating example mentioned in the previous section. This main objective is divided into the following research questions which are motivated in the rest of this section:

- *RQ1*: How to support the software architect in migrating a legacy application into cloud-edge environments?
- *RQ2*: How to suggest a software architect the most suitable cloud offering to replace an existing component/ to move an existing component into the cloud?
- *RQ3*: How to consider quality attributes (such as scalability, performance) and how to get a trade-off among them?
- *RQ4*: How to support the software architect in selecting cloud migration patterns?

Many companies already have legacy applications deployed to existing, on-premises hardware. To leverage cloud scalability and cost saving opportunities such applications should be moved into cloud environments. Often these are large applications consisting of many components which are connected in a complex architecture. The migration implies multiple architectural decisions to make by a software architect which are often not that easy to do.

Furthermore, in industrial environments data is gathered from machines in the field which are located at the edge of the network. Due to the massive amount of data generated it is important to pre-process this data directly there where it was generated. So besides cloud environments also edge environments must be considered for software architecture decision making such as where to migrate a certain component (see *RQ1*). Since a migration process into cloud-edge environments should be done in an iterative manner to minimize complexity and risks, applications partly deployed into cloud-edge environments should be considered as well.

Considering a cloud platforms' capabilities which are mainly realized by cloud offerings we come to the second research objective (*RQ2*). Cloud platforms such as AWS provide multiple offerings supporting different capabilities. Offerings supporting the same capabilities, e.g. different data storage solutions, can be further differentiated by the QAs they fulfil. So it is not always that easy to select the right one which should fulfil a given set of QAs.

Regarding QAs often different, competing QAs must be considered (e.g. performance vs cost efficiency). A software architect must somehow find a trade-off among them to select the most suitable one. This work should find a solution to this problem (see *RQ3*).

Finally, also cloud migration patterns are helpful to support migrating an application into the cloud. However, since there are many of them it is not always that easy for a software architect to select the right pattern to solve a certain problem. So the architect should be supported somehow in this selection process (see *RQ4*).

1.3 Thesis Outline

This work is structured in the following way: Chapter 2 presents the methodology this Master's thesis follows. Chapter 3 summarizes the related work relevant for this work by starting with a general introduction about software architecture knowledge management and software architecture decision support. Then the topic of cloud-edge computing is discussed as well as the problem of migrating to cloud-edge environments. Chapter 4 introduces the requirements relevant for architecture decision support and important concepts of this work to describe the whole problem space. Chapter 5 presents the solution for architecture decision support proposed in this work, a tool assisted architecture decision support workflow. This chapter also presents the tool's architecture, an algorithm developed for offering ranking, and the prototypical implementation which was used for the evaluation presented in Chapter 6. Finally, the main findings and lessons learned of this work are summarized in a conclusion in Chapter 7 also giving directions for future work.

CHAPTER 2

Methodology

The purpose of this chapter is to present the methodology this work follows which is illustrated in Figure 2.1.

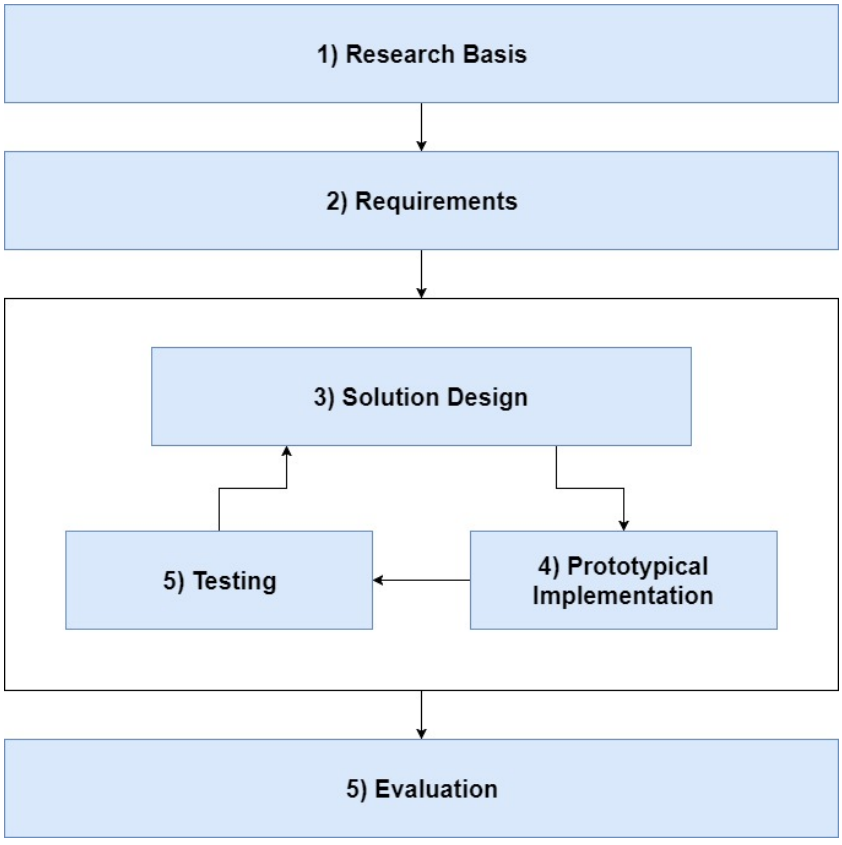


Figure 2.1: Methodology

At the beginning of the work the state-of-the-art in decision making of software architects was examined. Based on literature research it was found out that no structured approach exists and decision making is currently done in an ad-hoc fashion [DMAO15]. So a software architect's decisions are mainly based on his knowledge and when no knowledge about a certain technology exists research work must be done. For example, when a software architect wants to migrate his application into the AWS he must have a good understanding of the available cloud offerings and their properties to make good decisions. Since knowledge like that is needed by many software architects, other software architects who do not know about the platform must make the same research work again when they want to migrate their application. So it would be beneficial that this knowledge could be shared among the software architects since they could benefit from each other by sharing knowledge to make better decisions and to avoid making bad decisions. This first step is also illustrated in Figure 2.1.

Based on the research work done in the first step some requirements to support a software architect in making decisions were identified. A first requirement is that a structured process is needed rather than an ad-hoc process since this will help making better decisions. Furthermore, a structured process also decreases the probability of overlooking some decision which could be better than all the decisions currently considered.

In software architecture decision making also many constraints such as the QA constraints of a particular component must be considered. This increases the complexity of decision making and decision support will be indeed helpful to find the best solution when respecting a given set of QA constraints. People still need to do their research work and learn about new technologies including its benefits and drawbacks regarding their QAs, but decision support could help them bootstrapping their projects by pointing them to the right technologies to learn the right things to get their work done.

The literature review of step one also included research work about the *cloud migration* problem which could be one reason for architecture evolution. For the cloud migration problem *cloud migration patterns* were defined [JPCL15]. Since there are multiple cloud migration patterns which must be considered it is also not that easy for a software architect to find out which pattern to choose. So as a further research goal it was defined to answer the question how a software architect can be supported by selecting cloud migration patterns.

To support architecture decision support in cloud-edge environments also collecting and modelling of cloud offerings is needed since for each cloud provider there are multiple offerings usable for multiple purposes as well as fulfilling different QA constraints. For example, Amazon Glacier [Amaf] will be good for long time storage of data, but not that suitable for data which must be frequently accessed and where write performance matters. The definition of the requirements is illustrated as step 2 in Figure 2.1.

After all the requirements were defined it was decided to develop a structured workflow supporting architecture decision making in cloud-edge environments as core of this work (see step 3 of Figure 2.1). To support architecture decision making the decisions relevant

for this problem had to be identified. So using the identified decisions a taxonomy of decisions was created. These decisions are the basis for the workflow and are distributed among the steps of the workflow to distribute complexity as well as to simplify decision making.

Furthermore, since some decisions also required the examination of multiple cloud offerings suitable for a certain task, an algorithm for matching and ranking of cloud offerings was developed. For example, if a local MySQL database should be replaced by a Amazon cloud offering Amazon's Relational Database Service (RDS) [Amaj] would receive a much better score than for example Amazon Glacier. Based on the scores the algorithm returns, the software architect should be pointed to the right cloud offerings to solve his problem.

To persist information such as cloud offerings or other software engineering domain knowledge needed a model was required. For that purpose the Domain Entities Meta Model and Application Description meta models introduced by Plakidas et al. [PSZ18] were used to structure the information saved in the knowledgebase, however these are not the focus of the work.

A next step (step 4 in Figure 2.1) was the prototypical implementation of a decision support workflow as a web based tool fulfilling all the needed requirements. This tool should put the concepts identified in this work into practise so that they can be evaluated in a user experiment with professional software architects in industry (step 6 in Figure 2.1). An important point was also the testing of the prototypical implementation so that the evaluation could return useful results (step 5 in Figure 2.1). As it can be seen in Figure 2.1 the steps 3 to 5 were executed multiple times in an iterative manner.

CHAPTER 3

Related Work

This chapter examines the related work of the fields this work is placed in as well as the main terminology related to this thesis is introduced for a common understanding. It starts with related work about *software architecture knowledge management* to deal with the approaches in literature enabling reusable knowledge and knowledge sharing. Then the topic of *software architecture decision support* is treated which is strongly related to software architecture knowledge management since a solid domain knowledge is needed to give decision support in this field. Finally, the topic of *cloud migration* is discussed also referring to edge-based migration. To motivate why cloud migration should be done at all *cloud computing* is explained focusing on its service models which are all relevant for this work.

3.1 Software Architecture Knowledge Management

Capilla et al. [CJT+16] reviewed the architecture knowledge management practises from 2004 to 2014. They considered solutions found in literature as well as how it is done by professionals in industry. For the solutions found in literature they identified three generations, namely a first generation (2004-2006) were simple tools were developed independently of each other such as the works of Tyree et al. [TA05], Capilla et al. [CNPDn06], and Tang et al. [TJH07]. These tools provided simple functionality for capturing, managing and documenting of design decisions and architectural knowledge. The first generation of tools are the basis for the second generation (2007-2010) of tools which focused on the sharing aspects between the relevant stakeholders. Examples for such second generation tools are the works of Farenhorst et al. [FLvV07], Schuster et al. [SZP07], Burge and Brown [BB08], Jansen et al. [JdVAvV08], and Capilla et al. [CNC08]. Furthermore, they also provided basic personalization mechanisms to include stakeholder-specific content. They are mostly based on web and wiki technologies to provide means for capturing and organizing architectural knowledge as well as to enable collaboration between users by

using collaborative and communication facilities. Finally, the third generation (2011-2014) of tools (e.g. the works of Nowak et al. [NP13], Lytra et al. [LTZ13], and Manteuffel et al. [MTK+14]) focus on reuse of architectural knowledge as well as (fuzzy) decision making also supporting uncertainty. This includes tools enabling the creation of reusable architecture design decision models as well as generating questionnaires for making decisions based on new questions. All the developed tools have in common that they are based on some meta model used for decision making.

The results of the interviews done with professionals from industry showed that requirements are mostly captured in Word files, whereas architecture views are made using PowerPoint/Visio [CJT+16]. These views often use a loose interpretation of UML semantics meaning that practitioners in industry do not adhere to its standards. Further results of the interview indicate that there is no systematic architecture documentation approach, although the participants are aware of the importance of capturing architecture knowledge. The participants who have already tried out some knowledge management tools identified issues in the maturity as well as the compatibility with other tools preventing the usage of knowledge management tools. Furthermore, they mentioned that clarity and familiarity of the underlying concepts as well as simplicity are important factors of such a tool to succeed.

Weinreich et al. [WG14] did a systematic literature review about software architecture knowledge management technologies including works from 2003 to 2013. Compared to the work of Capilla et al. [CJT+16] their methodology is much clearer, because it is clearly stated how the works considered are selected. Weinreich et al. state that most of the approaches are based on a template by Three et al. [TA05] using similar core terminology such as decisions, their rationale, and relationships between decisions. However, there is no common meta model used and no agreement on the overall terminology. Main activities for software architecture knowledge management are knowledge capturing, maintaining, sharing, using, and reusing. Under architecture knowledge capturing the authors understand the documentation of design decisions to make this knowledge explicit. Knowledge maintaining means to keep the knowledge up-to-date and knowledge sharing means to share the knowledge among different stakeholders and in different contexts. For knowledge using they identified main use cases such as architecture analysis and architecture review. Architecture knowledge reusing means to reuse knowledge in other projects.

Using qualitative analysis techniques Weinreich et al. [WG14] considered seven approaches as relevant. They concluded their work by confirming that most approaches enable capturing knowledge which supports the understanding of architecture design decisions since they are explicitly documented as well as enables architecture evaluation. Furthermore, the authors identified works enabling the reuse of knowledge, supporting different stakeholders by providing different views as well as including decision making capabilities to support an architect's decisions. Weinreich et al. could not find core evidence for architecture knowledge maintenance, although Capilla et al. [CJT+16] mentioned multiple solutions that support architecture knowledge management which includes architecture

knowledge maintenance. Both works agree on the evidence that there are only few solutions supporting knowledge reuse. Furthermore, both indicate that decision support solutions are only appearing in recent works. For example, Capilla et al. [CJT+16] state that there is only one solution supporting decision making with uncertainty support.

3.2 Software Architecture Decision Support

Dasanayake et al. [DMAO15] examined the decision making practices and challenges in industry. The information, they gathered based on a case study conducted in three different companies in Europe, showed that decision making is mainly based on tacit knowledge and no structured process exists. The most frequently used knowledge management technique are design documents and ad-hoc processes based on the experience of software architects. This is quite error prone since the design documents may be incomplete or the software architect may lack knowledge about the used technologies which could lead to the problem that a better solution is overlooked. The authors found out by interviewing professionals from industry that a knowledge process must be lightweight to be accepted since tools which currently exist were mostly too complicated to use. Also the compatibility of their data formats with other tools used is an issue.

The rest of this section presents some solutions for software architecture decision support found in literature.

Already in 2003, Svahnberg et al. [SWLM03] presented a method for ranking different software architecture candidates based on QA constraints with the goal to find out which architecture candidate best suits the quality requirements. For the first step of their method, the generation of architecture candidates and QA constraints, they refer to methods from requirements engineering. For the ranking an Analytic Hierarchy Process (AHP) algorithm [Saa90] is used, however they state that there method is not specific to this AHP algorithm, but any other ranking algorithm can be used. Compared to this work, the method proposed by Svahnberg et al. is not supported by some knowledge model. It is a manual method for ranking architecture candidates based on quality requirements. Furthermore, it also does not target cloud migration but software architecture decision support in general.

Wehling et al. [WWSS17] present a method to identify and reduce IT complexity. This method should help experts such as software architects to remove artifacts which are not required by using derived variability information. With variability the authors mean artifacts that only differ in small points, e.g. some implementation details, but their purpose and the problem they solve is the same. Their proposed iterative decision process deals with application architectures, but the authors also claim that there approach can be applied to software architectures including artifacts such as modules and microservices. By removing redundant artifacts their negative impacts such as higher costs, decreased adaptability, higher effort for maintaining and evolving systems should be reduced. Similar to the work of Svahnberg et al. [SWLM03] this is a manual approach without a

knowledge model this solution is based on. Furthermore, by reducing IT complexity it targets a more specific purpose than software architecture decision support.

Telschig et al. [TSS+16] propose a decision support system based on patterns for software ecosystems. Similar to software patterns, software ecosystem patterns consider recurring, high level problems in software ecosystems. Besides the decision support system these patterns are also meant to be used standalone. Compared to this work the work of Telschig et al. does not target cloud-edge environments, but software ecosystems in general.

Carlson et al. [CPP16] present a model in their work with the goal to explicitly model the context of a decision case. This should enable the application of the same decisions from past problems to new problems with similar context. Compared to the other works mentioned in this section this work more refers to decisions not only considering the software architecture but also organizational and business aspects.

A decision support ecosystem approach is presented in the work of Axelsson et al. [AFC+17]. Using an ecosystem approach should help sharing of knowledge such as knowledge about past decisions while addressing integrity constraints and intellectual property issues. Using this decision support ecosystem with a common knowledgebase should help to make decisions based on facts rather than on incomplete knowledge or just guessing.

As key requirements the authors identified four key stakeholders of the system, namely the decision makers, contributors, who can for example implement their own decision processes and estimation models, system administrators, as well as process analysts who analyse the aggregated data or further improve the tool. Furthermore, they identified non functional concerns for such a decision support system based on ISO 25010 [ISO] such as *usability in use*, *flexibility in use*, or *security*.

Data sources for the decision system would be a company's previous decisions but also scientific literature, or websites. Reasoning is done using analogy based reasoning which means that information from similar cases is incorporated to give recommendations. To support analogy based reasoning a taxonomy of decision case data, called GRADE, was developed by Axelsson et al. [AFC+17]. GRADE is the abbreviation for Goals, Roles, Assets, Decisions, and Environments. With Goals the key objectives of a decision are meant, Roles are the stakeholder involved, Assets are the options available for decision making, Decisions are methods used to make decisions, and Environment is the context of a decision. Besides the GRADE taxonomy also a context model is used describing the environment perspective of GRADE by incorporating information about aspects such as organization or business domain.

The authors also presented the architecture of their decision support system which is based on micro services. Similar to this work a graph database is used as knowledgebase. Compared to this work, the work of Axelsson et al. does not focus on software architects as core users of their system, but they mention decision makers in a more general sense. They introduced additional key stakeholders such as system administrators, contributors, or process analysts. Furthermore, their work does not specially target

cloud-edge environments and no decision support algorithm such as an AHP algorithm is proposed. It seems that such special aspects are delegated to contributors of the tool who can model their own decision support process.

3.3 Cloud-Edge Computing and Cloud-Edge Migration

This section starts by giving an overview of Cloud Computing and its concepts. Then the problem of cloud migration is discussed mentioning some solutions found in the literature.

3.3.1 Cloud-Edge Computing

Cloud computing is a paradigm enabling on-demand network access to shared pools of configurable computing resources such as hardware capacities, storage, and software services [MG11]. Compared to provisioning these resources on-premises delegating these tasks to an external party provides a lot of benefits such as reduced costs, improved flexibility and business agility. In the rest of this section the service models and deployment models of cloud computing as defined by the National Institute of Standards and Technology (NIST) [MG11] are discussed. Furthermore, the concept of edge computing is discussed.

Service Models

- *Software as a Service (SaaS)*: The cloud vendor provides on-demand services running in the cloud to his customers. The services are either accessible via a thin client interface such as a web browser or a program interfaces such as a REST API. Besides specific application configuration changes the user has no access to the underlying infrastructure. Examples are business software such as enterprise resource planning but also managed database solutions (e.g. Amazon's Relational Database Service [Amaj] or messaging middleware (e.g. Amazon's Simple Queuing Service [Amak].
- *Platform as a Service (PaaS)*: Whole application runtime environments including support for certain programming languages, libraries, services and tools are provided to a customer. The customer has no control over the underlying cloud infrastructure including hardware capacities, network, storage, or operating systems. He only has control over his deployed applications as well as configuration settings for the application hosting environment. PaaS examples are SAP HANA Cloud Platform [SAP] or Amazon's Elastic Beanstalk [Amaa].
- *Infrastructure as a Service (IaaS)*: Virtualized IT infrastructure, including processing, storage, networks, and other fundamental computing resources, is provided to the customer. So he can deploy and run his applications including operating systems. However, he has no control over the underlying hardware, e.g. where his applications are run. An example for IaaS is Amazon's Elastic Compute Cloud (EC2) [Amad].

Besides the above mentioned well known service models also other service models exist such as *Function as a Service (FaaS)* (e.g. AWS Lambda [Amai]) or *Container as a Service (CaaS)* (e.g. Amazon's Elastic Container Service (ECS) [Amae]). More general the term *Anything as a Service (XaaS)* can be found in literature [GC12].

Deployment Models

- *Public Cloud*: The cloud vendor provides his services open for public use which is mostly shared by multiple customers. It is hosted on the premises of the cloud provider who may be a business, academic, or governmental organization, or some combination of them.
- *Private Cloud*: Cloud infrastructure is exclusively provisioned by a single organization for their customers such as business units. Besides the organization itself also a third party or a combination of both may own, manage and operate the infrastructure. This can be done on or off premise.
- *Community Cloud*: Cloud infrastructure is provided exclusively to an organizational community sharing specific concerns such as mission, policy, or security requirements. Some organization in the community, a third party, or a combination of them may own, manage and operate the cloud infrastructure which may exist on or off premise.
- *Hybrid Cloud*: A hybrid cloud is a combination of two or more distinct cloud infrastructures (private, community, or public) connected by standardized or proprietary technology enabling data and application portability.

Edge Computing

In contrast to cloud computing, in edge computing the computing nodes are running de-central mostly only having a limited amount of computing resources such as CPU performance or RAM [SD16]. The computing takes place near where the processed data was produced. In this way the bandwidth required for transferring data into the cloud is reduced because it is already processed at the edge of the network.

Especially the popularity of the IoT increases the requirement for edge computing. With IoT there will be sensor networks consisting of billions of sensors [SCZ+16] all producing data. It is not feasible to send all this data into the cloud due to bandwidth limits, thus the speed of transport becoming the bottleneck. So applications must also run at the edge of the network consuming this data.

Furthermore, devices running at the edge of the network are also becoming more powerful, enabling for example the use of container technologies similar as in cloud environments [IGK+15]. This also reduces the effort for developers since they can test their solutions locally. Nevertheless, they must keep resource constraints in mind.

3.3.2 Cloud-Edge Migration

Cloud migration describes the process of migration data and applications from one hosting environment into another hosting environment [JAP13]. An example would be migrating from a private, on-premises cloud into a public cloud environment. Often there are incompatibilities in technologies used which must be resolved. This means refactoring of the application's architecture may be required. Since often multiple valid solutions exists all having their benefits and drawbacks, cloud migration can be considered as an not that easy to solve problem. Since besides cloud environments an application can also be migrated into an edge environment, cloud migration is also more generally referred as *cloud-edge migration* in this work.

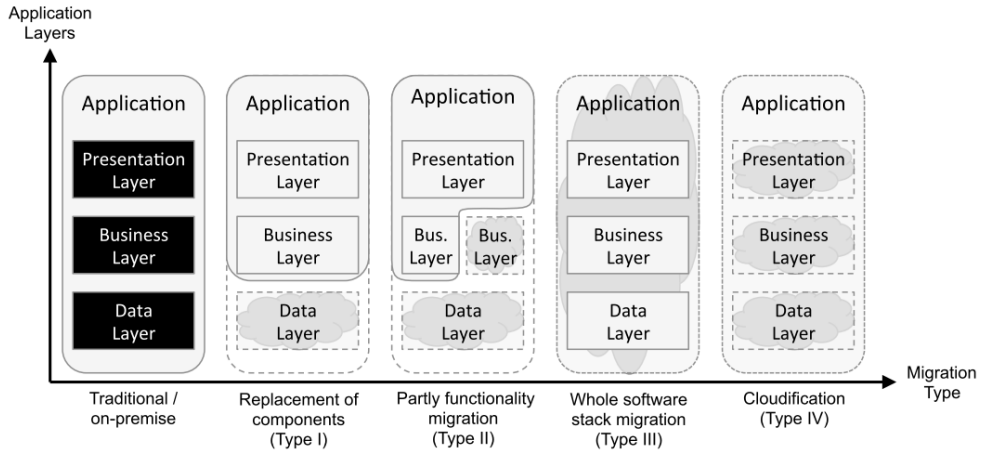


Figure 3.1: Cloud Migration Types [ABLS13]

Andrikopoulos et al. [ABLS13] present four different cloud migration types in their work, namely *replacement of components (type I)*, *partial functionality migration (type II)*, *whole software stack migration (type III)*, and *cloudify the application (type IV)*. All the types can be seen in Figure 3.1. They assume that initially all components are hosted on-premises. Their description also assumes a *three-tier-architecture* consisting of *presentation layer*, *business layer*, and *data layer*. With replacement of components they mean replacing components with cloud offerings, which is similar to the REPLACE action which will be discussed in Section 4.1.1, and is limited to the data layer. Type I migration also includes the migration and/or adaptation of business logic for accessing the data store as well as configuration rewriting. In case of partial functionality migration whole application functionalities are migrated to the cloud involving several architectural components. Whole software stack migration describes the classical form of migration where a whole application is run in a VM in the cloud. Finally, with cloudify the application Andrikopoulos et al. mean a complete migration of the whole application by implementing the application functionality as composition of services. In contrast to that, Jamshidi et al. [JPCL15] mean with *cloudification* in their work about *cloud*

migration patterns a completely different thing, similar to type I migration presented by Andrikopoulos et al. [ABLS13].

Compared to the work of Andrikopoulos et al. [ABLS13], this work does not assume a three-tier-architecture for the migration problem. It is simply looked at the problem component wise, meaning that also a component belonging to the business logic could be replaced by a cloud offering if their *capabilities* match. The only constraint for a component is that it must be independently deployable from the rest of the application and connected to other components via a network interface. So a monolithic application would consist of a single component and VM migration would be possible, referred by Andrikopoulos et al. as *whole software stack migration*. Furthermore, the SPLIT-UP action which will be introduced in Section 4.1.1 could be used which means splitting up the application's functionality into different, independent services to leverage the cloud platform's elasticity and scalability opportunities.

In 2013, Jamshidi et al. [JAP13] did a systematic literature review about cloud migration research. They identified cost reduction and increased business agility as common drivers for cloud migration. They compared cloud migration with migration to Service Oriented Architecture (SOA) also comparing their work with secondary studies about SOA migration. Besides cost saving and increased business agility also application scalability, efficient utilization of resources as well as flexibility are primary motivation factors for cloud migration. They grouped their findings about cloud migration literature using the cloud migration types definition provided by Andrikopoulos et al. [ABLS13]. They could not find any work using *type I* migration. The most solutions target *type IV* migration (43%), followed by *type II* migration (34%) and *type III* migration (21%). So it can be seen that the migration of the whole application into the cloud is gaining increased attention in literature, whereas the migration of whole software stacks, e.g. using VM migration, has decreasing importance. This is also against the assumption made by Jamshidi et al. [JAP13] that *type III* migration is mostly used since it is the easiest to do.

3.3.3 Cloud-Edge Migration Decision Support

This section discusses other approaches to support the decision making in cloud-edge migration. An overview of these approaches is given in Table 3.1. The search was mainly done using Google scholar and search terms such as *architecture decision support cloud edge*, or *cloud edge migration decision support*.

In general, it can be said that there are multiple approaches using an AHP algorithm, but nearly all of them do not use such an algorithm for cloud offering ranking as it is done in this work. These approaches mostly only consider a limited set of QAs used as criteria for the AHP algorithm and the objects, which are ranked, are decision alternatives. No approach could be found considering such a detailed set of QAs as it is done in this work.

Some of the solutions found in literature also do not specially target software architects, but they are targeting companies to support them with cloud migration decision making.

Therefore also more general decisions can be found such as deciding who is responsible for which task required to do the cloud migration, but software architecture decision making is not considered in that detail as in this work. Furthermore, no approach could be found who considers besides cloud environments also edge environments.

Table 3.1: Current approaches in decision support for cloud migration

(Name), Author(s), Title	**Year**	**Reference**
"SMICloud", S. K. Garg, S. Versteeg, and R. Buyya, "SMICloud: A Framework for Comparing and Ranking Cloud Services"	2011	[GVB11]
V. Andrikopoulos, Z. Song, F.Leymann, "Supporting the Migration of Applications to the Cloud through a Decision Support System"	2013	[ASL13a]
A. Juan-Verdejo, H. Baars, "Decision Support for Partially Moving Applications to the Cloud – The Example of Business Intelligence"	2013	[JVB13]
"InCLOUDer", A. Juan-Verdejo, S. Zschaler, B. Surajbali, H. Baars, H. Kemper, "InCLOUDer: A Formalised Decision Support Modelling Approach to Migrate Applications to Cloud Environments"	2014	[JVZS+14]
"cDSF", V. Andrikopoulos, S. Strauch, F. Leymann, "Decision Support for Application Migration to the Cloud: Challenges and Vision"	2013/ 2014	[ASL13b] [Dar14]
A. Alhammadi, C. Stanier, A. Eardley, "A Knowledge Based Decision Making Tool to Support Cloud Migration Decision Making"	2015	[ASE15]
"CloudGenius", M. Menzel, R. Ranjan, L. Wang, S. U. Khan, J. Chen, "CloudGenius: A Hybrid Decision Support Method for Automating the Migration of Web Application Clusters to Public Clouds"	2015	[MRW+15] [MR12]
B.R. Wilson, L. Hirsch, B. Khazaei, "Cloud adoption decision support for SMEs Using Analytical Hierarchy Process (AHP)"	2016	[WKH16]

SMICloud

Garg et al. [GVB11] presented SMICloud, a framework for comparing and ranking of cloud services which is based on the Service Measurement Index (SMI) developed by the Cloud Service Measurement Index Consortium (CSMIC). Such a measurement index can be used by customers to compare cloud services. The CSMIC already defined the QAs which are used for comparing cloud services, namely accountability, agility, assurance of service, cost, performance, security, privacy, and usability. Garg et al. defined metrics for all these QAs which also includes how these QAs can be measured. For the purpose of ranking cloud services based on their QAs the authors proposed an AHP algorithm which is their main contribution.

In contrast to the work of Garg et al., the focus of this work lies on software architecture decision support in cloud-edge environments with the definition of a workflow supporting decision making. The contribution of this work also includes an AHP algorithm, but this algorithm considers much more QAs compared to the solution of Garg et al. Furthermore, the definition and elaboration of metrics for QAs is out of scope of this work. The design and the runtime properties of the proposed algorithm are more the focus of this work. However, the metrics defined by Garg et al. considering runtime measurements could be used by the algorithm proposed in this work, for example by mapping such metrics onto the simple metrics defined as part of this work.

Supporting the Migration of Applications to the Cloud through a Decision Support System

Andrikopoulos et al. [ASL13a] present a decision support system for application migration into the cloud. It mainly consists of two components, namely a *cost calculator* and a *offerings matcher*. Similar to the solution proposed in this work, the solution of Andrikopoulos et al. consists of a three-tier-architecture. However, QAs are not considered in that detail as it is done in this work. Regarding QAs it is mainly focused on the costs of cloud services by considering the different cost models the cloud vendors provide. Matching is not done based on *capabilities* as it is done in this work, but it is done based on *parameters* and *usage patterns* given by the user of the decision support system. Examples for parameters are the number of CPUs, or a license required, and examples for usage patterns are for example the scalability of a storage meaning that the needed storage can be increased or decreased on-demand.

Decision Support for Partially Moving Applications to the Cloud – The Example of Business Intelligence

Juan-Verdejo et al. [JVB13] consider *hybrid* deployments of Business Intelligence (BI) solutions in their work about application migration into the cloud. They focus on hybrid deployments because BI solutions have requirements regarding security, privacy, and performance which make a cloud deployment of the whole application impossible. Part of their cloud migration framework is the generation of migration alternatives based on a

model for component placement. How this is done in detail is not explained in their work. A ranking of the migration alternatives is done using an AHP algorithm with the goal to maximize monetary benefits and respecting functional and non functional requirements.

Compared to this work, the work of Juan-Verdejo et al. [JVB13] does not consider *multi-cloud* deployments, but only focuses on hybrid cloud deployments. It especially focuses on BI solutions, whereas the solution proposed in this work can be applied to any internet-enabled solutions. The AHP algorithm is only briefly discussed, but it seems to follow the method proposed by Satty [Saa90] using the rating scale 1 to 9, from *equal importance* to *extremely more important*, which requires the pairwise comparison of the alternatives, e.g. by some expert. Furthermore, the AHP algorithm is used for ranking migration alternatives rather than cloud offerings as it is done in this work.

InCLOUDer

In 2014, Juan-Verdejo et al. [JVZS+14] present a decision support system targeting organizations which considers formal parameters affecting the cloud migration as well as metrics for objective and subjective criteria. This work considers the same QAs as the SMICloud solution [GVB11] discussed above by mapping them onto their formal definition of the cloud migration problem. The proposed decision support system builds on top of an AHP algorithm for ranking migration alternatives. One alternative simply consists of the assignment of an application's components onto cloud as well as on-premises platforms.

In comparison to the work of Juan-Verdejo et al. [JVZS+14], this work considers much more QAs as well as the AHP algorithm is not its main contribution, but an architecture decision support workflow targeting cloud-edge environments. Furthermore, the decision support system proposed by Juan-Verdejo et al. targets organization in general, not specially software architects as this work does.

Decision Support for Application Migration to the Cloud

In 2013, Andrikopoulos et al. [ASL13b] presented a decision support framework for cloud migration. Their work also included the identification of challenges such as how to distribute the application while respecting security, privacy, or legal requirements. Further challenges are how to decide about the elasticity requirements for a given application, which cloud service model (IaaS, PaaS, or SaaS) fits best the multi-tenancy needs for some application, or how to respect Quality of Service regarding service availability and performance variability an application needs. Furthermore, they differentiate between the migration and the operation costs. A main part of the migration costs are the effort required for refactoring an application to prepare it for cloud deployment. For the operation costs the authors consider besides the costs of the services itself also licensing costs. Finally, a last challenge discussed is security and data confidentiality.

Their proposed decision support framework is based on two concepts, decisions and tasks. Decisions would be for example how to distribute an application among the cloud

and a local data centre or how to select a cloud service provider. Examples of tasks would be work load profiling of an application to migrate to expect its performance required, compliance assurance, identification of security concerns, cost analysis, effort estimation for adapting of an application to deploy into the cloud to only mention some of them. Darsow [Dar14] elaborated and refined the concepts presented in the work of Andrikopoulos et al. [ASL13b].

The works of Andrikopoulos et al. and Darsow provide decisions relevant for companies who want to migrate their applications into the cloud. So there is not that special focus on software architects who need to refactor their applications for a cloud deployment as it is in this work. Furthermore, this work considers software architecture decision support in cloud-edge environments which means that not only doing a cloud migration is supported, but also the refactoring of applications already deployed into cloud environments, and edge environments are considered as well.

A Knowledge Based Decision Making Tool to Support Cloud Migration Decision Making

Alhammadi et al. [ASE15] developed a model to support decision making for cloud migration. Their approach combines *case based reasoning* (CBR), which incorporates information gathered from past decisions, and the AHP. Their CBR approach involves attributes, which were identified by doing literature research and fieldwork, such as enterprise size, industry sector, enterprise status (e.g. for a start-up it is much easier to adapt cloud computing due to not relying on legacy applications), IT maturity level, and technology diffusion. The purpose of these attributes is to classify the data used for decision making. Their alternatives used for the AHP seem to be quite high level such as the decision between doing a cloud deployment at all or staying on-premises. So it can be concluded that this approach also targets organizations in general as it is the case in most works found during the literature review.

CloudGenius

Menzel et al. presented with CloudGenius [MR12, MRW+15] a hybrid decision support method targeting IaaS migration of web application servers. Their approach supports the selection of cloud VM image and cloud service combinations using a hybrid decision support approach using an AHP based approach combined with a genetic algorithm. The best VM image is selected based on its hourly price and the image's popularity, e.g. how often it is used. For the best service also its hourly price is taken into account, but also the network latency as well as quality regarding performance, uptime, as well as service popularity.

Compared to this work, the work of Menzel et al. only focuses on IaaS migration, which means that PaaS and SaaS offerings are not considered. Furthermore, the focus of their work is more on the algorithmic part and its formal and experimental evaluation focusing on the formal model of their approach. The decision support process of their work is

only briefly explained and supported by a figure using the Business Process Model and Notation (BPMN).

Cloud adoption decision support for SMEs Using Analytical Hierarchy Process (AHP)

Wilson et al. [WKH16] developed an AHP based approach for cloud migration decision making with the focus on small and medium enterprises (SME) in Tamil Nadu (India). They defined a taxonomy of criteria which is similar to the taxonomy of QAs for architecture decision support in cloud-edge environments presented in this work. They do not rank cloud offerings using their AHP approach as it is done in this work, but they just decide among five alternatives they identified as relevant for cloud migration such as virtualisation/ server consolidation, SaaS implementation, PaaS implementation, IaaS implementation, and server co-location. They did a case study with two participants, SMEs belonging to the IT sector of Tamil Nadu, to test their approach in practise.

In summary, it can be said that their used alternatives are quite high level and not specific to a certain cloud platform. However, using a SaaS implementation, which is one alternative of their approach, is quite specific to some cloud platform. For example, there could be no SaaS offering supporting the required capabilities which would make it impossible to use a SaaS implementation although their approach suggested it to do so.

3.3.4 Comparison of Cloud-Edge Migration Decision Support Solutions

The tables 3.2 and 3.3 show a comparison regarding features of the solutions presented in the previous section with this work. Some of the software architecture decision support solutions introduced in Section 3.2 which have at least one of the features examined are also considered (see last column of the tables 3.2 and 3.3). The features are clustered into five categories to give a bigger picture of all the features relevant for this work.

When looking at Table 3.2 there are many solutions considering QAs of some application or component (1.1). Moreover, also some solutions have developed a software architecture for a decision support system (1.3). However, no other solution could be identified which specially targets software architects (1.2). Many other solutions consider multiple stakeholders and some of them also focus on business aspects such as migration risks or costs.

Regarding cloud-edge computing only a single other solution could be identified which provides a solution for cloud offering matching (2.1), but the matching is done in a different way. Only two other solutions could be found which enable cloud offering ranking (2.2), but compared to this work they mostly only consider a limited set of QAs. There are many solutions considering IaaS, PaaS, and SaaS migration (2.4) and only one solution was found which just focuses on IaaS migration (2.5) without considering the other two service models. Finally, there could be no other solution found which also considers migrating to edge environments (2.3).

In Table 3.3 it can be seen that there was no other solution found which has developed a decision support workflow (3.1) as it was done in this work. Some of the works also consider historical decisions (3.2) and use techniques such as case based reasoning or analogy based reasoning to apply them onto new decision cases. Considering historical decisions is not the scope of this work since a fixed set of decisions were identified and grouped using the taxonomy of decisions which will be presented in Section 4.1.1. Only one other tool assisted solution could be found (3.3). Some more solutions could be found which present manual approaches (3.4). Note that there are also approaches which are neither considered as manual or tool assisted since they do not present a decision support system. Furthermore, they are only focusing on algorithms for decision alternative ranking or presenting conceptual models.

Regarding ranking there are many works which developed an AHP based solution (4.1). Moreover, many solutions could be identified which apply other kinds of ranking (4.2) such as ranking of decision alternatives or ranking of architecture candidates.

Finally, there are some solutions which developed taxonomies (5.1) such as a taxonomy of decisions relevant in a particular context. Some solutions are also using a knowledgebase as basis for their decision support system enabling knowledge sharing (5.2).

Table 3.2: Comparison of cloud-edge migration decision support solutions (1)

Feature		**ID**	**This work**	SMICloud [GVB11]	Andrikopoulos et al. [ASL13a]	Juan-Verdejo et al. [JVB13]	InCLOUDer [JVZS+14]	cDSF [ASL13b, Dar14]	Alhammadi et al. [ASE15]	CloudGenius [MR12, MRW+15]	Wilson et al. [WKH16]	Svahnberg et al. [SWLM03]	Carlson et al. [CPP16]	Axelsson et al. [AFC+17]
Software Architecture	Quality Attributes	1.1	✓	✓		✓	✓	✓			✓	✓		
	Targets Software Architects	1.2	✓											
	Software Architecture of a Decision Support System	1.3	✓	✓	✓			✓						✓
Cloud-Edge	Offering Matching	2.1	✓		✓									
	Offering Ranking	2.2	✓	✓	✓									
	Edge Environments	2.3	✓											
	IaaS, PaaS, SaaS Migration	2.4	✓	✓	✓	✓	✓	✓			✓			
	IaaS Migration only	2.5								✓				

Table 3.3: Comparison of cloud-edge migration decision support solutions (2)

Feature		ID	**This work**	SMICloud [GVB11]	Andrikopoulos et al. [ASL13a]	Juan-Verdejo et al. [JVB13]	InCLOUDer [JVZS+14]	cDSF [ASL13b, Dar14]	Alhammadi et al. [ASE15]	CloudGenius [MRJ2, MRW+15]	Wilson et al. [WKH16]	Svahnberg et al. [SWLM03]	Carlson et al. [CPP16]	Axelsson et al. [AFC+17]
Workflow	Decision Support Workflow	3.1	✓											
	Historical Decisions	3.2							✓				✓	✓
	Tool Assisted	3.3	✓		✓									
	Manual	3.4				✓	✓	✓			✓	✓		
Ranking	AHP Based	4.1	✓	✓		✓	✓		✓	✓	✓	✓		
	Other Ranking	4.2				✓	✓		✓	✓	✓	✓		
Knowledgebase	Taxonomy/ies	5.1	✓					✓	✓		✓			✓
	Use of a Knowledgebase	5.2	✓		✓				✓					✓

CHAPTER 4

Architecture Decision Support Requirements

This chapter introduces the requirements relevant for architecture decision support (Section 4.1) as well as the concepts needed to understand this work are explained (Section 4.2). Furthermore, it is elaborated why all these concepts are needed.

4.1 Requirements

We structure the requirements into four groups:

- A *taxonomy of decisions* containing decisions for architecture decision support in a cloud-edge environments must be identified. These decisions will be the base for a structured decision support process.
- Architecture decision making must respect many *quality attributes* (QAs) and constraints for each component which is considered. Therefore there is a need for organizing them in a structured way, e.g. as a taxonomy of QAs. Organizing them in a structured way also helps a decision maker by identifying the right QA to consider since so he may only have to look for the superordinate concept. Using some well known standard for these concepts will increase the probability that an architecture is already familiar with them. Furthermore, using some well known standard helps to get a complete picture of all the decisions required and the risk of missing out some important QA in this taxonomy is minimized.
- *Cloud migration patterns* are another important concept for cloud-edge environments since besides migrating legacy applications into other cloud environments they are also helpful for refactoring applications already deployed onto some cloud

platform, e.g. when changing a legacy application's architecture to a micro services architecture. Cloud migration patterns may be helpful for edge platforms as well when generalizing them to cloud-edge platforms, especially in the refactoring example mentioned before.

- Finally, architecture decision support also requires the *collection and modelling of cloud offerings*. To model the offerings with their capabilities and constraints a good understanding of the cloud platform and its offerings is needed. This must be done for each cloud platform and is an important precondition for the tool assisted decision support workflow presented in this work.

4.1.1 Taxonomy of Decisions

The following section elaborates on the decisions needed to consider when doing software architecture refactoring in a cloud-edge environment. These decisions also apply when the application is already partly deployed into a cloud environment, but in a next step further components should be moved into another environment. The decisions presented in this section are also relevant for edge platforms when more generally speaking from cloud-edge platforms. The taxonomy of all decisions can be seen in Figure 4.1.

Selection of Deployment Scenario

A first decision a software architect must make is selecting a deployment scenario. Possible deployment scenarios would be deploying all components to a single cloud provider (*cloud* deployment scenario as referred in Figure 4.1). Due to different constraints such as hardware constraints (e.g. GPU computing support is required) or legal constraints (e.g. data privacy is of importance) deploying all components into the cloud will often not be possible. Therefore a hybrid deployment could be chosen, meaning that some of the components are deployed into the cloud and some are left on-premises (*hybrid-cloud* as referred in Figure 4.1). A third option would be to distribute the components among multiple cloud providers to get a multi-cloud deployment (*multi-cloud*, and *hybrid-multi-cloud*). *Hybrid-multi-cloud* deployment is similar to hybrid-cloud also hosting some components on-premises, but for this deployment scenario two or more cloud providers may be selected. Reasons for such deployment scenarios would be the lack of a feature required by some application, e.g. a lack of GPU computing support as previously discussed. So besides hosting this component on-premises also an additional cloud platform could be used supporting this feature. A reason for that would be if acquiring all the hardware required to host the component on-premises is too expensive. Furthermore, a multi-cloud deployment may also reduce the dependency to a single cloud provider. For example, if one cloud provider is not reachable any more due to some software bug availability of a company's critical business services would be still guaranteed due to deployment to multiple cloud providers. However, a downside of such a solution are increased costs as well as increased complexity for the application itself. For example the application's architecture must support redundant deployment (e.g. via stateless design).

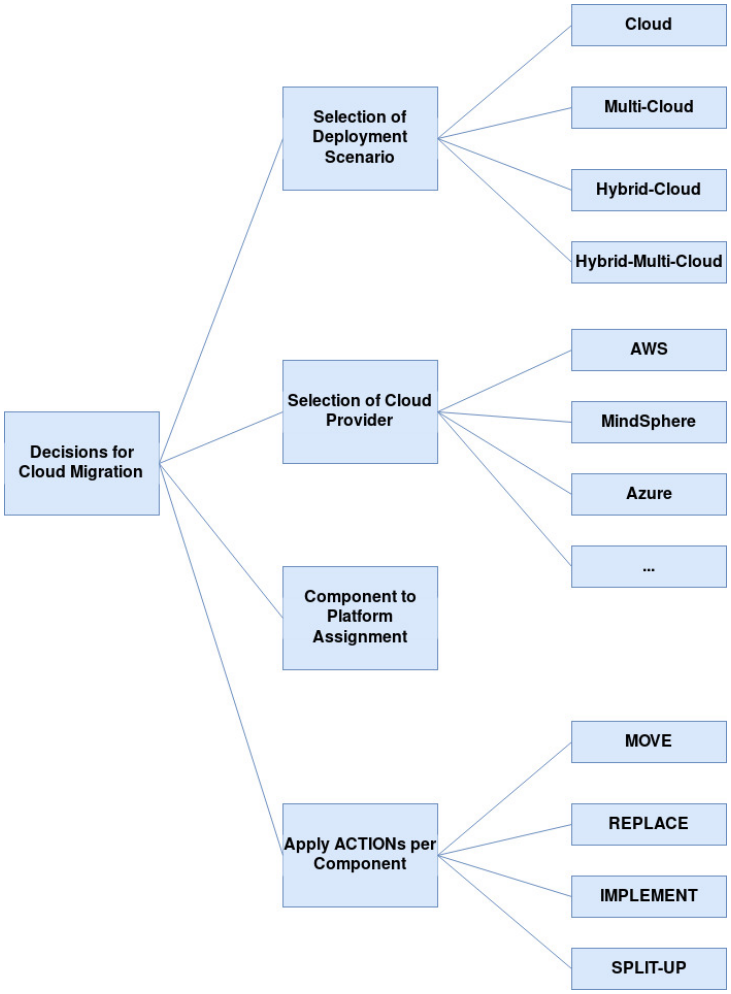


Figure 4.1: Taxonomy of decisions

In case of a legacy application this could mean that the application must be refactored which could mean a lot of effort for the developers.

Selection of Cloud Provider

A second decision which is closely related to the first one would be the selection of one or multiple cloud provider(s) depending on the deployment scenario (see Figure 4.1). Often the cloud provider to use is fixed by the management as well as multi-cloud deployment is not possible due to cost constraints. So the first two decisions are clear. In case the software architect is not given a cloud provider he can base his decision on the reputation as well as the capabilities (e.g. available offerings) of the cloud provider as presented by Gudenkauf et al. [GJGN13]. Furthermore, there already exists tools for selecting a cloud provider such as the cloud vendor shortlisting tool by Info-Tech research [Gro11].

Selecting a deployment scenario and selecting cloud providers also limits the decision state space, for example the cloud offerings to consider. Here it should be noted that besides cloud platforms also edge platforms can be chosen by an architect. This would allow the realization of cloud-edge architectures, which can be more suited for certain use cases.

Component to Platform Assignment

After a deployment scenario is selected and the cloud providers are chosen the next decision is which components to move into a cloud environment, in case of a multi-cloud deployment scenario also redundant deployment of a component to multiple cloud providers would be possible, and which components to leave on-premises. So one component may be used more than once when assigned to multiple cloud and/or edge platforms. As already mentioned an application often must respect hardware constraints or regulatory constraints which make a cloud deployment for at least some of the components not possible. This also means that the first decision of choosing a deployment scenario may have to be adapted, e.g. changing a cloud deployment to a hybrid-cloud deployment. To summarize the problem is examined component-wise respecting all the hard constraints of a component. Soft constraints such as certain QAs (e.g. scalability or performance) are not part of this decision, but they are delegated to the next decision.

Apply Actions per Component

Having an assignment from components to cloud platforms, some additional decisions must be made to determine what to do with each component. These decisions are referred as *actions* in this work. Possible actions would be moving a component as is into the cloud using some PaaS or IaaS offering (referred to as MOVE in Figure 4.1). For a PaaS offering this may not always be possible since they often only support a limited set programming languages/runtime environments. For example, it would not be easily possible to deploy your legacy C application onto an AWS Elastic Beanstalk since this does not support the C language out of the box [Amac]. So you would have to use an IaaS offering such as AWS Elastic Compute Cloud instead and cannot benefit from the PaaS properties such as capacity provisioning, load balancing, auto-scaling, or application health monitoring [Amab].

Another action, which could be taken for a single component, would be replacing it with some SaaS cloud offering such as a managed database service or a managed messaging solution such as Amazon Simple Queue Service [Amak] (referred to as REPLACE in Figure 4.1). However, here it is important that the cloud offering which should replace a local component has equal capabilities than the component. If the component and the offering are not equal, code may have to be adapted. For example when changing from a relational to a distributed NoSQL database some features will be missing as well as semantics will change (e.g. from ACID to eventual consistency) which would have to

be compensated in business logic. So an replacement with a not suitable offering would mean an additional burden on the developers.

To compensate problems such as incompatible data formats, changing semantics in the data handling, or communication problems due to an on-premises server behind a firewall being not reachable by a cloud client any more, additional components would have to be implemented (referred to as IMPLEMENT in Figure 4.1). But often there are already cloud offerings which would provide the required capabilities (e.g. AWS Glue [Amag] for data format transformations) or at least help providing the required capabilities with a little development effort (e.g. AWS Lambda [Amai] to host a data processing solution powered by some serverless scripts). So a complete reimplementation is not necessary.

An additional action would be to split up a component, for example to turn a monolithic application into a micro services architecture (referred to as SPLIT-UP in Figure 4.1). However, this will create a lot of development effort and should be done in an iterative manner to limit development effort per iteration. A first iteration step could also be to just deploy the monolith into the cloud which requires no additional development effort. But increasing runtime costs must be considered since horizontal scaling, e.g. adding more instances, will often not be possible due to state fullness of the monolithic application. So a solution would be vertical scaling, e.g. adding more computing resources, which becomes more expensive in operations due to limited to no elasticity. For example, the acquired resources may not be used always completely. Furthermore, vertical scaling also depends on the cloud provider and will not be possible without limits, e.g. limited CPU cores and RAM.

For the MOVE and REPLACE options multiple options are possible where some are more suitable than others. A decision for a MOVE option can be made more easier than for REPLACE since there are mostly only a few options where an component could be moved to. Especially, PaaS solutions should be prioritized compared to IaaS solutions since they provide helpful properties such as capacity provisioning, load balancing, auto-scaling, or application health monitoring. If a PaaS solution is not possible due to a not supported runtime environment a IaaS solution is the only choice if the component should be moved into the cloud. Besides moving the legacy component also a new cloud enabled component could be implemented exploiting PaaS capabilities. However, such a decision must always be carefully made since a implementation of some component from scratch may mean a lot of development effort.

For the REPLACE option the decision may be a bit more complicated due to multiple offerings with the same capabilities, e.g. data storage solutions. Here a ranking based on QAs would be very helpful to simplify the decision which offering to take. Such a ranking must take the similarity of the local component and the compared cloud offering into account, e.g. based on some type hierarchy. Furthermore, also QA constraints of the component should be considered. For example, if the local component requires a high read performance also the cloud offering should have a high read performance. The taxonomy for quality attributes used in this work is described in the following

Section 4.1.2. However, it must be always kept in mind that when multiple, often competing, QAs also a ranking algorithm can only find a trade-off between them.

4.1.2 Quality Attributes for Architecture Decision Support

For architecture decisions making, for example required when doing a *cloud migration*, also multiple *quality attributes* must be considered. Since this work focuses on software architecture the term quality attributes (QAs) is used instead of non functional requirements (NFRs), whereas the term NFR is more used from a requirements perspective.

To get an overview of all the required QAs to consider a taxonomy was defined which can be seen in Figure 4.2. This taxonomy is based on the ISO 25010 standard [ISO] proposing a quality model for the product quality evaluation of software systems.

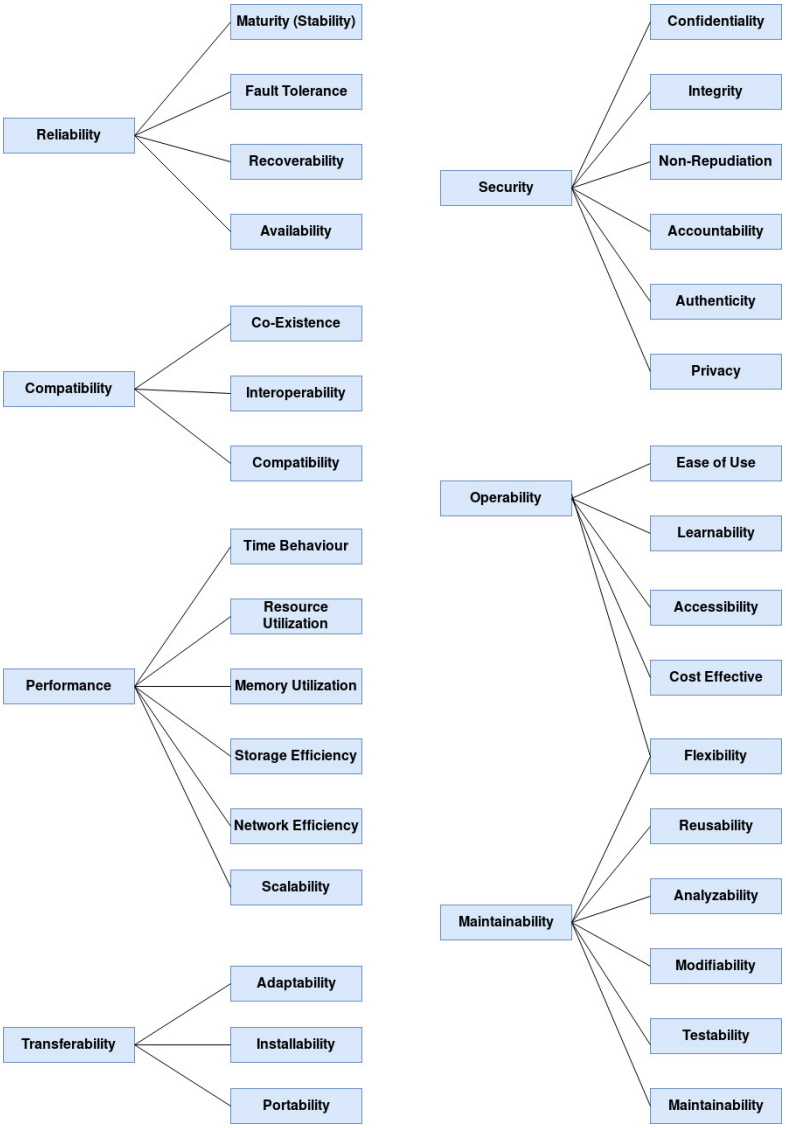


Figure 4.2: Taxonomy of QAs

QAs of the ISO 25010 Standard not included in the Taxonomy

Since this model also contains some QAs not required when describing the QAs of cloud offerings and components in context of cloud-edge environments, there was a need for adaptation. For example, the whole *functionality* sub tree was removed due to not being a NFR since only QAs are considered which are also NFRs. Values such as *functional completeness* or *functional correctness* are always required and need not be explicitly specified. Functional aspects are modelled using *capabilities* rather than QAs. The capabilities a component has clearly define what its functionality is and can be seen as a sort of hard constraints which must be always fulfilled when matching components.

Capabilities are later on further discussed in Section 4.2.3.

Furthermore, *appropriateness*, describing the degree to which users can recognize whether a product or system is appropriate for their needs, was removed since it is hardly usable as a QA for a cloud offering since this also requires some context, e.g. which needs and what is appropriate for them. So it will not make any sense to compare a component with a cloud offering as possible replacement based on their appropriateness value since appropriateness cannot be captured using a single value.

Modularity, describing the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components, is not helpful, because the modularity of some cloud offering is not of interest. Modularity could indeed be helpful to describe single components, but when using a ordinal scale, containing for example high and low modularity, it must be defined what this means and most importantly how this can be measured. This could be done by mapping the ratio scale of the measurements onto the defined ordinal scale, but such a mapping will never be intuitive and different people would define different mappings. Since architecture decision making involves humans, it is not a good idea to use a ratio scale directly since it is hard for humans to give ratings using a ratio scale. However, since the comparison between local components and cloud offerings is of greater importance for architecture decision support in cloud-edge environments compared to the observation of single components, modularity will not be that helpful.

Replaceability was also removed since this makes more sense when compared with anything, e.g. can component x be replaced by cloud offering y. This requires some kind of matching in the knowledgebase which will be described in some later section. As it will be pointed out later on such a matching could for example be done based on a component's capabilities or based on its supported protocol and data formats. If some protocol and data format required are not supported, some transformation component could be implemented or a cloud offering could be used for this job.

QAs of the Taxonomy not included in the ISO 25010 Standard

Besides the removal of some properties which are part of the ISO 25010 standard, also some new properties not included in this standard were added. One example is *availability* which is a very important QA when describing a cloud offering. For some critical business services it is of utmost importance that this value is high, meaning that a service is reachable nearly all the time. A high availability requirement could, for example, be assured by using some PaaS cloud offering as hosting environment which provides this QA or a redundant deployment could be used. For example the Cloud Foundry PaaS solution supports the blue-green deployment [Foua] which would help to increase availability.

A second important property for cloud computing is *scalability* meaning that more resources can be easily added on-demand. Using a scalability property it can also be differentiated between cloud and edge platforms since edge platforms normally have nearly no scalability. In the cloud especially PaaS offerings enable high scalability by

providing automatic load balancing and automatic scalability for an application executed in this environment.

A third property is *flexibility* meaning that some cloud service provides multiple configuration options to be usable for different purposes or it could also provide very flexible data storage mechanisms. Flexibility may for example be important when requirements change which also may have impacts on the data stored. If a cloud offering provides a high degree of flexibility these changes can be easily done, e.g. by reconfiguring a service or minor code refactoring.

Cost effectiveness is another important property companies must consider when selecting some cloud offering. Often there are constraints regarding costs which prevents the use of a certain cloud offering. Regarding cost effectiveness of a service also the different cost models must be considered and the most suitable one regarding cost effectiveness should be selected. For pricing models offered by cloud vendors we refer to the definition of Suleiman et al. [SSJL12]. They differentiate between a *per-use model*, a *subscription model*, a *prepaid per-use model* as well as a combination of a *subscription* and *per-use model*. The per-use model is also known as *pay-as-you-go* where computing resources (CPU, RAM, disk storage and/or network bandwidth) are grouped together and billed per time unit of usage. It is simple, resources can be requested on-demand and nothing has to be paid in advance. When using a subscription model customers sign a contract to reserve computing resources in advance for a certain period of time. The reserved resources are often called *dedicated servers* or *reserved instances* and their prices vary depending on the included resources. The prepaid per-use model is a variation of the per-use model where on demand servers are billed hourly from a prepaid credit. If the credit goes below a certain limit a customer's service is either blocked or he will be charged based on a per-use model. Finally, when using a combination of a subscription and a per-use model dedicated servers must be rented in advance and additional resources can be requested on-demand.

Finally, *privacy* often must be considered, for example when working with sensitive data such as patient data. Therefore, a cloud offering must fulfil certain standards to guarantee data privacy. Privacy can also be seen to some extend as a hard constraints making a cloud deployment of some component impossible. For example the deployment of some data storage component such as a database, could be prohibited due to regulatory compliance. This example very well shows how different QAs could compete with each other, especially when there is also a requirement for high scalability which would compete with privacy since only an on-premises deployment would be possible.

Grouping into Categories

The grouping into categories is done according to the ISO 25010 standard, having the categories *reliability*, *compatibility*, *performance*, *transferability*, *security*, *operability*, and *maintainability* which can be seen in Figure 4.2. For *maintainability* for SaaS cloud offerings such as managed databases it can be said that this is in general very well

supported, because they are maintained by the cloud vendor, e.g. the services are patched and updated by the cloud provider, making operations for a cloud consumer much easier. So a customer do not have to employ people managing such services, which he would have to do when hosting similar services on-premises, and thus reducing costs.

Categories especially important for cloud offerings would be *reliability*, *performance*, *security*, as well as *operability*. *Reliability* such as *stability*, *fault tolerance*, *recoverability*, or *availability* are of high importance especially for critical business services. Moreover, these properties also have consequences on each other, for example a bad fault tolerance will also have a negative impact on availability. Besides using a cloud offering where fault tolerance is well supported, also a *multi-cloud* deployment as discussed in the previous section could be used.

Performance is important when data must be accessed frequently which should be done as fast as possible. Since *performance* can refer to different aspects of hardware and software this is grouped into further categories such as *time behaviour*, *memory utilization*, *storage efficiency*, *network efficiency*, or *scalability*. Especially time behaviour will often be important to provide a service with low response times. Factors that could influence the time behaviour depending on the application would be storage efficiency or network efficiency.

Security is a general aspect which must be considered in all software applications and especially in the context of cloud computing since data is hosted off-premises meaning that a company has less control of their data compared to data hosted on-premises. Moreover, all the services are accessible via a network interface which enables various network attacks relevant for IT and web security. To limit the probability of certain risks such as data theft which would for example violate *confidentiality* certain measurements must be taken such as encryption of confidential data or the use of hash functions to guarantee data *integrity*. Cloud providers such as AWS already provide offerings to enforce data security (e.g. AWS Identity and Access Management (IAM) [Amah]).

Operability of services must also be considered since services selected may be used for a long period of time which means that aspects such as *costs* but also *ease of use*, or *flexibility* regarding the configuration of such services must be considered.

Furthermore, *compatibility* and *transferability* will also be important, for example to avoid a vendor lock in by assuring that cloud offerings use common standards regarding data storage and data transfer so that they could easily be replaced by some other offering of a different cloud vendor.

4.1.3 Cloud Migration Patterns

To support migrating a legacy application into the cloud as well as to refactor applications already (partially) deployed into the cloud, cloud migration patterns are defined in academic literature and by cloud providers such as Amazon [Amal]. The patterns defined by cloud vendors are often specific to their platform so the academic literature should be preferred for selecting the patterns used in an architecture decision support process.

Cloud migration patterns for *cloud*, *hybrid-cloud* and *multi-cloud* deployment (see deployment scenarios as defined in Section 4.1.1) are, for example, defined in the work of Jamshidi et al. [JPCL15]. Alone their work contains fifteen patterns with a detailed description of the problem, the solution, as well as benefits and drawbacks of a certain pattern. Since these patterns may not cover all possible scenarios, for example there is no *replacement with cloud and on-premises adoption* pattern, even more patterns must be considered in a decision support process. So it is not an easy decision which pattern to choose. Therefore some filtering would be helpful, e.g. based on previous decisions. Referring to *RQ4* defined in Section 1.2 it will also be examined in this work, how a software architect could be supported in selecting such cloud migration patterns.

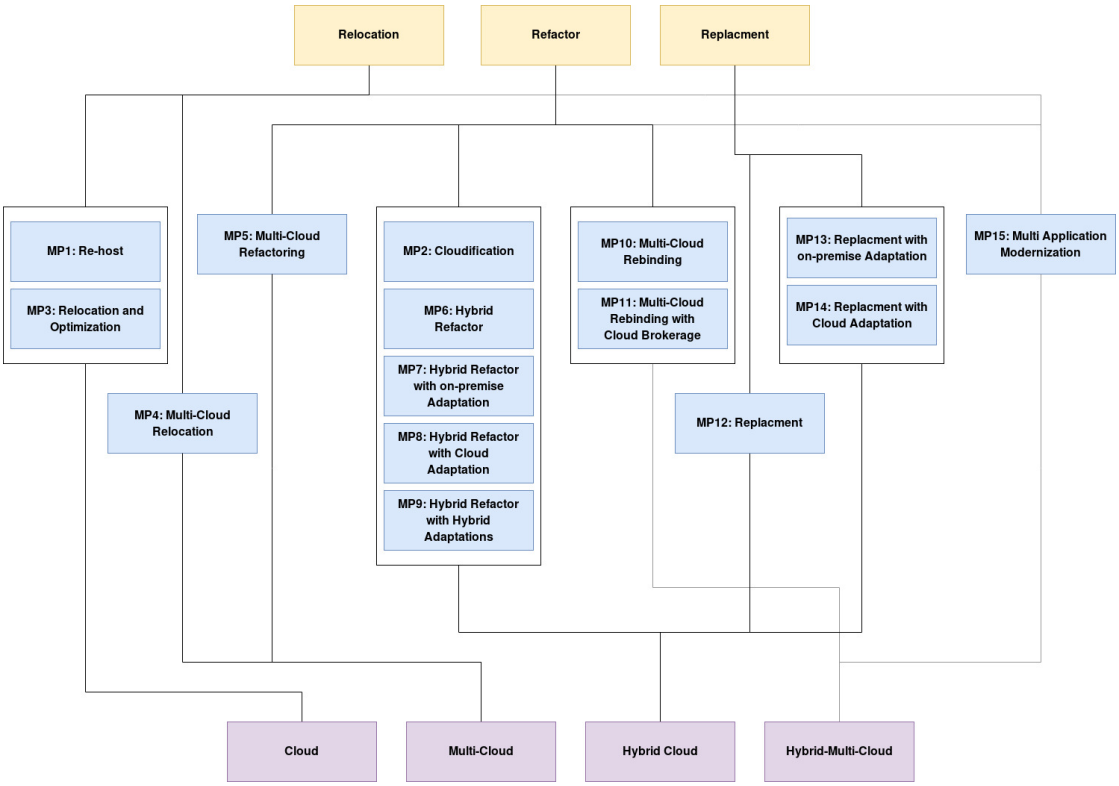


Figure 4.3: Taxonomy of Migration Patterns

In this work a taxonomy of cloud migration patterns was defined which can be seen in Figure 4.3. These patterns are grouped by the deployment scenarios *cloud*, *hybrid-cloud*, *multi-cloud*, as well as *hybrid-multi-cloud* defined in Section 4.1.1. Furthermore there is a grouping after the options *relocation*, *refactor*, and *replacement*. These options map to the actions MOVE, IMPLEMENT, and REPLACE which can be seen in Figure 4.1. The options relocation, refactor, and replacement were chosen to stick to the terms used by Jamshidi et al. [JPCL15]. However, they can be easily mapped onto the actions defined in Section 4.1.1. Note that patterns with the same deployment scenario and option are clustered together to simplify the diagram by saving some lines.

Using the taxonomy defined in Figure 4.3 the patterns can be filtered depending on the deployment scenario chosen (see the *selection of deployment scenario* decision in Figure 4.1). Furthermore, they can be filtered based on the action chosen (see the *apply actions per component* decision in Figure 4.1). In the following the most important patterns are briefly described including a reasoning why they are belonging to certain deployment scenarios and migration actions. For a detailed description we refer to the work of Jamshidi et al. [JPCL15].

The pattern *MP1: Re-host* is the simplest pattern for cloud migration where an application/component is simply moved as-is into the cloud without the need for any adaptations. Although it was believed from researchers including Jamshidi et al. it is the most commonly found solution in literature due to its simplicity, their literature review of Cloud Migration already made in 2013 showed that this is not the case. This pattern belongs to cloud and relocation in the taxonomy presented in Figure 4.3 because it only address a single cloud platform as well as it is a simple MOVE action with no requirements for any refactoring.

The *MP2: Cloudification* pattern describes a simple on-premises refactoring of an component to use one or multiple cloud offerings, but the component itself stays on-premises. This pattern belongs to refactor since an application must be often refactored to support the interface of a cloud provider's offering. Furthermore, it is only usable when a hybrid deployment is used (e.g. it belongs to hybrid-cloud and hybrid-multi-cloud deployment scenarios), because the component itself stays on-premises.

Furthermore, there are relocation patterns targeting cloud (*MP3: Relocation and Optimization*) as well as multi-cloud (*MP4: Multi-Cloud Relocation*) deployment scenarios. Here an application/ a component is moved into the cloud (similar to the *MP1: Re-host* pattern). The difference between MP3 and MP4 is that MP4 uses cloud offerings of different cloud providers.

The patterns *MP5-MP9* present various refactoring patterns. They would, for example, be valuable for the SPLIT-UP action presented in Figure 4.3 when part of the migration is a refactoring to a micro services architecture. Such patterns can be applied in an iterative manner to limit the risk of the refactoring. There are refactor patterns targeting multi-cloud deployment (*MP5: Multi-Cloud Refactor*) as well as various hybrid deployment patterns (*MP6: Hybrid Refactor*, *MP7: Hybrid Refactor with on-premises Adaptation*, *MP8: Hybrid Refactor with Cloud Adaptation*, and *MP9: Hybrid Refactor with Hybrid Adaptations*). Using pattern MP5 an application is split into two components which then can be deployed onto two different cloud platforms. But they could also be deployed onto a single cloud platform and because of the split-up they can better utilize a cloud platform's scalability opportunities. MP6 is similar to MP5 but here one component is left on-premises and the other is deployed onto a cloud platform. The patterns MP7 - MP9 are hybrid deployments with the need for implementing new components which are either hosted on-premises, in the cloud, or both depending on the pattern.

The patterns *MP10: Multi-Cloud Rebinding* and *MP11: Multi-Cloud Rebinding with*

Cloud Brokerage are both hybrid multi-cloud patterns requiring a refactoring of the architecture by at least adding a component. The patterns *MP12* - *MP14* patterns describe various replace actions for hybrid as well as hybrid multi-cloud deployment use cases. Such patterns are for example valuable to fix communication errors which could arise as consequence of cloud migration when a cloud client must connect to an on-premises server behind a firewall. However, as already previously mentioned a *replacement with cloud and on-premises adoption* pattern is missing which would be a combination of MP13 and MP14 deploying one new component on-premises and other into the cloud. Finally, there is the *MP15: Multi Application Modernization* pattern which describes the refactoring and relocation of an application hosted on multiple on-premises platforms.

4.1.4 Meta Taxonomy to Describe Cloud Offerings

Another requirement for architecture decision support in cloud-edge environments is that the cloud platforms including its offerings must be described. For that purpose the meta taxonomy in Figure 4.4 was developed to describe a cloud platform with all its offerings.

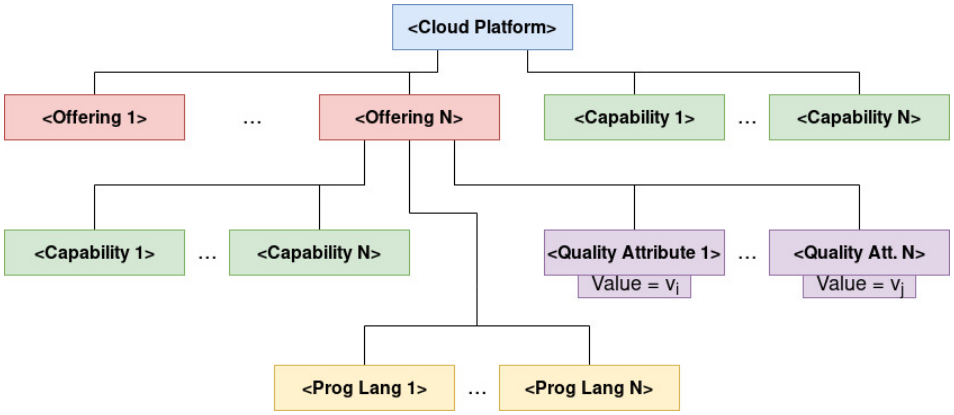


Figure 4.4: Meta taxonomy to describe cloud offerings

Each offering is described by their *capabilities* as well as *QA constraints*. For some offerings, such as PaaS offerings, also their *programming language support* is of interest. Each QA is given a *value* to describe how good some offering performs regarding this QA. As scale for this value an integer scale may be chosen, for example in the range 1 - 5, 5 meaning the offering performs very well regarding this QA. Especially, if the QA value is given by some human expert an integer scale with a small range is helpful compared to a much bigger scale such as a scale between 1 - 100 since so the human expert has fewer choices which simplifies decision making. Furthermore, metrics gathered at runtime such as metrics for availability of some service could also be easily mapped onto this scale. Having each QA of a cloud offering rated with a value enables ranking based on QAs.

Cloud offerings with the same capabilities can be compared in this way supporting a software architect in selecting the best offering based on his requirements regarding QAs.

The meta taxonomy presented in Figure 4.4 also contains the capabilities of the whole cloud platform. These capabilities are the union of all the offerings' capabilities. Having capabilities at this top level of the taxonomy is also very helpful, because it enables the matching of some on-premises application with a whole cloud platform. The on-premises application's capabilities are also the union of the capabilities of all its components, similarly as for cloud platforms. Now when matching the on-premises application with all cloud offerings, the number of matched capabilities could be used as one criteria for selecting some cloud platform. The cloud platform with the most matches supports more capabilities required by an application compared to all other cloud platforms considered. So regarding capabilities it would be beneficial to choose this platform.

However, since capabilities are not the only selection criteria for a cloud provider, other criteria must be considered as well (e.g. QAs such as costs, performance, scalability opportunities, etc.). Such QAs could also be modelled top level for the whole cloud platform, but it is much more difficult to give a useful value for them. For example, how would be the *performance* of the whole cloud platform be defined? Should it be the maximum value of performance found among all the cloud offerings of the platform? Or should it be some mean value of all of them? Taking the maximum value would not have that much expressiveness, since the offering having a maximum regarding this single value could be of no use for the application that should be migrated onto a cloud platform, e.g. due to supporting the wrong capabilities. Using some derived statistics such as the mean value would be more difficult to maintain since if one of the QA values of any cloud offering is updated, the mean value must be updated as well. Furthermore, the expressiveness of such a value could also be problematic since similar to the mean value also in this case there could be multiple offerings affecting this value positively which are not usable for the application, e.g. as in the previous case due to support of the wrong capabilities. Therefore it was decided for the meta taxonomy in Figure 4.4 to omit QAs at the top level. Furthermore, some requirements such as the support for a certain cost model, or for certain hardware configurations could also be expressed via capabilities.

4.2 Conceptual Model and Concepts for Decision Support

The following section describes concepts relevant for software architecture decision support in cloud-edge environments as well as the conceptual model used in this work. *Quality attributes* (see Section 4.1.2) as well as *cloud migration patterns* (see Section 4.1.3) were already described in previous sections. Table 4.1 gives an overview of the concepts used in this work.

Table 4.1: Concepts relevant for this work

Concept	Description
Platform	A hosting environment for applications, e.g. *cloud platform*, *edge platform*
Software component	The entity used as part for decision making
Capabilities	The functionality an application/component provides.
QA/ QA Constraint	Quality attributes such as scalability, performance. Also referred to as *quality attribute constraints* in this work. Only QAs are considered which are also NFRs (see Section 4.1.2).
Domain	A business domain where some application belongs to.
On-Premises App	Application (component) is hosted in-house on own hardware.
Workflow	An orchestrated, repeatable, and systematic sequence of operations
Hybrid-Cloud	An application is deployed partly on a cloud platform with parts on-premises
Multi-Cloud	An application is deployed on multiple cloud platforms (two or more)
Cloud Offering	Some SaaS, PaaS, or IaaS solution provided by a cloud platform

4.2.1 Platforms

Platforms such as *cloud platforms* or *edge platforms* are a main concept in this work since they are the source and/or target environment when doing a cloud-edge migration.

All platforms have certain technical and non-technical constraints which must be considered when doing a migration into this environment. Examples for technical constraints would be hardware constraints such as GPU computing support on a certain platform. Non-technical constraints could be for example legal constraints such as licensing constraints, e.g. that extensions for a certain database offering licensed under GPL are not permitted.

All these constraints have in common that they can be seen as hard constraints. An important concept belonging to cloud platforms are *cloud offerings*. Regarding cloud offerings it can be differentiated between the three different service models IaaS, PaaS, and SaaS which were described in detail in Section 3.3.1. Especially for the SaaS service model there exist many offerings, multiple of them targeting a similar purpose and having

the same capabilities, only differentiated by their QAs. A very good example are different data storage solutions which all have the capability persistent data storage.

4.2.2 Software Components

Software components are described by their *capabilities*, *QA constraints*, as well as *interfaces* they expose to other software components (or also referred as *links* in this work). Here it can be differentiated between the interfaces the component *provides* as well as the interfaces it *requires*. Examples for interfaces a component provides are a REST server, web socket server, etc. and examples for interfaces a component requires are a REST client, a web socket client, and the like. Furthermore, components' interfaces can also be described by their supported data formats, e.g. by referencing to some documentation of a communication interface such as a Swagger documentation of a REST interface.

Software components can be composed of multiple software components all connected via interfaces. This also enables software components to act stand-alone which means that they are *applications* on their own.

4.2.3 Capabilities

An important concept used in this work are the *capabilities* a software component or application provides describing its functionality from a high level view. The term *capability* may also be referred as *business capability* or *functionality* in this work.

Examples for capabilities are *persistent data storage*, *data visualization*, *access control*, *geospatial data handling*, *messaging*, or *data processing*. These can be used as hard constraints to compare software components with each other. This is for example helpful when one component should be replaced by some other component which means that the new component must provide at least the same capabilities the current component provides.

Furthermore, special capabilities such as *operating system hosting*, *application hosting*, or *container management* can be used to identify IaaS or PaaS cloud offerings or container management technologies such as Docker. Moreover, capabilities can also be used to model hard constraints such as GPU computing support on a certain cloud platform.

4.2.4 Domains

The idea of *domains* or *business domains* is that an application belongs to one or multiple domains which may have to respect their own constraints (e.g. QA constraints). These constraints must be considered for each application belonging to this domain.

For example in the health care domain data privacy is important which must be respected by all applications in this domain working with patient data. Now by specifying a domain their QA constraints can be automatically imported. Further examples for domains

would be a companies business units which could also have their own constraints they must follow.

4.2.5 Application Types

A taxonomy of *application types* was developed for that work which is used for the ranking algorithm presented in Section 5.3. An important point of the taxonomy of application types is that multi inheritance is also possible, creating a type hierarchy in the form of a directed acyclic graph. In Figure 4.5 an example of the application type hierarchy with some AWS storage offerings as well as an on-premises NAS (network attached storage) application can be seen.

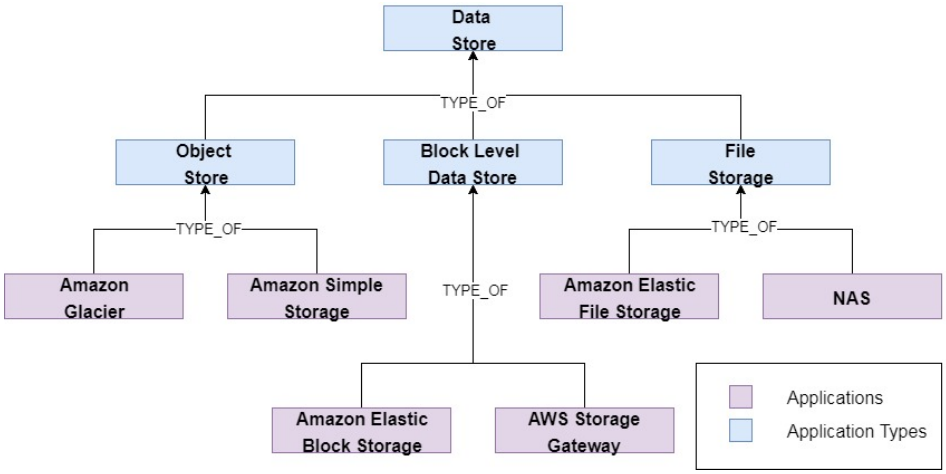


Figure 4.5: Application type hierarchy example with AWS storage offerings and an on-premises NAS application.

The application type hierarchy taxonomy can be very well shown on the example of data storage technologies. Here a general root node called *data store* is defined which can be further grouped into *dataset storage*, *object store*, *block level data store*, and *file storage*. The *dataset storage* can be further divided into *relational database*, *distributed database*, *object-relational database*, *object database*, and *NoSQL database*. Having this taxonomy structured as an acyclic graph a *similarity* metric between a cloud offering and some on-premises component can be defined by using the path length between the two application nodes.

For example, when comparing the NAS with Amazon Elastic Block Storage in Figure 4.5, a path length of 4 would be returned, but for the NAS and the Amazon Elastic File Storage only a path length of 2 would be returned since both applications have the same type. So by examining this example it can be seen that the minimal path length returned is 2, e.g. 2 means that the applications are equal in type. Every longer path length is somehow worse since the application types are only similar due to some super type relation. So for calculating a normalized score in the interval from 0 to 1 the following

formula could be used where *Length* is the path length between two nodes a and b in the taxonomy graph and $a \to b$ means that a is connected with b:

$$S(a,b) = \begin{cases} \frac{1}{Length(a,b)-1} & \text{if } a \to b \\ 0 & \text{otherwise} \end{cases}$$

This formula has the properties that a shorter path length gives a better score, the minimal path length of 2 gives the maximal score 1 and since the path length increases the result of this function converges to 0. In case of there is no path between two components, e.g. they are not similar in any way, the score is 0.

CHAPTER 5

Architecture Decision Support Design and Prototype

The purpose of this chapter is to present the solution for architecture decision support developed in this work. The contributions of this work consists of the following parts:

- An *architecture decision support workflow* was developed to support architecture decision making in cloud-edge environments which targets software architects. Using this workflow decision making is done in a structured manner rather than an ad-hoc process and in that way decisions are improved. This workflow is presented in Chapter 5.1.
- The *architecture* of the decision support tool developed in this work is explained in Chapter 5.2. It consists of three layers and for each layer its purpose as well as its responsibilities are explained.
- The architecture decision support workflow is supported by an *Analytic Hierarchy Process algorithm for offering ranking* which was specially adapted to this use case. The algorithm helps a decision maker to select the best offering as replacement for some component based on the component's *QA requirements* as well as its *application type*. Chapter 5.3 presents the algorithm, its runtime properties, as well as an illustrative example.
- A *prototypical implementation* of the tool which is briefly explained in Section 5.4.

5.1 Architecture Decision Support Workflow

To support the software architect based on the requirements regarding the decisions to support defined in Section 4.1 a workflow consisting of five steps was defined. The whole

workflow is illustrated in Figure 5.1.

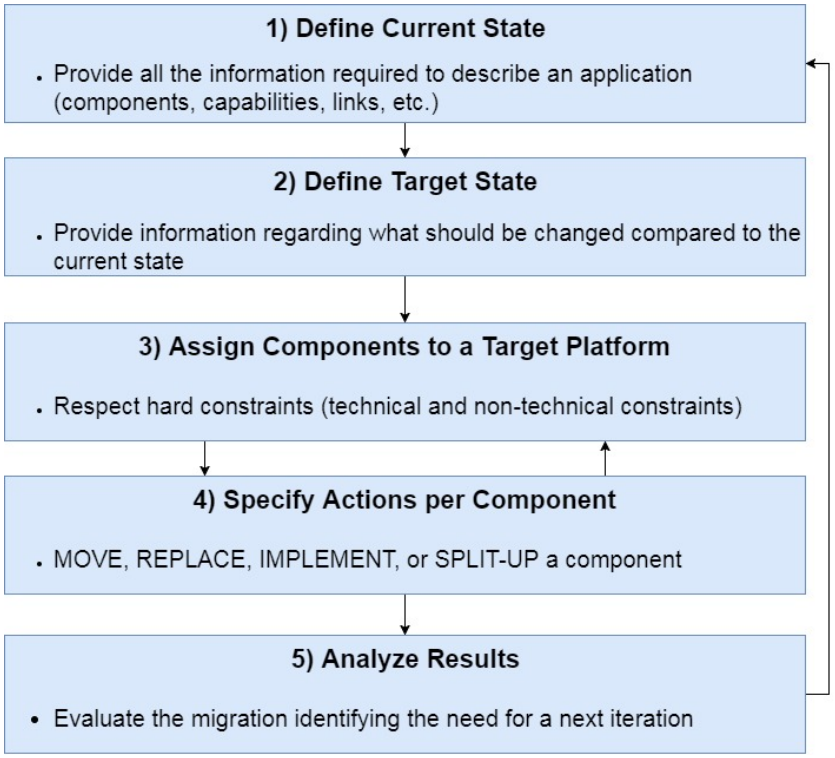


Figure 5.1: Decision Support Workflow

One important precondition of the workflow is that all the cloud offerings are already in the knowledgebase. They should be described with all their offerings, and for each offering their *capabilities* and *QA constraints* are of importance since they are used for matching and ranking in step 4 of Figure 5.1. For some offerings such as PaaS offerings their *programming language* support is also of importance. The meta taxonomy to describe cloud offerings presented in Section 4.1.4 could be used for that purpose.

5.1.1 Define Current State

The first step contains the description of the current application. In this step the software architect may provide all the data needed to describe his application. It is an important precondition of the workflow that an application is described with all its *domains*, *capabilities*, *QA constraints*, *components* as well as the *links* between them including the communication style/protocol used (e.g. REST, web socket, ...).

The components itself may be described by their capabilities, QA constraints, *application type* as well as the programming language used. Especially, a component's QA constraints as well as its application type are of huge importance later on since they are used by the

ranking algorithm in step 4 of Figure 5.1, e.g. when a component should be *replaced* by some cloud offering to get the most suitable cloud offering.

Besides providing all the information required via some user interface also a data import from common architecture description formats such as UML diagrams would be possible to get all the information required for this first step.

5.1.2 Define Target State

In the second step the *target application* is defined. This means that goals regarding the improvement of QA constraints of some components, or the addition of new capabilities may be specified. When adding new capabilities this means that new components may have to be implemented or cloud offerings could be used fulfilling some of these capabilities if there are any. Furthermore, the domain this application is placed in could be changed. This would have effects on constraints relevant for the whole application.

Finally, a deployment scenario together with the platforms used may be selected. So this step includes the first two decisions, *Selection of Deployment Scenario* and *Selection of Cloud Provider*, defined in Section 4.1.1 limiting the platforms to consider.

5.1.3 Assign Components to a Target Platform

In the third step all components must be assigned to a target platform as defined for the third decision of Section 4.1.1, the *Component to Platform Assignment*. Here hard constraints are checked such as hardware constraints or legal constraints to filter out platforms where a deployment is not possible due to a violation of any these constraints. Apart from that the architect can freely decide which platform to chose. In this way the workflow is quiet flexible and it does not limit a software architects decisions.

5.1.4 Specify Actions per Component

The forth step is the most important step of the workflow where the *actions* MOVE, REPLACE, SPLIT-UP, DEFER, and DELETE are applied component wise. It represents the *Apply Actions per Component* decision defined in Section 4.1.1. The semantics for MOVE, REPLACE, and SPLIT-UP were already defined there. The MOVE and REPLACE actions are additionally supported by a ranking algorithm defined in Section 5.3 to get the most suitable offering based on their QA constraints and their application type hierarchy.

The scope for the component to consider is limited to the platform selected in the previous step which reduces the complexity of this decision. For example if AWS was selected in the previous step and in the forth step REPLACE is selected only offerings of AWS are returned and no other platforms are considered. In this step it is also possible to go back to the previous step as indicated in Figure 5.1 to change the platform assignment for a certain component. In case the MOVE and REPLACE actions return no result for a

component, DEFER may be chosen which means leave the component where it currently is, e.g. an on-premises component will stay on-premises.

To support the architects decisions the forth step also contains a graph view of the current architecture of the application showing components and the links between them. Furthermore, this graph view also shows communication errors which may for example appear when a cloud client must communicate with an on-premises server in the *target architecture* which is no longer possible due to firewall restrictions. To solve such an error the architect is supported by a *cloud migration pattern*. In case of the communication error the addition of a proxy component would be suggested. This would be the IMPLEMENT action as described in Section 4.1.1. Additionally, matching in the knowledgebase is applied to find a possible cloud offering which could be used as proxy. If nothing can be found a new component must be implemented, but if something is found one of the results can be used by the architect. This has the same semantics as the REPLACE action, but without having an existing component.

Furthermore, in the forth step also incompatibilities of a local component and a cloud offering regarding the used data format could appear. To solve such an issue a software architect is also supported by a *cloud migration pattern* suggesting the architect the implementation of some kind of an adapter, e.g. by hosting a data transformation script on AWS Lambda. Moreover, there could also be the lack of some capabilities which will have to be implemented using a new component. An example would be when changing from a relational to a NoSQL database there would be no stored procedures which would have to be implemented in business logic. To get the patterns filtered by the current use case, e.g. if it is a relocation, a replacement, or a refactoring, the taxonomy of cloud migration patterns defined in Section 4.1.3 can be used.

Finally, the DELETE action means that a component is permanently deleted which may also be sometimes needed. After an action is applied to each component and there are no errors left, such as the communication errors previously discussed, the architect can proceed to the fifth step.

5.1.5 Evaluate Migration

In the fifth step the results of the migration workflow are presented including the final target architecture and a report documenting the decisions made. The report is supported by some statistics such as ratio of used cloud offerings to stand-alone/self-hosted components so that a software architect can assess the quality of his solution. Moreover, when having data from multiple iterative cycles in the workflow graphs can be created, e.g. how the ratio of used cloud offerings to stand-alone/self-hosted components evolved over time. After the fifth step a software architect can decided to start a new iteration and proceed to step 2. Here he can define a new target state, but with the current state being the result of the previous iteration.

5.2 Tool Architecture

For the design of the decision support tool a *Three-Tier-Architecture* was used consisting of a *Decision Support Workflow Frontend*, a *Middleware for Decision Support Services*, as well as a *Knowledgebase* as data store which can be seen in Figure 5.2.

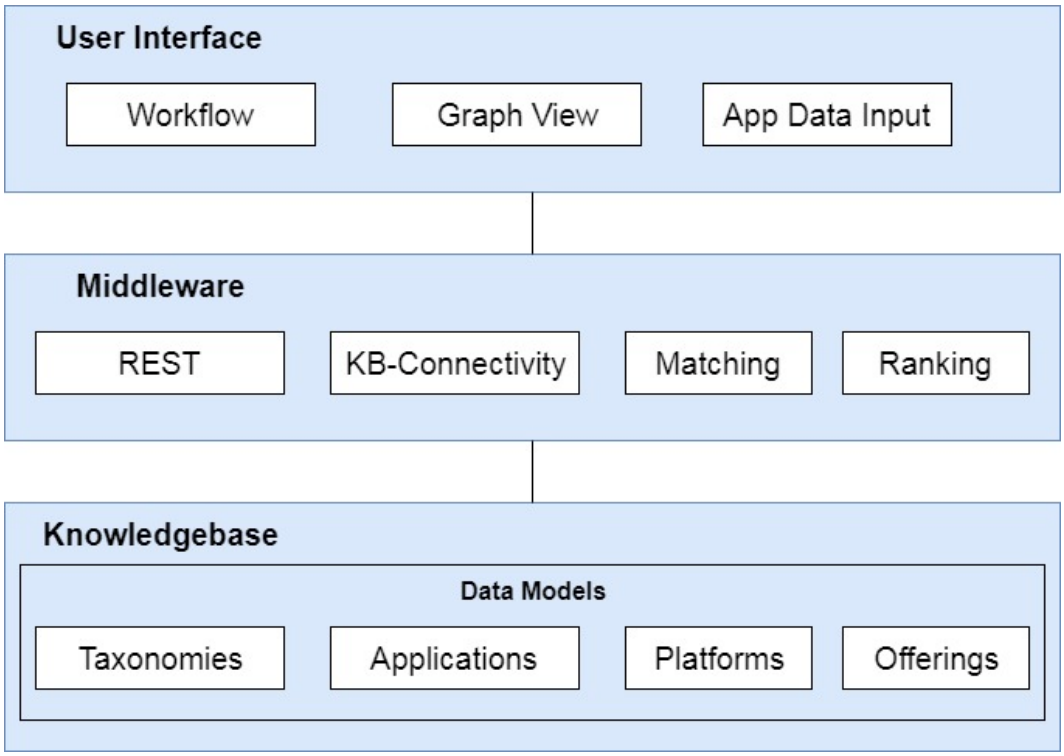


Figure 5.2: Decision Support Tool Architecture

Main components and requirements for the Decision Support Workflow Frontend are that it is structured using the *workflow* presented in the previous section. Furthermore, each step should be supported by a *graph view* showing the architecture of the application with its components and connections between the components. Such a graph view provides the necessary context informations for an architect's decisions. The frontend also has the requirement to provide the necessary *data input* capabilities for a software architect to enable the description of his application regarding capabilities, QAs, domains, components, links between components, etc. Usability is an important NFR for the whole frontend since it is the software architect's interface to the decision support system which must be intuitive to use.

The Middleware for Decision Support Services must provide connectivity to the Decision Support Workflow Frontend, e.g. via a REST interface. Furthermore, it must have the main decision support functionality including *matching* and *ranking* capabilities of

cloud offerings. Finally, also the connectivity to the knowledgebase is of importance to efficiently access the data stored there.

The Knowledgebase is responsible for storing all the data needed for decision support. This includes various *taxonomies*, for example the taxonomies presented in Section 4.1. Furthermore, the software architects' *applications* must be saved described by their capabilities, QAs, information where they are currently deployed, etc. Finally, all the *platforms* such as cloud and edge platforms must be stored there. For cloud platforms also the description of their *offerings* are of importance, for example by using the meta taxonomy presented in Section 4.1.4.

The following sections give an overview of each layer also mentioning their responsibilities.

5.2.1 Decision Support Workflow Frontend

Figure 5.3 shows the main screen of a Decision Support Workflow Frontend.

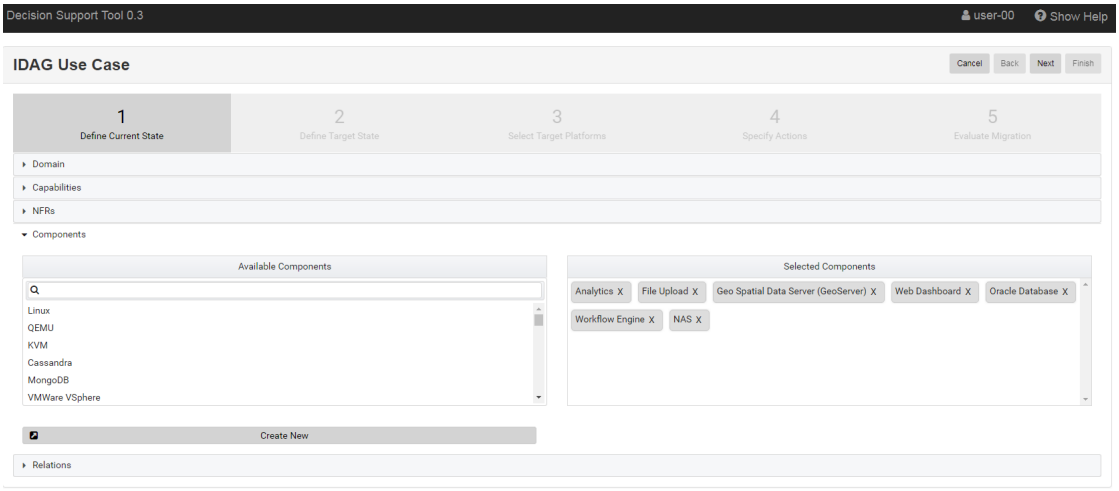


Figure 5.3: Decision Support Workflow Frontend

In Figure 5.3 it can be very good seen that it is structured using the workflow presented in Section 5.1. This gives the architect an overall guidance in the software architecture decision process. Furthermore, context information is provided when clicking on the *Show Help* button in the upper right corner. After that button is clicked a help appears with a description of the current step of the workflow, its purpose, as well as of important concepts used in this step. An example of the concepts appearing in the first step of the workflow which is visible in Figure 5.3 would be domains, capabilities, NFRs, components, and relations. Note that the UI uses the term NFRs instead of QAs as a label because we assumed that users are more familiar with this term as well as that a user of the tool is more likely looking at the problem from a requirements perspective.

After the work in one step of the workflow is finished it can be proceeded to the next step by clicking the *Next* button. At any step it can be always returned to the previous step by using the *Back* button.

The main responsibility of the Decision Support Workflow Frontend is the interaction with the user, the software architect. So usability as well as simplicity are main requirements for it, because if such a tool is difficult and cumbersome to use no software architect will use it. The structured workflow proposed in the previous section also helps to reduce complexity by distributing decisions among the different steps of the workflow.

Flexibility is also an important property of the frontend which is already solved by the workflow since it does not restrict an architect in his decisions by having a choice among different options and decide what to do with each component.

A third point is transparency and traceability of the decisions. This is achieved by showing a graph view of the architecture with its components and the links between the components annotated with the technology used. Having a graph view of the architecture is very important for the transparency of the workflow. The traceability is improved by showing relevant information about the decisions made, some consequences of the decisions, and statistics incorporating different metrics such as the ratio of used cloud offerings to stand-alone/self-hosted components mentioned in the previous section.

The frontend will access the functionality provided by the *Middleware for Decision Support Services*. When designing the tool as a web application the frontend would be the state-full part running in a software architect's web browser. This means it also has the responsibility for keeping track of a user's session. Designing the decision support tool as web application also has the benefit that multiple software architects can collaborate sharing a common knowledgebase.

Enabling multi user support, e.g. by using a frontend built with state-of-the-art web technology, also entails further requirements, namely various security requirements. For security especially authentication is important. So it should not be possible for some user to access the applications of someone else if he does not want to share them.

5.2.2 Middleware for Decision Support Services

The purpose of this section is to describe the Middleware for Decision Support Services which can be seen in Figure 5.4.

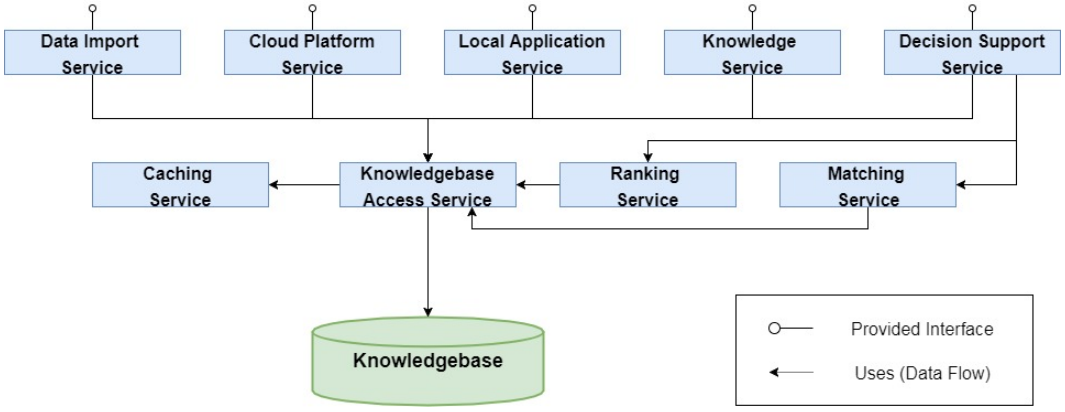


Figure 5.4: Middleware Services

The Middleware for Decision Support Services is responsible for persisting, querying, updating, and deleting from the knowledgebase. So one point is providing simple CRUD functionality for all the knowledgebase elements. This should be done in a stateless way so that the middleware can exploit the opportunities provided by the cloud platform, e.g. when deploying multiple instances of the middleware and distributing the traffic using a load balancer. Furthermore, besides simple CRUD functionality this middleware must also contain the logic for architecture decision support, e.g. the matching and ranking capabilities required to return possible offerings in case when a MOVE or REPLACE action as defined in Section 4.1.1 was chosen.

The middleware is composed of multiple services which can be seen in Figure 5.4. There is the *Cloud Platform Service* providing CRUD functionality for cloud platforms in general. Ordinary users would only have access to its query capabilities whereas the create, update, and delete capabilities are only limited to administrators of the decision support system.

Furthermore, there is the *Application Service* usable by every software architect using the decision support tool. This service provides the CRUD functionality for a software architect's application providing means to specify an application's components with their capabilities and NFRs as well as the links between the components including the communication technology used. The applications of each user are kept private unless he specifies that he wants them to share with other users in the system. Moreover, there is the *Knowledge Service* for providing CRUD functionality for software engineering domain knowledge such as protocols, standards, well known applications and terms which has similar access rights such as the *Cloud Platform Service*.

There is the *Decision Support Service* proving all the functionality required for an architecture decision support system. This includes services for matching and ranking of cloud offerings which can be for example done with the algorithm presented in Section 5.3. All the above mentioned services are connected to the knowledgebase via the *Knowledgebase Access Service* implementing the client technology for the knowledgebase.

Finally, there is a *Caching Service* which speeds up requests made to the knowledgebase. This service must be carefully designed to enable redundant deployment of the whole middleware. For example the caching of knowledgebase elements which does not change that often such as cloud platforms and their offerings would be possible. Also the caching of a user's applications is possible, but changes such as updates and deletes must be propagated among all running instances of the middleware.

5.2.3 Knowledgebase

When designing an architecture decision support tool it is very important how to represent common knowledge to be able to give helpful decision support and to persist the decisions as well as domain knowledge to enable knowledge sharing among all software architects using such a system. Besides persisting decisions and knowledge also querying this information in an efficient way is important. So a well defined structure is needed.

On the other hand, due to the evolving nature of software engineering domain knowledge, meaning that some technology now being state-of-the-art could be replaced by some other technology soon in the future, the model used for representing knowledge in an architecture decision support system must also provide some kind of flexibility.

So in order to have a well defined structure and be easily query-able as well as to provide the required flexibility a graph based model was chosen. Such a graph based model enables the composition of entities in a recursive way as well as the representation of type hierarchies which is all needed for a solid foundation of an architecture decision support system.

The graph based models used in this work are the domain Entities Meta Model as well as the Application Description Model [PSZ18]. These models consist of a number of object collections and hierarchies represented in a very flexible graph model. The former model represents the *general* software engineering domain knowledge (all relevant products, terms, and concepts). The latter model is a detailed description of a certain application including all its components, architecture concepts, business capabilities, data inputs, data outputs, implementation languages, licenses, and deployment as well as QA constraints.

The knowledge model describes a type hierarchy from specific to more abstract entities, enabling inheritance. This model can be extended with further levels such as a deployment level showing the infrastructure (e.g. hardware nodes) where a specific application is deployed or a software ecosystem level showing product relationships on a high level view.

However, these two models only provide a rough structure for knowledge data but no knowledge itself which is needed to support decisions. So a taxonomy of software engineering domain knowledge was designed in this work exploiting the two meta models. Parts of this taxonomy is the taxonomy of QAs as defined in Section 4.1.2 as well as the taxonomy of cloud migration patterns as defined in Section 4.1.3. Furthermore,

taxonomies of application type hierarchies, capabilities, licences, as well as protocols and standards were defined. Moreover, a taxonomy of AWS as well as MindSphere cloud offerings was defined by collecting information from documentation pages to describe their capabilities and NFRs. This taxonomy is structured using the meta taxonomy to describe cloud offerings presented in Section 4.1.4.

5.3 Algorithm to Rank Cloud Offerings

In Section 4.1.2 it was already discussed that many QA constraints may have to be considered for software architecture decision making. Cloud vendors such as AWS provide multiple offerings with similar capabilities which differ in the QA constraints they have. So decisions such as MOVE or REPLACE mentioned in Section 4.1.1 could return many offerings where it is not always that easy for a software architect to choose the right one to fulfil all the QAs required. For example, when considering offerings with the capability *persistent data storage* multiple offerings would be returned which could only be differentiated by their QAs as well as their *application type*. The application type is only relevant for the REPLACE option since for the MOVE option special application types describing the cloud offering such as *application hosting* or *operating system hosting* are used and such offerings are not comparable with a component considered for moving onto a cloud platform, also having different capabilities.

The application type is also of importance when replacing a component with a cloud offering because if the application types differ there could be incompatibilities. For example when changing from a relational to a NoSQL database this would mean huge changes in the dataformat as well as some features would have to be implemented in business logic if they are not supported by the new data storage solution.

So it can be concluded that a ranking of the offerings returned for the MOVE or REPLACE options by their QAs as well their application type would be very helpful for a software architect to identify the right offering to solve his problem. Nevertheless, he must do his research as well and learn about the offerings, their benefits and drawbacks, but the ranking helps to guide the software architect into the right direction and speed up his decision making especially when he do not know the cloud offerings that well.

5.3.1 AHP Algorithm

For ranking offerings based on their *QA constraints* and their *application type* an Analytical Hierarchy Process (AHP) algorithm was created. It is based on the original idea of the AHP developed by Satty [Saa90] adapted to this use case. Figure 5.5 shows an example hierarchy for ranking some cloud offerings based on their QA constraints an their application type. In this example only the QAs scalability, storage efficiency, availability, operability, performance, and cost effectiveness are considered which are weighted according to the values visible in the figure. This weighting vector is derived from the QA values given by a local, on-premises component, which should be replaced by a cloud offering. The weights for the QAs root node (0.8) as well as the application type node (0.2) are assigned statically.

For each QA an individual weight vector considering all cloud offerings is calculated by eigenvalue/eigenvector calculations. The individual QA weight vectors are then merged to build a matrix which can be multiplied with the weight vector got from the on-premises component to get a final vector considering all the QAs for each offering. For simplicity

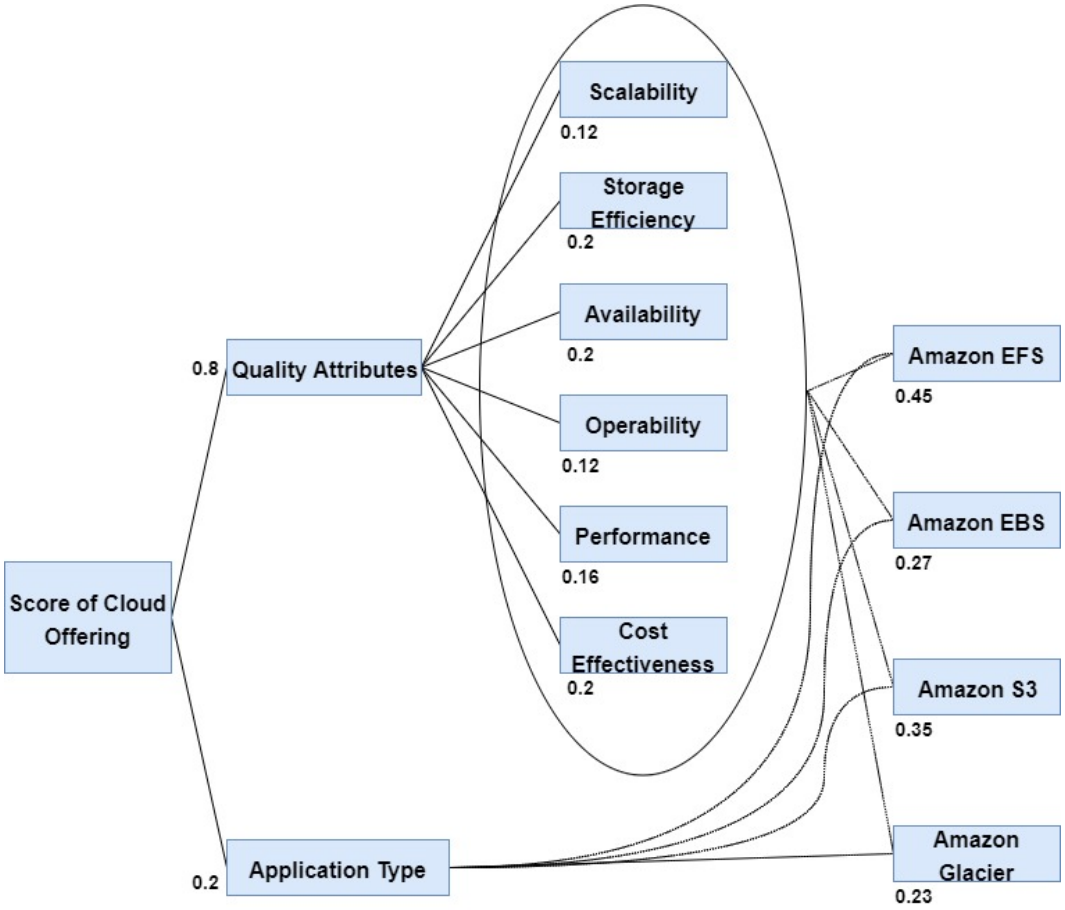


Figure 5.5: AHP example for ranking cloud offerings

reasons that is shown with the bubble around the QAs connected to each offering in Figure 5.5.

Now the QA weight vector can be merged with the application type vector into a new matrix. The application type vector could for example be calculated using the formula presented in Section 4.2.5. This matrix can be multiplied with static weights given for the QAs and the application type to get the final scores. The final scores after ranking based on this six QAs and the application type are given in Figure 5.5 for each cloud offering considered.

The ideas discussed in the short introduction based on the AHP example in Figure 5.5 are further elaborated in more detail based on the Algorithm 5.1. This algorithm shows the pseudo code for the core of the algorithm without the ranking by application type. So when looking at Figure 5.5 only the QAs part is discussed. The remainder of the algorithm, the ranking based on application type as well as the binning step, is left as exercise to the reader.

Algorithm 5.1: AHP algorithm for offering ranking

Input :

- A list N of all NFR values to consider.
- A list O_{in} of all offerings to consider.
- A list of lists L of NFRs for each offering of O_{in}.
- Length of O_{in} must be equal the length of L.
- Each entry of L must have equal length of N.

Output : A list O_{out} containing the score for each offering

```
1  eigenVectors := [];
2  foreach nfr in N do
       // Step 1: Create matrix A
3      A := identity matrix;
4      foreach a_ij in A do
5          a_ij := L_ij / L_ji;
6      end
       // Step 2: Calculate eigen vector for matrix A
7      eigenVect := CalcEigenVector (A);
8      eigenVect := NormalizeVector (eigenVect);
9      Add eigenVect to eigenVectors;
10 end
   // Step 3: Aggregation calculating final scores for each
   offering
11 importanceVect := NormalizeVector (N);
12 Populate B with entries from eigenVectors where eigenVectors_i is column i of B;
13 O_out := B × importanceVect;
```

An assumption of the whole algorithm is that the QAs fulfilled by some cloud offering (e.g. high scalability) as well as the required QAs of some component are already defined by an importance weight on a scale from 1-5, 5 meaning very important. A scale of the importance weight from 1-5 is used, because the importance weight for some component of a software architect's application is assigned by himself. It is easier for a human user of a decision support system to give a value in an integer scale such as the scale from 1-5 rather than for example using a continuous scale from 1-100. However, the algorithm would also work when using a continuous scale.

For each importance weight of a QA to consider given by N in Algorithm 5.1 the $n \times n$ matrix A is created (n is the number of offerings to consider) and the normalized eigenvectors are calculated (lines 2 to 10). In lines 3 to 6 the matrix A is populated by comparing the importance of a certain offering's QA currently considered (L_{ij}) with each

other (L_{ji}) by calculating L_{ij}/L_{ji} (i is the matrix's row and j its column). The matrix creation is quite similar to the example presented in the Satty paper [Saa90] in section 3 which is based on the comparison of n stones $A_1, ..., A_n$ with weights $w_1, ..., w_n$.

This is done for each QA which should be considered in the loop in the lines 2 to 10 which adds the calculated (line 7) and normalized (line 8) eigenvector into the $eigenVectors$ list. The matrix A is used for the eigenvalue/eigenvector calculation as defined for the AHP [Saa90]. The $CalcEigenVector$ function called in line 7 has the responsibility for calculating the eigenvectors and eigenvalues of the matrix A as well as choosing the eigenvector where the eigenvalue is maximal, called the principal eigenvalue of A. Furthermore, in line 8 a normalization of the calculated vector must be done by multiplying with a constant, the reciprocal of the vector's sum. By the means of its construction the matrix A fulfils the consistency condition of $A = (w_i/w_j), i, j = 1, ..., n$ defined by Satty [Saa90]. For a detailed explanation of this condition as well as a proof it is referred to the work of Satty. Since it is guaranteed by design that the consistency condition is fulfilled there is no need for checking for inconsistencies.

After the calculations in lines 2 to 10 are finished in an aggregation step in the lines 11 to 13 the final scores are calculated which are put into O_{out} where each score maps to each offering in O_{in}. In line 11 of this aggregation step a normalized importance vector is calculated using the list N of the QA values as input to the $NormalizeVector$ function which was already used in line 8 to get a normalized eigenvector by multiplying the whole vector with the reciprocal of the sum. In line 12 the contents of the $eigenVectors$ list are merged column-wise into a new matrix B which is used in line 13 to calculate the final score vector O_{out} using a cross product.

After the score vector O_{out} is calculated it can be used together with the application type weights vector already mentioned in the introduction, which is calculated by comparing the *application types* of the local, on-premises component and the cloud offerings. This vector can, for example, be calculated using the formula defined in Section 4.2.5 by taking the path length in the graph model between the component to consider and a cloud offering into consideration. A shorter path length gives a better score, if no path could be found the score is 0. Both score vectors can be again combined column-wise to a new matrix C which can be multiplied with a vector containing a vector of constants, e.g. static weights (as shown in Figure 5.5) to calculate a final score vector combining both the QA score vector as well as the application type score vector.

Since the scores consist of numbers between 0 and 1 a binning step is applied before returning the result mapping the scores onto a scale from 1-10. Such a binning step could be for example implemented using histogram calculations. This makes the scores easier understandable for the software architect.

5.3.2 Runtime Properties

The algorithm proposed in the previous section is an on-line algorithm, meaning that a user who starts the calculation will wait till the results are returned. Only after the

results are returned he can proceed with the next steps of his work, such as making MOVE or REPLACE decisions for the next components which again requires a sequential execution of the algorithm. Therefore a runtime experiment is needed to guarantee that the algorithm behaves fast also when using larger data sets.

Table 5.1: Runtime experiment data sets

Data Set	**Small**	**Medium**	**Large**	**Extra Large**
Number of Offerings	30	60	120	240
Number of Matches	10	20	40	80

Table 5.1 shows the setup of the runtime experiment to determine how the runtime evolves using increasing data sets. The set sizes (referred as *Number of Offerings* in Table 5.1) were chosen to correspond to instance sizes which could be expected in practise. The *Number of Matches* mean how much offerings are matched based on their capabilities. These matched offerings are input to the algorithm presented in the previous section inclusive matching via the application type hierarchy.

Besides the total time required for the whole algorithm, also an interim time after the matching step was calculated to see how much time is spent for the matching alone. Furthermore, all experiments were execute with and without using a cache which speeds up the whole algorithm due to not having to make requests to the knowledgebase. In this way also the effect of the cache should be examined. All experiments were executed using 30 iterations and taking the mean value of all iterations to reduce noise which could for example be introduced by the operating system. Moreover, a total number of 5 QAs were considered which also corresponds to real conditions in practise. This means that the outer loop of Algorithm 5.1 is executed a total of 5 times requiring 5 eigenvalue/eigenvector calculations on matrices of size 10×10, 20×20, 40×40, and 80×80.

For executing the experiments a low end machine (Intel Core i7 M 620 @ 2.67GHz) was used as execution environment and as software the prototypical implementation of the proposed solution was used. It is written in Python 3 using numpy for the eigenvalue/eigenvector as well as the matrix calculations (see Section 5.4 for more details).

In Figure 5.6 the results of the runtime experiment are graphically shown using a bar chart. When comparing the results with and without using a cache, it can be very well seen that using a cache speeds up the execution of the algorithm resulting in run times between $20 - 25\%$ lower than without using a cache.

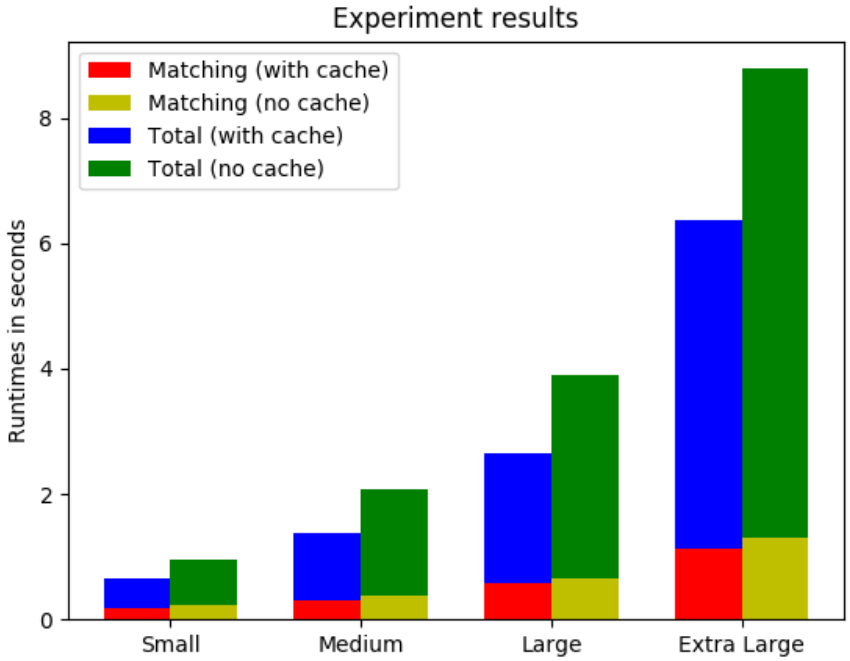


Figure 5.6: Runtime experiment results

Most of the runtime of the algorithm is spent for the ranking algorithm presented in the previous section including the eigenvector/eigenvalue calculations. The results would indicate that the run times increase linearly with the set sizes, for example a doubled set size means approximately the doubled run time. But the eigenvector/eigenvalue calculations indeed have an impact on the execution times and in this test cases only a total of 5 of them are required for each data set since the number of QAs considered is fixed. So it cannot be concluded that the algorithm's runtime really increases linearly with the set size. In fact there cannot be expected a linear runtime since the eigenvector/eigenvalue has an algorithmic complexity of $O(n^3)$ in practise similar to other matrix operations such as matrix multiplication or matrix inversion [CRLS09]. Using the Coppersmith–Winograd algorithm a complexity of $O(n^{2.376})$ can be achieved [CW87].

However, when looking at the total run times and considering that a low end device was used for the execution of these experiments, it can be concluded that the proposed algorithm behaves very well. For example, in a practical setting this algorithm would be executed in a cloud environment using some high end processor which means that only a few seconds response time can be expected also for larger data sets. Furthermore, also the use of a cache decreases execution times since in this way requests to the knowledgebase can be reduced. To further improve the runtime of the algorithm techniques such as approximation or pre-computing and caching of AHP results could be done similar as in

Google's PageRank algorithm [BP98, PBMW99].

5.3.3 Example: Replace an on-premises solution by a cloud native service

The process of how the matching and ranking works is now illustrated with an example. The hierarchy shown in Figure 5.5 is part of this example. For the example it is assumed that a cloud migration to the AWS was selected. The component to migrate is an on-premises Network Attached Storage (NAS) that uses the Network File System (NFS) protocol. It was chosen by the software architect to replace this component with some cloud offering. The NAS provides the capability 'Persistent Data Storage' which is used for filtering the available AWS offerings. Result of the overall filtering are the following offerings: 'Amazon Elastic File System (EFS)', 'Amazon Elastic Block Store (EBS)', 'Amazon Redshift', 'Amazon Simple Storage Service (S3)', 'Amazon DynamoDB', 'Amazon Aurora', 'Amazon SimpleDB', 'AWS Storage Gateway', 'Amazon Glacier', and 'Amazon Relational Database Service (RDS)'. These offerings are now input to the AHP algorithm defined in the previous section together with the component's (NAS) QA constraints as well as the offerings QA constraints. The QA constraints for the NAS can be seen in Table 5.2.

Table 5.2: QA constraints of the NAS component

QA Constraint	Importance
Scalability	3
Storage Efficiency	5
Availability	5
Operability	3
Performance	4
Cost	5

To simplify the example a bit in the calculation only the offerings 'Amazon Elastic File System (EFS)', 'Amazon Elastic Block Store (EBS)', 'Amazon Simple Storage Service (S3)', and 'Amazon Glacier' are considered. EFS's QA constraints can be seen in Table 5.3.

Table 5.3: QA constraints of the EFS cloud offering

QA Constraint	Importance
Scalability	5
Storage Efficiency	4
Availability	5
Operability	5
Performance	4
Cost	2

EBS's QA constraints can be seen in Table 5.4.

Table 5.4: QA constraints of the EBS cloud offering

QA Constraint	Importance
Scalability	4
Storage Efficiency	4
Availability	1
Operability	5
Performance	4
Cost	1

S3's QA constraints can be seen in Table 5.5.

Table 5.5: QA constraints of the S3 cloud offering

QA Constraint	Importance
Scalability	5
Storage Efficiency	4
Availability	4
Operability	5
Performance	3
Cost	4

Glacier's QA constraints can be seen in Table 5.6.

Table 5.6: QA constraints of the Glacier cloud offering

QA Constraint	Importance
Scalability	1
Storage Efficiency	1
Availability	1
Operability	5
Performance	1
Cost	5

Note that an offering can also have more than the QA constraints which are mentioned here, but since they are not important for the component to examine they are not considered. Some of the QA constraints may also be not present for a certain offering. As a default the value 1 is used.

So for Performance the 4*4 matrix shown in Figure 5.7 can be constructed (the rows/-columns belong to the following offerings: EFS, EBS, S3, and Glacier).

Now using this matrix the following eigenvector can be calculated: [0.356, 0.356, 0.267, 0.089] A normalized vector can be calculated by dividing through the sum of this vector

$$\begin{pmatrix} 1 & \frac{4}{4} & \frac{4}{3} & \frac{4}{1} \\ \frac{4}{4} & 1 & \frac{4}{3} & \frac{4}{1} \\ \frac{3}{4} & \frac{3}{4} & 1 & \frac{3}{4} \\ \frac{1}{4} & \frac{1}{4} & \frac{1}{4} & 1 \end{pmatrix}$$

Figure 5.7: Comparison matrix for Performance.

(1.069): [0.333, 0.333, 0.25, 0.083]. This is done in the same way for the other QA constraints. The results can be seen in Table 5.7.

Table 5.7: Eigenvectors of all QA constraints considered.

QA Constraint	**Eigenvector**	**Normalized Eigenvector**
Scalability	[0.61, 0.49, 0.61, 0.12]	[0.33, 0.27, 0.33, 0.067]
Storage Efficiency	[0.57, 0.57, 0.57, 0.14]	[0.31, 0.31, 0.31, 0.077]
Availability	[0.76, 0.15, 0.61, 0.15]	[0.46, 0.09, 0.36, 0.09]
Operability	[0.5, 0.5, 0.5, 0.5]	[0.25, 0.25, 0.25, 0.25]
Performance	[0.62, 0.62, 0.46, 0.15]	[0.33, 0.33, 0.25, 0.08]
Cost	[0.29, 0.15, 0.59, 0.74]	[0.17, 0.08, 0.33, 0.42]

In a next step these vectors are merged column-wise to get another matrix which is multiplied with the importance vector [0.12, 0.2, 0.2, 0.12, 0.16, 0.2] constructed from the importance values of the QA constraints of the NFS component. The final result of this multiplication is the vector [0.309, 0.21, 0.311, 0.17].

For the application type hierarchy the vector [1.0, 0.5, 0.5, 0.5] can be obtained. The type hierarchy is shown in Figure 5.8. EFS gets the score one, the best score because it is of the same type as the NAS component. All other offerings get a lower score because they are one level away in the type hierarchy.

Now by combining the QA as well as the type hierarchy scores to a new matrix and multiplying them with a static weight vector [0.8, 0.2] the final result vector [0.45, 0.27, 0.35, 0.23] can be obtained. By applying binning based on histogram calculations the final (human readable) scores are obtained which can be seen in Table 5.8.

Table 5.8: Ranked offerings with final score.

Rank	**Cloud Offering**	**Score**
1	EFS	10
2	S3	6
3	EBS	2
4	Glacier	1

As it can be seen EFS is best suitable to replace a local NAS since it is a distributed file

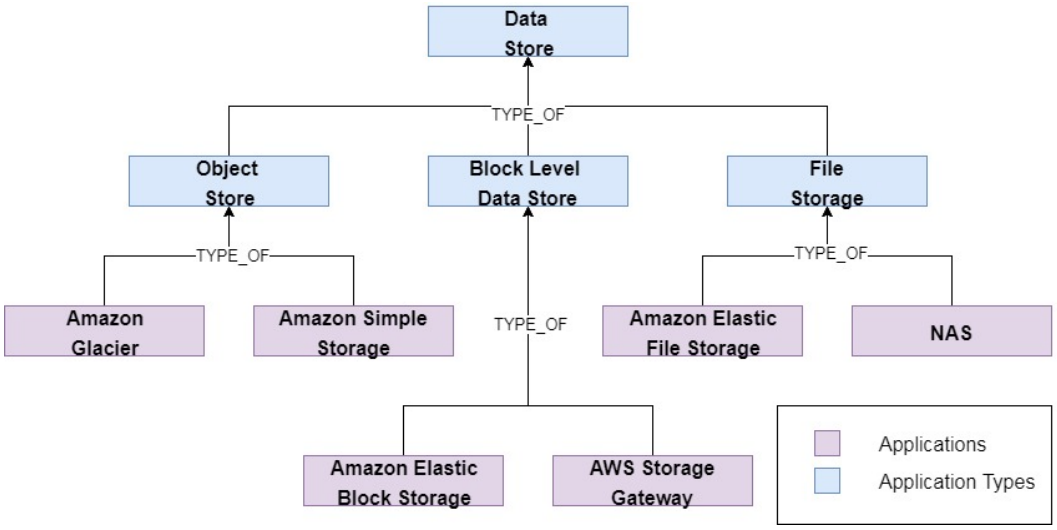
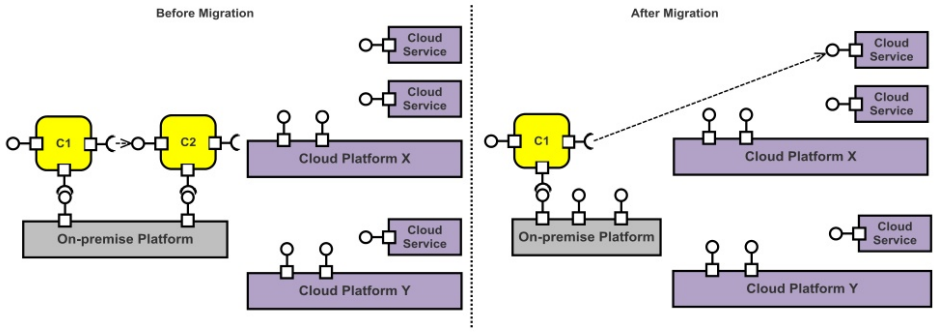


Figure 5.8: Application type hierarchy for NAS component and cloud offerings.

system. S3 is also a good choice since costs may be saved compared to EFS. Glacier on the other hand is not a good replacement since it has bad Availability and Performance due to being a backup storage solution with low performance.

After an offering is selected the software architect is supported by the selection of a suitable cloud migration pattern. In this work they are taken from the work of Jamshidi et al. [JPCL15]. Since the hybrid cloud migration option as well as the replace action where chosen one of the following patterns in Figure 5.9 can be chosen.



(a) Simple Replace

(b) Replace with on-premises adaptation

(c) Replace with cloud adaptation

Figure 5.9: Replace cloud migration patterns [JPCL15]

5.4 Prototypical Implementation

This section briefly discusses the implementation of the concepts discussed in this chapter which was required for doing the evaluation. Figure 5.10 shows the architecture of the proposed solution as discussed in Section 5.2 including the technologies used.

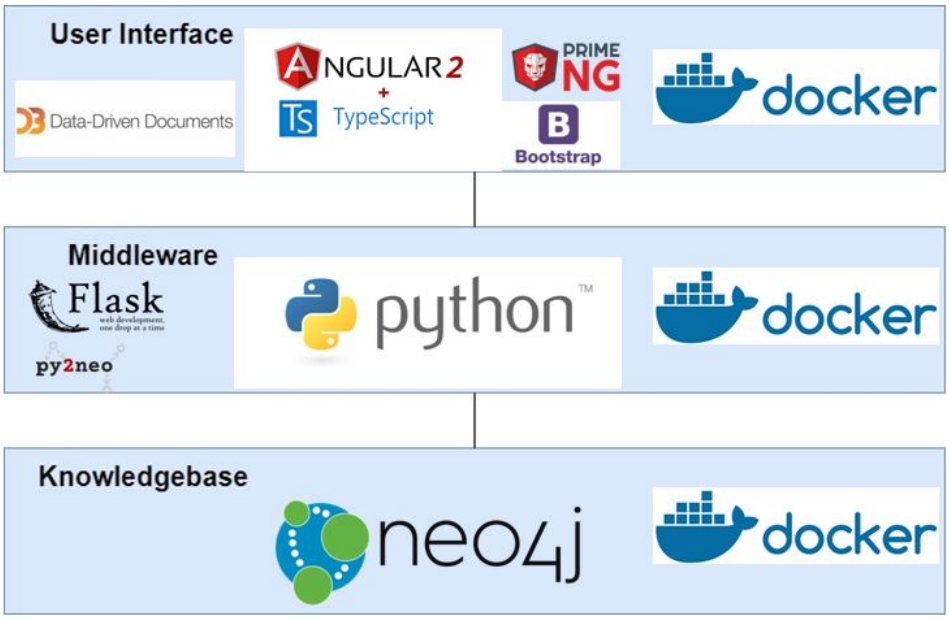


Figure 5.10: Architecture of prototypical implementation with technologies

The *Decision Support Workflow Frontend* was implemented using Angular 2 including Bootstrap for layouting and PrimeNG which provides user interface components such as tables and lists. In addition, Data-Drive Documents (D3) was used for drawing the architecture graph view with the components and connections.

The *Middleware for Decision Support Services* was implemented using Python 3. Flask was used to provide the RESTful connectivity to the frontend, Numpy for numeric calculations such as the eigenvalue/eigenvector or the matrix calculations required for the AHP algorithm presented in Section 5.3, and Py2Neo was used to provide connectivity to the Neo4J database used as knowledgebase. All the components support deployment with Docker which for example enables the execution of them on Amazon's Elastic Container Service (ECS) [Amae]. The RESTful endpoints were documented using Swagger (via the Flask RestPlus extension) and for user authentication Java Web Tokens (JWT) were used.

In Figure 5.11 a package diagram of the implemented middleware can be seen. As it can be seen the packages map to the middleware services presented in Section 5.2.2. The *service.py2neo_wrapper* package contains the implementation of the *knowledgebase access service*, the *data import service* is realized within the *service.data_import* package, the

knowledge service is contained in the *service.knowledge* package, and the *cloud platform service* is implemented in the *service.offerings* package. The *service.local_application* package contains the *local application service*, the *decision support service*, as well as the *matching* and the *ranking services*. Furthermore, besides the services there are also the resources (*resources.knowledge*, *resources.local_application*, and *resources.offerings*) which contain the RESTful endpoints. The RESTful endpoints are grouped using different name-spaces such as names-spaces for *applications*, *capabilities*, *QAs*, or *software components*.

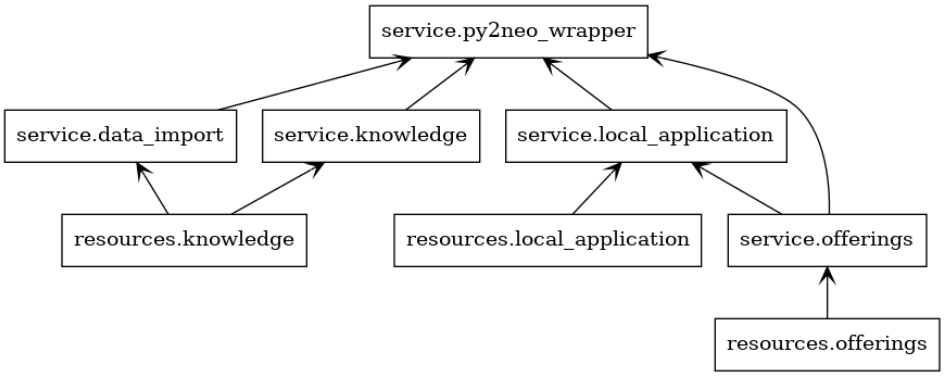


Figure 5.11: Packages Decision Support backend

Since a Neo4J database is used as knowledgebase, the usage of its query language Cyper was needed multiple times. Besides simple queries required for the CRUD functionality, also more complex queries were employed. The query in Listing 5.1 was used in the capability matching function to retrieve all business capabilities of a component given its *componentUuid*.

```
1 MATCH (component:Application {uuid: '$componentUuid'})
2     <-[:INSTANCE_OF]-(localApp:Local)
3     -[:HAS*2..2]->(localBusinessCap:Local)
4     -[:INSTANCE_OF]->(businessCap:Functionality)
5 RETURN businessCap
```

Listing 5.1: Example Cyper query to get a component's capabilities

This query simply consists of a MATCH and a RETURN statement returning all capabilities found. The component itself is an Application as defined by the Knowledge Entity Meta Model [PSZ18]. The INSTANCE_OF relations are used to mark the border between the Application Description and the Knowledge Entity Meta Model. The Application Description Model mainly consists of Local nodes which are connected to other Local nodes via a HAS relation and via INSTANCE_OF relations to Knowledge Entity Meta Model instances such as Functionality in this example.

CHAPTER 6

Architecture Decision Support Evaluation

The following chapter describes how the implemented tool was evaluated. An experiment in an industrial context was used for that purpose which will be described in the following sections. Furthermore, it will be described how data was acquired and how this data was analysed also referring to threats to validity as defined by Wohlin et al. [WRH+12].

6.1 Experiment Setup

The implemented tool was evaluated with ten professional software architects in an industrial context at Siemens. A cross over design was used for the experiment splitting the participants into two groups. One group worked with the tool and the other group, the *control group*, worked with the traditional, manual approach. The manual group was supported by an answer sheet and some supplementary material containing all offerings of a cloud platform together with a description of their purpose. Using this instrumentation the information given was tailored to the decisions defined in Section 4.1.1 in order to be comparable with the results of the tool. All the material used for instrumentation, also including the task description sheets of the two use cases, were reviewed by a colleague not participating in the experiment in advance to guarantee its understandability and soundness.

At the beginning of the experiment the tool was introduced using a short live demo showing all concepts of the workflow the tool is built on and how it should be worked with the tool. Two different use cases which will be discussed in the next section in more detail were provided. The participants worked on each use case for a total of 45 minutes with one of the groups working with the tool and the other group working with the manual approach. For the second use case the roles were swapped.

6.1.1 Feedback Questions

After both sessions were finished the participants were encouraged to answer a questionnaire giving feedback about the tool and the workflow as well as stating what experience they have in the field this experiment is placed in. The following feedback questions were asked:

- *FQ1*: Do you think such a tool assisted workflow is usable in your projects?
- *FQ2*: Do you think such a workflow is an improvement compared to your current approach?
- *FQ3*: Do you think the workflow covers questions/problems appearing in your daily work?
- *FQ4*: Do you think the workflow is intuitive?

A scale including the answers *Strongly disagree*, *Disagree*, *Agree*, and *Strongly Agree*, which can be easily mapped onto a numeric scale, was used. A scale only consisting of four answer options was chosen to minimize the variability of the answers given and to differentiate between equally distributed positive and negative answer options. Furthermore, some space was left to give further remarks such as giving reasons, especially in case of negative answers and additional comments. This helps to get insights about the reasoning behind the answers given.

The general motivation for the feedback questions is to get subjective feedback from each participant which should help, together with the qualitative analysis of the results, to answer all goal questions defined for this experiment (see Section 6.1.3). *FQ1* in particular should find out if such a tool assisted workflow developed as part of this work is usable in practise at all. Using *FQ2* the participants' subjective impression about the question if the workflow is an improvement compared to their current, manual approach should be captured. This can be compared with the results of the experiment if their answers agree with the actual results. *FQ3* gives insights on how relevant the questions and problems covered by the workflow are in a participant's practical work. Negative answers would indicate that more research is required to further improve the presented workflow. Finally, *FQ4* should find out how intuitive the workflow is regarding a participant's subjective feeling. Having an intuitive workflow helps a decision support tool based on this workflow to be accepted by potential users.

6.1.2 Experience Questions

For a participant's experience the following questions were asked:

- *EQ1*: How much experience do you have with cloud/edge computing in general?

- *EQ2*: How much experience do you have with AWS (including knowledge of at least some of the available offerings)?
- *EQ3*: How much experience do you have with architectural design problems in general, such as architecture refactoring or reconfiguration?

Here a scale including *None*, *Moderate*, and *Expert* was used which can be again easily mapped onto a numeric scale. Similar as for the feedback questions only a limited set of answers was chosen to minimize the variability of the answers.

Main motivation for having experience questions is that they enable the correlation between the results of the experiment and the experience for each participant. In particular *EQ1* was chosen since the proposed workflow targets cloud-edge environments which means that experience or a lack of experience with cloud/edge computing is an important consideration when interpreting a participant's results. Since one use case also includes an application partially deployed to AWS, also experience with AWS and knowledge about the properties of its offerings could influence the outcomes of the experiment (see *EQ2*). Especially for the participants working with the manual approach this could explain better results compared to participants having less experience with this cloud platform. Finally, since architectural design problems are considered as use cases in this experiment, experience with similar problems could also explain better results compared to participants having less experience (see *EQ3*). So *EQ3* helps to consider this as well in the evaluation analysis.

6.1.3 Goals of the Experiment

The following questions should be answered by the experiment:

- *GQ1*: Is a solution found with the tool better than a solution found with the traditional approach (regarding quality and soundness)?
- *GQ2*: Is a tool assisted workflow an improvement compared to the traditional approach?
- *GQ3*: Do participants with more experience perform better?
- *GQ4*: Is a solution found with the tool more innovative compared to solutions found with the traditional approach, e.g. the tool helps to find solutions which would not have been found using the traditional approach?

All of the presented goal questions have in common that they can be answered positively, negatively or none of both. The third option means that further research must be done, e.g. conducting further experiments to get an answer for this question.

GQ1 describes the primary goal of this evaluation to find out if the presented tool-assisted workflow is better compared to the traditional approach. This question should

be answered by analysing the results of the experiment regarding quality and soundness of the solutions found, e.g. by comparing the number of errors made using each approach.

GQ2 is an extension of the first question which should incorporate into the analysis besides the qualitative results also the answers obtained by the feedback questions. Here especially the feedback question *FQ2* is of importance.

GQ3 should also incorporate the experience of the participants by comparing it with their qualitative results. So it should find out if experience is also of importance when considering architectural decision problems such as the problems used as evaluation use cases.

Finally, *GQ4* should examine if the tool also enables to find solutions which are more innovative compared to solutions found by human decision makers using the traditional approach. A positive answer to this questions would point out the benefits of the matching and ranking facilities of the decision support tool presented in this work.

6.1.4 Measurement Design and Data Analysis

The results of the experiment were mostly validated qualitatively, especially regarding their *soundness* compared to a reference solution. To get a quantitative metric the number of errors in a participant's solution can be counted. An example for an error would be violating a hardware constraint such as deploying a component requiring GPU computing support into a cloud environment which does not provide this capability. But it must be said that a solution may also contain no error if it does not stick to the reference solution since for each use case multiple solutions are possible. Other quantitative analysis techniques do not make that much sense due to the limited number of participants. So statistical analysis techniques are only used for analysing the results of the questionnaire.

It was decided to not measure *time* in that detail since results would not be that useful because the participants working with the manual approach were instrumented as mentioned in Section 6.1 which would not be the normal case in practice. Furthermore, the time a user needs may vary due to his experience such as experience with solving similar problems or experience with similar tools. So for the factor *time* only an absolute time limit of 45 minutes was given. Therefore, the *effort* cannot be measured using the the *time* a participant needs to solve a use case. However, some kind of a subjective *effort* can be retrieved by examining the results of *FQ2* (see Section 6.1) to find out if the participant thinks the workflow is an improvement compared to the manual approach.

As already mentioned in this section quantitative results are obtained by a questionnaire (with the exception of the number of errors found in a solution). This enables the application of descriptive statistics such as the calculation of mean, min and max values, or the calculation of a correlation between feedback and experience. Furthermore, the question if participants with more experience perform better can be answered.

6.1.5 Threats to Validity

Wohlin et al. [WRH+12] described for types of validity threats, namely threats to the *internal validity*, *external validity*, *construct validity* as well as *conclusion validity*. The following section will mention some threats relevant to this experiment referring to the terms defined by Wohlin et al. as well as how they are prevented or their effects are minimized.

Internal Validity

Internal validity is the most important type of validity for applied research where also this works belongs to. The following threats were identified:

- Two completely different use cases were chosen to prevent *learning effects* among the participants which could bias the results of this experiment.
- The participants were split into two groups having a *control group* to prevent various *single group threats* such as *maturation effects* due to learning from past sessions using the tool, e.g. if the whole group is using the tool in a first session as well as the manual approach in a second session to solve the same problem. All participants used the tool only once the first time.
- Instrumentation artifacts were reviewed by a colleague to prevent *instrumentation* threats due to badly designed artifacts affecting the experiment negatively.
- Only volunteers were selected for the experiment, e.g. people were not forced to take part into the experiment to prevent the *selection* threat mentioned by Wohlin et al. [WRH+12].
- To mitigate the risk that participants working with the manual approach perform not as good as they generally would (the *restful demoralization* threat), making the results useless, appropriate instrumentation was used.

External Validity

Also the generalization of results, e.g. from the context of this experiment to a wider context is important in applied research. Therefore the following threats were identified:

- To prevent the *interaction of selection and treatment* threat a subject population representative of the population it will be generalized to, e.g. software architects, was encouraged to take part in the experiment. Furthermore, to find out if there are participants having not that much experience in the field this experiment is placed in, their experience is queried in the questionnaire. So bad results can be correlated with bad experience.

- To prevent the *interaction of settings and treatment* threat problems originated from real projects are chosen as use cases instead of toy problems to get complex enough architecture problems.

Construct Validity

Compared to internal and external validity, the construct validity is less important in applied research. In this work no threats to the construct validity could be identified. For example, since the quality of the results do not depend on a single quantitative metric but they are interpreted qualitatively, no *mono-operation bias* can occur.

Conclusion Validity

Construct validity is the least important in applied research. Furthermore, only few and simple statistical methods are used to analyse the results. Only the *Random heterogeneity of subjects* threat could be relevant for this experiment describing the risk of variation due to individual differences in the experience of the participants. The effects of this threat are minimized by also querying a participant's experience in the questionnaire.

6.2 Use Cases

The following section discusses the two use cases used in the experiment. The discussion starts with a description of each use case, then the goals are mentioned and finally a reference solution used for validating the results is presented.

6.2.1 Infrastructure Monitoring System (IMS) Use Case

The architecture of the first use case, called Infrastructure Monitoring System (IMS) use case, was taken from a geospatial data handling application developed at Siemens. All components are hosted on-premises and as much of them as possible should be deployed into a cloud environment. So this is a classical cloud migration scenario. Figure 6.1 shows the current application with its components and links between the components. The application architecture consists of the following components:

- A *Web Dashboard* for data visualization and workflow creation which is written in PHP and JavaScript
- An *Analytics* component doing computer vision calculations implemented on GPUs using CUDA written in C++
- A *Network Attached Storage (NAS)* for file storage
- A *Workflow Engine* written in C++ using the Qt framework hosting a Web Socket server

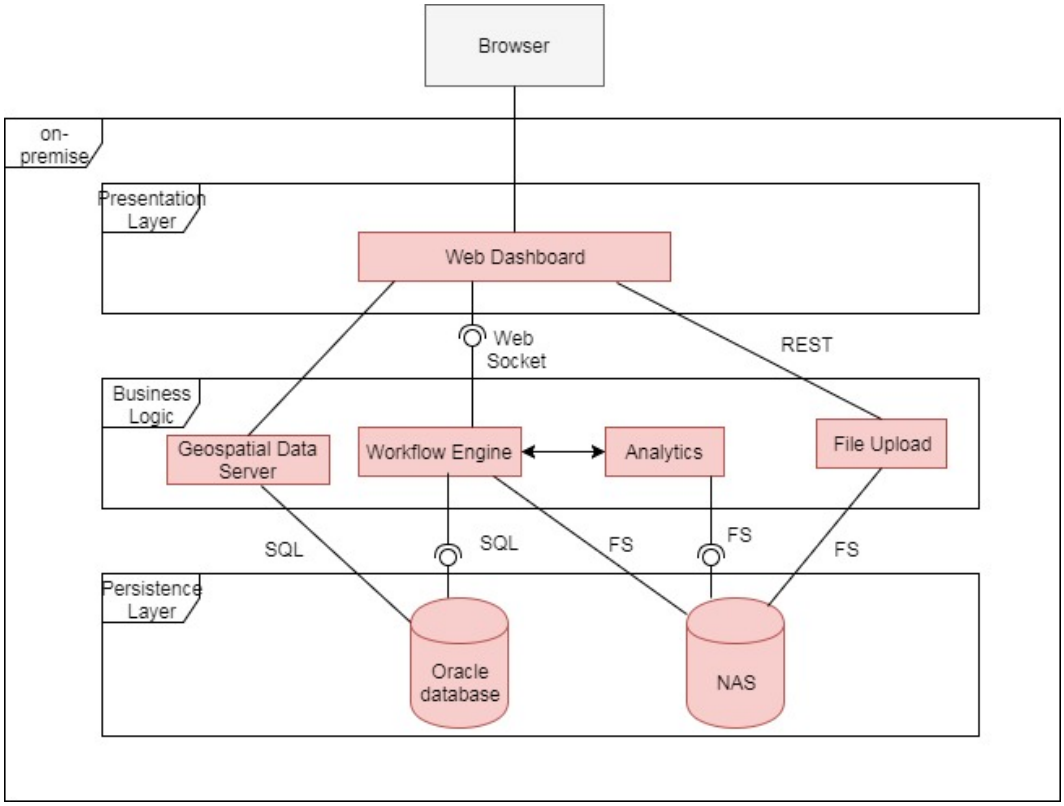


Figure 6.1: IMS use case application architecture

- A *Geospatial Data Server* (GeoServer) written in Java
- A *File Upload* component written in Java accessible via a REST interface
- An *Oracle database* for storing additional data such as user administration data, workflow data, etc.

Constraints

The deployment scenario as described in Section 4.1.1 was limited to *hybrid cloud* deployment and as cloud platform Siemens MindSphere [Sie] was fixed. MindSphere is a PaaS building on Cloud Foundry [Foub]. Furthermore, as constraints for the cloud platform it was given that only the programming languages Java, JavaScript, and PHP are supported. Moreover, the *PostgresSQL* cloud offering does not support geospatial data, so e.g. the *Oracle database* cannot be easily replaced by the *PostgresSQL* cloud offering. Both points must also be considered by the participants. Table 6.1 shows all offerings the cloud provides. This information was given in printed form to the participants doing the manual approach as instrumentation.

Table 6.1: MindSphere Cloud Offerings.

Compute	*Area*	*Service*	*Description*
	PaaS	Cloud Foundry	Managed hosting platform providing easy to use services for deploying and scaling web applications and services. This service provides no GPU computing support. Supported programming languages: Java, JavaScript, PHP
Storage	*Area*	*Service*	*Description*
	Relational database	PostgreSQL	Relational database-as-a-service (DBaaS) where the database resilience, scale, and maintenance are primarily handled by the platform.
	NoSQL - document storage	MongoDB	A highly scalable document oriented database managed by the platform.
	Shared file storage	File System as a Service	Provides a simple interface to create and configure file systems quickly, and share common files. It's shared file storage without the need for a supporting VM and can be used with traditional protocols that access files over a network.
	Object storage	Object Store	Object storage service, for use cases including cloud applications, content distribution, backup, archiving, disaster recovery, and big data analytics.
	NoSQL - key/value storage	Redis	An in-memory database service implementing a distributed, in-memory key-value store with optional durability.
Messaging	*Area*	*Service*	*Description*
	Message Queueing, Pub/Sub Messaging	RabbitMQ	A managed RabbitMQ service with high scalability.

Goals

The following goals for this use case were given:

- Deploy as much as possible into the cloud (MindSphere only) without violating constraints to
  - Increase the processing capabilities (performance, scalability)
  - Use scalable data storage (e.g. managed data storage)
  - Bring the UI into the cloud
- Limit the need for implementing new components as much as possible
- Make sure that all components are still reachable via the network, e.g. connecting a cloud client to a server running on-premises is not possible due to firewall restrictions.

Reference Solution

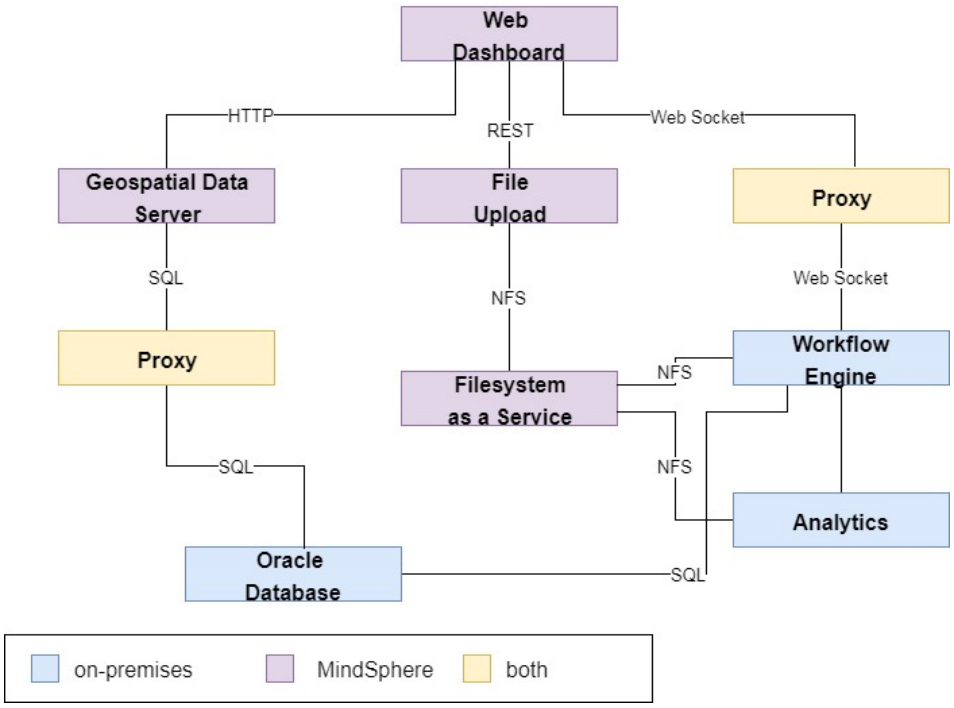


Figure 6.2: IMS use case reference solution

In Figure 6.2 the reference solution for the IMS use case used for validating the participants' results is shown. As it can be seen in this figure the *Web Dashboard*, the *Geo Spatial*

Data Server, the *File Upload* components are deployed into the cloud. They are hosted using the PaaS capabilities of MindSphere, e.g using Cloud Foundry. Furthermore, the *NAS* is replaced by the *Filesystem as a Service* offering.

The *Workflow Engine*, the *Analytics* component, as well as the *Oracle Database* are left on-premises. Deployment of the *Workflow Engine* into the cloud is not possible due to a incompatible programming language (C++). The *Analytics* component cannot be deployed into the cloud because it lacks GPU computing support. Finally, the *Oracle Database* can neither be replaced by some cloud offering nor it can be deployed into the cloud.

In order to fix possible communication problems between an on-premises server and a cloud client *Communication Proxy* components are added between the *Web Dashboard* and the *Workflow Engine* as well as between the *Geospatial Data Server* and the *Oracle Database*.

6.2.2 DevOps Use Case

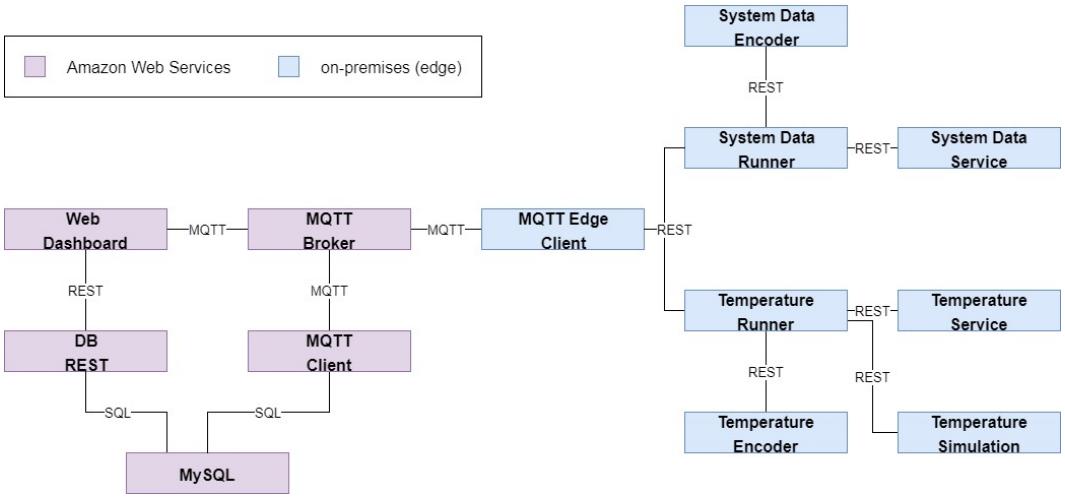


Figure 6.3: DevOps use case application architecture

The architecture for the second use case, called the DevOps use case, was also taken from a Siemens project. Figure 6.3 shows the architecture of the current application with its components and links between the components (the violet components are deployed on AWS and the blue components are deployed on-premises (edge, e.g. Raspberry Pi)).

The application described by this architecture is a cloud-edge application with some parts already deployed to AWS. It consists of a simple data acquisition, processing, storage and visualization use case. It already uses the micro services architectural style that structures the application as a collection of loosely coupled services. All components support deployment via Docker. The application consists of the following components:

Cloud (AWS)

- A *Web Dashboard* for data visualization written in JavaScript deployed on Elastic Container Service (ECS)
- The *DB-REST* component for database access via a REST interface deployed on ECS
- A self-hosted *MySQL database* deployed on ECS
- A *MQTT Broker* deployed on ECS
- A *MQTT Client* deployed on ECS writing data received from the *MQTT Broker* into the *MySQL database*

Edge (on-premises)

- A *MQTT Edge Client* sending data collected by the *Temperature Processing* and the *System Data Processing* Layer to the cloud (*MQTT Broker*)
- The *Temperature* and *System Data Processing* Layer consists of processing components such as encoders
- The *Temperature* and *System Data Acquisition* Layer consists of components for temperature and system data (e.g. CPU, RAM usage) collection (from sensors or simulated)

Constraints

In this use case no special constraints were given. For the description of the AWS offerings which was used as instrumentation for the participants working with the manual approach the information found at [Mica] (excluding the Azure offerings) was used.

Goals

The goal of this use case is to address the following issues:

- On the AWS side scalability and operability of the services may not be that good due to suboptimal decision made regarding selection of available cloud offerings.
- The MQTT Client on the cloud side must handle both system data and temperature data processing which requires the MQTT Client to be refactored even if only one of the two data formats changes. During the redeployment of the refactored MQTT Client, which causes a short down time, data values of both temperature and system data cannot be collected. So it would be beneficial if the temperature and system data processing capabilities would be separated like it is done on the edge side. Additionally an appropriate cloud offering should be used to benefit from available cloud offerings.

Reference Solution

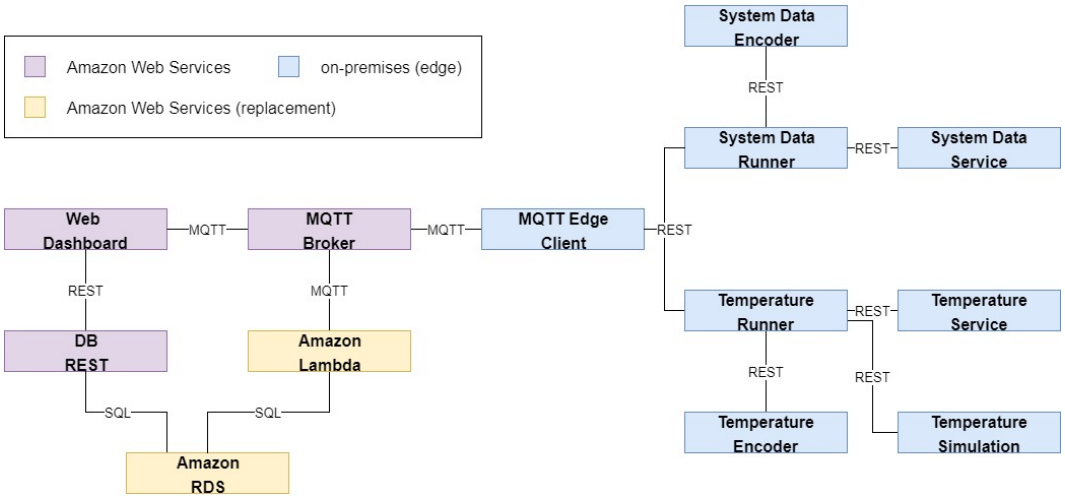


Figure 6.4: DevOps use case reference solution

In Figure 6.4 the reference solution for the DevOps use case can be seen. Changes compared to the architecture at the beginning (see Figure 6.3) are highlighted in yellow. As it can be seen in this Figure the edge (on-premises) part is left unchanged. On the cloud side the *self hosted MySQL database* was replaced by Amazon's Relational Database Service (RDS) to fix the first issue. In order to fix the second issue the *MQTT Client* is replaced with *AWS Lambda* to split up the temperature as well as the system data processing parts using a separate script hosted on AWS Lambda for each capability.

6.3 Results

This section presents the results of the experiment. At first the results of the questionnaire are examined starting with the feedback questions and also incorporation additional textual feedback. Then the results of the questions regarding the experience of the participants are examined. After that follows the qualitative analysis of the results for each use case comparing the results with and without tool support. Finally, this section is concluded by a discussion about the results of the use cases and the questionnaire.

6.3.1 Feedback

For the feedback questions and their identifiers it is referred to Section 6.1.1. A numeric mapping of the feedback scale defined in Section 6.1.1 was used for the analysis which can be seen in Table 6.2.

Table 6.2: Feedback scale to numeric value mapping.

Feedback	Value
Strongly agree	4
Agree	3
Disagree	2
Strongly disagree	1

Table 6.3: Results of the feedback questions for all participants.

Participant	FQ1	FQ2	FQ3	FQ4	Overall Feedback	Variance
user-01	2	1	1	3	1.75	0.916
user-02	2	3	2	4	2.75	0.916
user-03	4	3	3	3	3.25	0.250
user-04	4	-	2	3	3.00	1.000
user-05	4	3	2	3	3.00	0.666
user-06	3	3	3	3	3.00	0.000
user-07	3	4	4	3	3.50	0.333
user-08	3	3	2	3	2.75	0.250
user-09	3	3	2	3	2.75	0.250
user-10	2	2	2	2	2.00	0.000

Table 6.4: Average and variance per feedback question.

Feedback Question	Average Value	Variance
FQ1	3.00	0.667
FQ2	2.78	0.694
FQ3	2.30	0.678
FQ4	3.00	0.222

Table 6.3 shows the feedback each participant gave and Table 6.4 shows the averages calculated for each feedback question. So overall it can be said that there was positive feedback. So most of the participants think that such a tool assisted workflow could be usable in their projects (*FQ1*) as well as that the workflow is intuitive (*FQ4*). Furthermore, a majority of the participants believe that such a workflow is an improvement compared to the traditional, manual approach (*FQ2*). Only few participants think that the workflow covers questions and problems appearing in their daily work (*FQ3*). So further research must be done, e.g. by doing a survey, to find out what additional questions and problems are appearing in their daily work which such a workflow could also solve.

The textual feedback indicates that most of the negative feedback is related to the usability of the tool, especially since some of the features did not work properly, e.g.

an undo of decisions made. Furthermore, one participant suggested that changes in the architecture should be visible dynamically in an WYSIWG style as well as that the graphical representation of the architecture should be always visible.

6.3.2 Experience

The mapping of the experience scale defined in Section 6.1.2 onto a numeric scale can be seen in Table 6.5. The results of the experience questions can be seen in Table 6.6. Table 6.7 shows the average experiences among all participants calculated for each question. The *Overall Experience* value in Table 6.6 is the average value of the three answers. Here it can be seen that most of the participants have *moderate* experience. There was only a single participant being *expert* in all fields this experiment is placed in as well as one participant having *none* experience with architectural design problems.

Table 6.5: Experience scale to numeric value mapping.

Experience	**Value**
None	1
Moderate	2
Expert	3

Table 6.6: Results of the experience questions for all participants.

Participant	**EQ1**	**EQ2**	**EQ3**	**Overall Experience**	**Variance**
user-01	1	2	1	1.33	0.33
user-02	1	1	2	1.33	0.33
user-03	2	2	2	2.00	0.00
user-04	2	1	1	1.67	0.33
user-05	2	2	2	2.00	0.00
user-06	2	2	2	2.00	0.00
user-07	3	2	3	2.67	0.33
user-08	2	2	3	2.33	0.33
user-09	1	1	1	1.00	0.00
user-10	3	3	3	3.00	0.00

Table 6.7: Average and variance per experience question.

Experience Question	**Average Value**	**Variance**
EQ1	1.9	0.54
EQ2	1.8	0.40
EQ3	2.1	0.67

6.3.3 Qualitative Analysis of the Results

IMS Use Case - Tool Supported

Two of the participants using the tool got really close to the reference solution (*user-04* and *user-05*), one of them finding it exactly (*user-04*). The other three participants (*user-01*, *user-09*, and *user-10*) had problems using the tool and did not produce good results. Table 6.8 shows an overview of the issues found per participant. Since not usable results were produced by the participants *user-09* and *user-10*, no number of issues could be calculated.

User-09 complained in the feedback that the back button (undo functionality) was not working. So this would explain the not usable results. Furthermore, the experience questions also showed that he has not that much experience in the fields this experiment is placed in. *User-01* and *User-10* gave bad feedback about the tool which seems to be related to their bad performance. Furthermore, it must be said that this first use case is quite complex which may have caused the bad performance of some participants.

Table 6.8: Number of issues found per participant

Participant	Issues
user-01	Not rated
user-04	0
user-05	0
user-09	Not rated
user-10	2

IMS Use Case - Manual

The results of the manual approach showed that three participants decided to replace the *Oracle database* with *PostgresSQL*. But here they clearly overlooked that *PostgresSQL* does not support geospatial data out of the box which is the reason why this solution would not work in practice. Such an issue could not be found in the results with tool support, because in the knowledgebase, the tool uses, it is explicitly encoded that this is not supported. Therefore the *Oracle database* cannot be replaced and should be left on premise.

Table 6.9 shows the number of issues found for each participant. For *user-02* no issues at all could be found. He only decided to replace the *Workflow Engine* by a newly implemented component using Java to leverage the scalability opportunity of the cloud. Besides *user-02* also *user-07* decided to replace the *Workflow Engine* by a newly implemented Java component. It was decided to not count this as an error since it is a legitimate decision. However, when making such a decision it must always be considered how much effort such a change would mean for the developers and if it is too high such a decision could also be postponed to a later iteration. The participants' *user-06*, *user-07*, and *user-08* solutions had one issue due to replacing *Oracle database* with *PostgresSQL*

as already discussed at the beginning of this section. *User-08* wants to implement the communication proxy using *RabbitMQ* which is not counted as an issue since it was not required for a valid solution to explain how the proxy should be implemented.

Finally, *user-03*'s solution had the most issues when working with the manual approach. He moved the *Workflow Engine* into the cloud although its programming language (C++) is not supported there. Furthermore, he wants to replace the *File Upload* component with the *Filesystem as a service* cloud offering although their capabilities do not match in any point. Finally, he wants to replace the *NAS* as well as the *Oracle database* with a *MongoDB* which would not be a very good idea. Using a tool supported workflow as it is presented in this work would have mitigated the first two issues. In addition, it would also have prevented him from replacing the *Oracle database* due to the missing geospatial data support of all the data storage cloud offerings. It would not have prevented him from replacing the *NAS* with *MongoDB*, but the tool would at least indicate that this is not a good idea by giving the MongoDB offering a bad score.

Table 6.9: Number of issues found per participant

Participant	**Issues**
user-02	0
user-03	4
user-06	1
user-07	1
user-08	1

DevOps Use Case - Tool Supported

Similar as in the first use case, two participants could find the reference solution (*user-03*, and *user-08*). Interesting to see is that *user-03*, who badly performed without the tool, submitted one of the best solutions when using the tool. This really shows the benefits of using the tool compared to the traditional approach.

Only one of the participants working with the tool had problems using the tool (*user-02*). They are mostly related to the issue with the undo functionality not working. However, by deeper analysis of his results a usable solution could be retrieved which gets close to the reference solution.

The other two participants also performed quite well. *User-06* just had forgotten to substitute the self hosted *MySQL database* with some cloud offering. *User-07* came really close to the reference solution, but he additionally replaced the *MQTT Broker* in the cloud with *AWS Lambda*. This is not counted as an error since the MQTT broker could indeed be replaced by AWS Lambda when hosting a script handling the brokerage of messages. Table 6.10 shows an overview of issues found for the participants working with the tool.

Table 6.10: Number of issues found per participant

Participant	Issues
user-02	0
user-03	0
user-06	1
user-07	0
user-08	0

DevOps Use Case - Manual

The participants working using the traditional approach performed quite well. Only one participant, *user-09* had problems solving the tasks. He also performed bad in the previous use case when working with the tool. As already mentioned in this section this is mostly caused due to a lack of experience with architectural design problems. One user (*user-01*) needed some additional help to come to the solution of replacing the *MQTT Client* with *AWS Lambda*. In Table 6.11 the number of issues found per participant are summarized.

Table 6.11: Number of issues found per participant

Participant	Issues
user-01	0
user-04	0
user-05	1
user-09	2
user-10	0

Generally, it can be said that the solutions were not that innovative compared to when working with tool support. Many users reached the reference solution with some small variations. Two participants (*user-04* and *user-05*) want to replace the *Web Dashboard* with *Elastic Beanstalk*. When sticking to the semantics defined in this work which are for example explained in Section 4.1.1 the correct action would be MOVE, not REPLACE, but this is not counted as an error. *User-05* also wants to delete the *DB-REST* component, arguing that this component seems to be unnecessary and the *Web Dashboard* could make the SQL call directly. This seems not to be a good idea since so the UI-layer would directly access the persistence layer. Furthermore, some more development work must be done incorporating the *DB-REST* component's functionality into the *Web Dashboard*.

6.3.4 Discussion

To answer the question, if the participants working with the tool performed better than the participants working with the manual approach (*GQ1*), the fact that participants working with the manual approach easily overlooked important constraints must be considered. For example, as it was discussed in Section 6.3.3, multiple participants working with the manual approach wanted to replace the *Oracle database* with *PostgresSQL* although it does not support geospatial data. Furthermore, when comparing the number of issues found per participants as shown in the previous section it can be seen that more issues were found when the traditional approach was used compared to the tool assisted workflow. So the solutions found with tool were better than the solutions found with the traditional approach, especially when not considering the invalid solutions mainly caused by bugs in the tool or by lack of experience of a participant. This means *GQ1* can be answered positively.

When incorporating the already positive answered *GQ1* as well as analysing the results of *FQ2* as it was done in Section 6.3.1 the question if a tool assisted workflow is an improvement compared to a traditional, manual approach (*GQ2*) can also be answered positively.

When comparing the *Overall Experience* with the *Overall Feedback* a weak positive correlation between both can be seen when looking at the scatter plot in Figure 6.5. Similar correlations can also be found when comparing the correlation of the overall experience with the results of single feedback questions such as *FQ2* which should find out if the workflow is an improvement compared to a participant's current approach. Since the correlation is only weak the *GQ3* asking if participants with more experience perform better cannot be answered. Furthermore, too few participants took part in the experiment and therefore the scatter plot is not statistically significant. It can only give some idea, but cannot answer the question if participants with more experience perform better. To answer this question further evaluation must be done, for example by using students in an academic setting to get a higher number of participants.

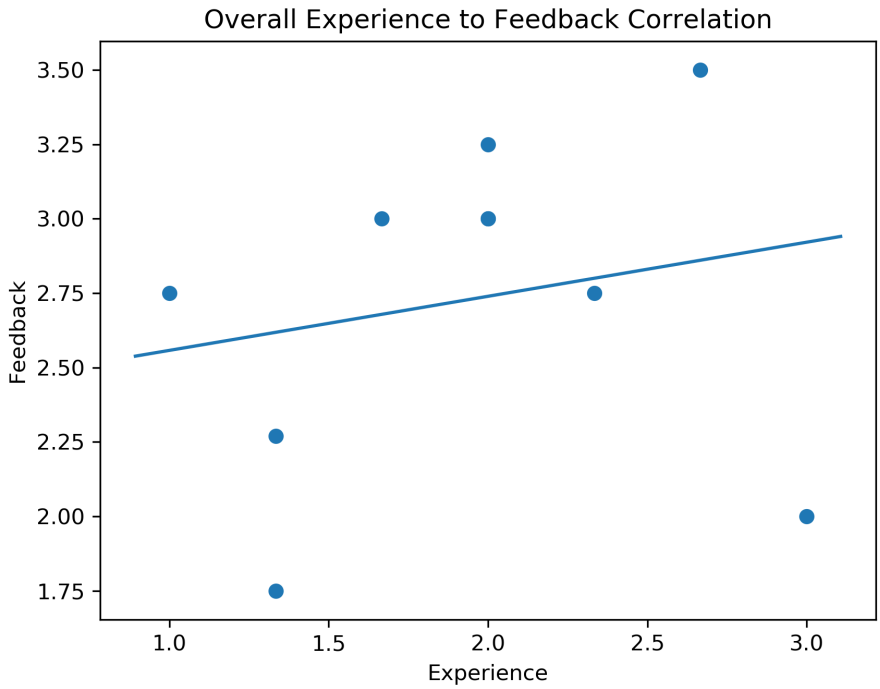


Figure 6.5: Scatter plot: Overall Experience - Overall Feedback.

Finally, especially the second use case (DevOps) showed that using the tool more innovative solutions compared to the traditional approach could be found. The matching and ranking functionality working on data stored in the knowledgebase clearly helps to identify solutions which would have been overlooked when not using such a tool assisted workflow. So *GQ4* defined in Section 6.1.3 can also be answered positively.

To even get more accurate results when ranking the cloud offerings based on their QAs such as performance, or availability live data measured by monitoring the offering could be imported into the knowledgebase. For this experiment all the data to describe the cloud offerings' QAs was simply gathered by reading their documentation and extracting the required information since the data handling is out of scope of this work.

CHAPTER 7

Conclusion and Future Work

7.1 Conclusion and Research Objectives

This work identified and taxonomical described architecture decisions relevant in cloud-edge environments from the point of view of a software architect. Using literature studies it was found, that such architectural decisions are currently often performed in an ad-hoc way [CJT+16, DMAO15]. Since software architecture decision making is a knowledge intensive work, software architects must rely on their knowledge when making decisions. If architects do not have knowledge about a certain technology, research work must be done. Knowledge sharing would be one way to improve decisions.

Especially, the *cloud migration* problem often requires an evolution of a legacy application's architecture. Each cloud platform provides many offerings which could replace components of an existing application as well as there are multiple hosting options for an existing component which can be chosen when moving it into the cloud. Therefore a software architect must also have a good understanding of these cloud offerings.

Based on the findings identified using the literature research work the problem context of architecture decision support in cloud-edge environments was elaborated. For that purpose the main requirements and all the concepts needed were defined. The most important concept discussed in this work are the application types since they can be used to compare components and cloud offerings based on their similarity.

In the following the research questions stated in Section 1.2 are answered and the important findings are discussed.

- *RQ1*: Section 5.1 presents a structured workflow to support a software architect in migrating a legacy application into cloud-edge environments.

  This workflow improves decisions and reduces errors which is shown in Chapter 6 by an experimental evaluation with professional software architects in industry. For

example, when working with tool support less errors, such as neglecting important constraints, could be found compared to working with the traditional approach as discussed in Section 6.3.3.

The participants also gave positive feedback about the tool assisted workflow developed in this work. Most of them think that such a workflow would be usable in their work as well as that the workflow is intuitive. The majority of them also believes that the workflow is an improvement compared to their traditional approach. This is confirmed by the results of the experiment regarding the quality of the solutions since less errors were made with tool support compared to the manual approach.

- *RQ2*: Section 4.1.4 introduces the meta taxonomy to describe cloud offerings which presents how cloud offerings stored in a knowledgebase can be encoded. The most important entities of this meta taxonomy are a cloud offering's capabilities and QA constraints.

  The capabilities enable matching of cloud offerings with a component which should be replaced based on the capabilities they support. After this step only possible replacements for a given component are left and unsuitable offerings are removed. The second step is the ranking of the remaining offerings based on the QA constraints they support as well as their application type.

  The evaluation described in Chapter 6 showed that the tool helps to find solutions for the replacement of a component with a cloud offering which could normally not be found when working with a traditional approach (see discussion in Section 6.3.4). Especially, the matching and ranking capabilities and the shared knowledgebase of the tool help to find solutions which are not found that easily without tool support.

- *RQ3*: Section 5.3 presents an AHP algorithm for cloud offering ranking. This algorithm mainly operates on the QAs a component supports. In this way a trade-off between the QAs required by some component as well as the QAs provided by all offerings which could possibly host this component is achieved by ranking them based on their suitability regarding the component's QA requirements.

  All the QAs required are described using the taxonomy of QAs introduced in Section 4.1.2. For that purpose a well known standard, ISO 25010 [ISO], is used to help an architect selecting the QAs required by his components. For example, the ISO 25010 standard already defines a grouping of QAs into super categories such as security or performance. The taxonomy of QA's enhanced the ISO 25010 standard with QAs also relevant in cloud-edge environments such as availability or cost effectiveness.

  Since the proposed algorithm is an online algorithm also its runtime performance must be evaluated. Therefore, a runtime experiment was conducted (described in Section 5.3.2) using conditions which are expectable in practise. This experiment showed that the algorithm is reasonable fast and does not slow down a decision maker due to bad runtime performance.

- *RQ4*: This work presents a taxonomy of migration patterns in Section 4.1.3 to support a software architect in selecting cloud migration patterns. The patterns used in the taxonomy were taken from the work of Jamshidi et al. [JPCL15] since they are vendor agnostic.

  Their taxonomical classification into migration actions and deployment scenarios enables filtering based on the migration action and deployment scenarios chosen for a given component. This narrows down the migration patterns, which must be considered by a software architect, and thus simplifies the decision which one to choose. A use case for such a pattern selecting, as also demonstrated in the IMS evaluation use case (see Section 6.2.1), would be fixing some communication error caused by an on-premises server no longer reachable by a cloud client due to firewall restrictions.

7.2 Lessons Learned and Future Work

In the following the lessons learned regarding applicability in practise of the proposed concepts and solution of this Master's thesis are discussed also giving directions for future work.

One issue for applicability in practise would be the entry barrier for a software architect. An architect must describe his application's architecture using provided concepts, which means that he has to learn and understand these concepts. For that purpose sufficient documentation should be provided. Using an existing set of concepts is required to enable matching of existing concepts and offerings and to effectively support a user in a structured process.

Regarding the matching and ranking capabilities of the tool, it is not always that easy for a user to reproduce a solution, e.g. to understand how it was derived. Similar as for the previous issue, documentation will help to make the calculations which are performed in the background transparent. Here it is important to give a high level overview of how the matching and ranking basically works to not discourage new users due to its complexity. Additional resources should also be provided for interested users, e.g. how the ranking algorithm works in more detail.

The tool provides access to a reusable knowledgebase containing software engineering domain knowledge including technological entities such as libraries and frameworks. This knowledge evolves quickly meaning that a technology which is currently state-of-the-art could be replaced soon in the future by some new technology. Therefore updating the knowledgebase is an open issue which was not in scope of this work and should be addressed in future works.

Another open issue is providing a solution so that users can benefit from other users who have gone through similar workflows. For that purpose decisions of other users must be stored in the knowledgebase and rated if they were good or bad decisions. Based on such historical information even better decision support can be given.

The ranking algorithm proposed in this work builds on the precondition that the QAs considered are already quantified using a meaningful weight. In this work this was done manually. This was a very time consuming activity since the documentation of all the used cloud offerings had to be read to get a basic understanding of their QAs. The obtained results are not that accurate as for example when QAs such as performance are actively measured. By measuring such QAs it need not be relied on the information given by the cloud vendor. In the literature there are already solutions for quantifying at least some of the QAs presented in this work such as the work of Garg et al. [GVB11]. Now a future work would be to enhance the data stored in the knowledgebase using similar approaches found in the literature.

To further improve the tool and prepare it for practical use the architecture graph view should always be visible as it was suggested by one participant in the experiment (see Section 6.3.1). Furthermore, changes should be visible dynamically in this graph view in an WYSIWG style. Here it is important that the tool aligns with other software architecture tools used in practise and uses well known concepts regarding UI handling in such tools.

Furthermore, the proposed AHP algorithm should also be further improved regarding its runtime performance in future works. For example, to improve its performance well known techniques could be applied such as caching or pre computing results as it is also done by Google's PageRank algorithm to speed up ranking users' search results [BP98, PBMW99].

Finally, more experiments should be conducted with an enhanced version of the tool in the future. This could be for example done in an academic setting with computer science students, to find out in which way the proposed workflow could be further improved.

CHAPTER 8

Appendix

8.1 Table of Abbreviations

In Table 8.1 and Table 8.2 a list of all abbreviations used in this work is given.

Table 8.1: Abbreviations used in this work (1)

Abbreviation	Description
ACID	Atomicity, Consistency, Isolation, Durability: Properties of database transactions to guarantee validity even in the event of errors
AHP	Analytic Hierarchy Process
API	Application Programming Interface
AWS	Amazon Web Services
BI	Business Intelligence
BPMN	Business Process Model and Notation
CaaS	Container as a Service
CBR	Case Based Reasoning
CPU	Central Processing Unit
CRUD	Create, Read, Update, Delete
CSMIC	Cloud Service Measurement Index Consortium
CUDA	Compute Unified Device Architecture (by NVIDIA)
D3	Data Driven Documents
DevOps	Clipped compound of Development and Operations
EBS	AWS Elastic Block Storage
ECS	AWS Elastic Container Service
EFS	AWS Elastic File Storage

Table 8.2: Abbreviations used in this work (2)

FaaS	Function as a Service
GPL	GNU General Public License
GPU	Graphics Processing Unit
IaaS	Infrastructure as a Service
IMS	Infrastructure Monitoring System
IoT	Internet of Things
ISO	International Organization for Standardisation
IT	Information Technology
JWT	JSON Web Token
KB	Knowledgebase
MQTT	Message Queuing Telemetry Transport
NAS	Network Attached Storage
NFR	Non Functional Requirement
NFS	Network File System
NIST	National Institute of Standards and Technology
NoSQL	Non SQL, non relational
PaaS	Platform as a Service
QA	Quality Attribute
Q/A	Questions and Answers
RAM	Random Access Memory
REST	Representational State Transfer
RDS	AWS Relational Database Service
S3	AWS Simple Storage Service
SaaS	Software as a Service
SME	Small and Medium Enterprises
SMI	Service Measurement Index
SQL	Structured Query Language
SOA	Service Oriented Architecture
UI	User Interface
UML	Unified Modelling Language
VM	Virtual Machine
WYSIWYG	What You See Is What You Get
XaaS	Anything as a Service

List of Figures

List of Tables

List of Algorithms

Bibliography

[ABLS13] Vasilios Andrikopoulos, Tobias Binz, Frank Leymann, and Steve Strauch. How to adapt applications for the cloud environment. *Computing*, 95:493–535, 2013.

[AFC+17] J. Axelsson, U. Franke, J. Carlson, S. Sentilles, and A. Cicchetti. Towards the Architecture of a Decision Support Ecosystem for System Component Selection. In *2017 Annual IEEE International Systems Conference (SysCon)*, pages 1–7, 2017.

[AFG+09] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R.H. Katz, A. Konwinski, G. Lee, D.A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Available online at `http://home.cse.ust.hk/~weiwa/teaching/Fall15-COMP6611B/reading_list/AboveTheClouds.pdf`, last visited at 2018-01-11, 2009.

[Amaa] Amazon. Amazon Elastic Beanstalk Documentation. Available online at `https://aws.amazon.com/elasticbeanstalk`, last visited at 2018-01-19.

[Amab] Amazon. Amazon Elastic Beanstalk FAQ. Available online at `https://aws.amazon.com/elasticbeanstalk/faqs`, last visited at 2018-01-19.

[Amac] Amazon. Amazon Elastic Beanstalk Supported Platforms Documentation. Available online at `http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/concepts.platforms.html`, last visited at 2018-01-19.

[Amad] Amazon. Amazon Elastic Compute Cloud Documentation. Available online at `https://aws.amazon.com/ec2/`, last visited at 2018-01-19.

[Amae] Amazon. Amazon Elastic Container Service Documentation. Available online at `https://aws.amazon.com/ecs/`, last visited at 2018-01-19.

[Amaf] Amazon. Amazon Glacier Documentation. Available online at https://aws.amazon.com/glacier/, last visited at 2018-01-19.

[Amag] Amazon. Amazon Glue Documentation. Available online at https://aws.amazon.com/de/glue/, last visited at 2018-01-19.

[Amah] Amazon. Amazon Identity and Access Management Documentation. Available online at https://aws.amazon.com/iam/, last visited at 2018-01-19.

[Amai] Amazon. Amazon Lambda Documentation. Available online at https://aws.amazon.com/lambda/, last visited at 2018-01-19.

[Amaj] Amazon. Amazon RDS Documentation. Available online at https://aws.amazon.com/rds, last visited at 2018-01-19.

[Amak] Amazon. Amazon Simple Queuing Service Documentation. Available online at https://aws.amazon.com/sqs, last visited at 2018-01-19.

[Amal] Amazon. AWS Cloud Adoption White-Paper. Available online at https://d1.awsstatic.com/whitepapers/aws_cloud_adoption_framework.pdf, last visited at 2018-01-19.

[Ama15] Amazon. A Practical Guide to Cloud Migration. Available online at https://d0.awsstatic.com/whitepapers/the-path-to-the-cloud-dec2015.pdf, last visited on 2018-01-16, 2015.

[ASE15] Abdullah Alhammadi, Clare Stanier, and Alan Eardely. A Knowledge Based Decision Making Tool to Support Cloud Migration Decision Making. In *ICEIS 15 (International Conference on Enterprise Information Systems)*, pages 637–643, 2015.

[ASL13a] V. Andrikopoulos, Z. Song, and F. Leymann. Supporting the Migration of Applications to the Cloud through a Decision Support System. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 565–572, 2013.

[ASL13b] Vasilios Andrikopoulos, Steve Strauch, and Frank Leymann. Decision Support for Application Migration to the Cloud: Challenges and Vision. pages 149–155, 2013.

[BB08] Janet E. Burge and David C. Brown. Software Engineering Using RATionale. *Journal of Systems and Software*, 81(3):395 – 413, 2008. Selected Papers from the 2006 Brazilian Symposia on Databases and on Software Engineering.

[BGPCV12] Mark L. Badger, Timothy Grance, Robert Patt-Corner, and Jeffrey M. Voas. Cloud Computing Synopsis and Recommendations. Available online at `https://www.nist.gov/publications/cloud-computing-synopsis-and-recommendations`, last visited at 2018-01-11, 2012.

[BP98] Sergey Brin and Lawrence Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. *Computer Networks and ISDN Systems*, 30(1-7):107–117, 1998.

[CJT+16] Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 10 years of software architecture knowledge management: Practice and future. *Journal of Systems and Software*, 116(Supplement C):191 – 205, 2016.

[CNC08] R. Capilla, F. Nava, and C. Carrillo. Effort Estimation in Capturing Architectural Knowledge. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 208–217, 2008.

[CNPDn06] Rafael Capilla, Francisco Nava, Sandra Pérez, and Juan C. Dueñas. A Web-based Tool for Managing Architectural Design Decisions. *SIGSOFT Softw. Eng. Notes*, 31(5), September 2006.

[CPP16] J. Carlson, E. Papatheocharous, and K. Petersen. A Context Model for Architectural Decision Support. In *2016 1st International Workshop on Decision Making in Software ARCHitecture (MARCH)*, pages 9–15, 2016.

[CRLS09] Thomas H. Cormen, Ronald L. Rivest, Charles E. Leiserson, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 2009.

[CW87] D. Coppersmith and S. Winograd. Matrix Multiplication via Arithmetic Progressions. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 1–6, New York, NY, USA, 1987. ACM.

[Dar14] Alexander Maximilan Darsow. Decision Support for Application Migration to the Cloud. Available online at `ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/MSTR-3572/MSTR-3572.pdf`, last visited at 2017-12-11, 2014.

[DMAO15] S. Dasanayake, J. Markkula, S. Aaramaa, and M. Oivo. Software Architecture Decision-Making Practices and Challenges: An Industrial Case Study. In *24th Australasian Software Engineering Conference (ASWEC)*, pages 88–97, 2015.

[FLvV07] Rik Farenhorst, Patricia Lago, and Hans van Vliet. EAGLE: Effective Tool Support for Sharing Architectural Knowledge. *International Journal of Cooperative Information Systems*, 16(03n04):413–437, 2007.

[Foua] Cloud Foundry. Blue Green Deployment. Available online at `http://iso25000.com/index.php/en/iso-25000-standards/iso-25010`, last visited at 2018-01-19.

[Foub] Cloud Foundry. Cloud Foundry PaaS Solution. Available online at `https://docs.cloudfoundry.org/devguide/deploy-apps/blue-green.html`, last visited at 2018-01-19.

[GC12] R. C. Garcia and J. M. Chung. XaaS for XaaS: An evolving abstraction of web services for the entrepreneur, developer, and consumer. In *2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 853–855, 2012.

[GJGN13] S. Gudenkauf, M. Josefiok, A. Göring, and O. Norkus. A reference architecture for cloud service offers. In *2013 17th IEEE International Enterprise Distributed Object Computing Conference*, pages 227–236, 2013.

[Gro11] Info-Tech Research Group. Cloud vendor shortlisting tool. Available online at `https://www.infotech.com/research/cloud-vendor-shortlisting-tool`, last visited at 2017-12-21, 2011.

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

[IGK+15] B. I. Ismail, E. Mostajeran Goortani, M. B. Ab Karim, W. Ming Tat, S. Setapa, J. Y. Luke, and O. Hong Hoe. Evaluation of Docker as Edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)*, pages 130–135, 2015.

[ISO] ISO 25010 Standard. Available online at `http://iso25000.com/index.php/en/iso-25000-standards/iso-25010`, last visited at 2018-01-19.

[JAP13] P. Jamshidi, A. Ahmad, and C. Pahl. Cloud Migration Research: A Systematic Review. *IEEE Transactions on Cloud Computing*, 1(2):142–157, 2013.

[JdVAvV08] Anton Jansen, Tjaard de Vries, Paris Avgeriou, and Martijn van Veelen. Sharing the Architectural Knowledge of Quantitative Analysis. In Steffen Becker, Frantisek Plasil, and Ralf Reussner, editors, *Quality of Software Architectures. Models and Architectures*, pages 220–234, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[JPCL15] Pooyan Jamshidi, Claus Pahl, Samuel Chinenyeze, and Xiaodong Liu. Cloud Migration Patterns: A Multi-cloud Service Architecture Perspective. 2015.

[JVB13] Adrian Juan-Verdejo and Henning Baars. Decision Support for Partially Moving Applications to the Cloud: The Example of Business Intelligence. In *Proceedings of the 2013 International Workshop on Hot Topics in Cloud Services*, HotTopiCS '13, pages 35–42. ACM, 2013.

[JVZS+14] A. Juan-Verdejo, S. Zschaler, B. Surajbali, H. Baars, and H. G. Kemper. InCLOUDer: A Formalised Decision Support Modelling Approach to Migrate Applications to Cloud Environments. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 467–474, 2014.

[Kel17] Frank Kelley. AWS Migration Patterns. Available online at `https://docs.microsoft.com/en-us/azure/architecture/patterns/`, last visited on 2018-01-16, 2017.

[LTZ13] Ioanna Lytra, Huy Tran, and Uwe Zdun. Supporting Consistency Between Architectural Design Decisions and Component Models Through Reusable Architectural Knowledge Transformations. In *Proceedings of the 7th European Conference on Software Architecture*, ECSA'13, pages 224–239, Berlin, Heidelberg, 2013. Springer-Verlag.

[MG11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Available online at `https://csrc.nist.gov/publications/detail/sp/800-145/final`, last visited at 2018-01-08, 2011.

[Mica] Microsoft. Microsoft Azure AWS Comparision. Available online at `https://docs.microsoft.com/en-us/azure/architecture/aws-professional/services`, last visited at 2018-01-19.

[Micb] Microsoft. Migration to Microsoft Azure. Available online at `https://azure.microsoft.com/en-us/migrate/`, last visited at 2018-01-19.

[Mic17] Cloud Design Patterns. Available online at `https://docs.microsoft.com/en-us/azure/architecture/patterns/`, last visited at 2018-01-16, 2017.

[MR12] Michael Menzel and Rajiv Ranjan. Cloudgenius: Decision Support for Web Server Cloud Migration. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 979–988. ACM, 2012.

[MRW+15] M. Menzel, R. Ranjan, L. Wang, S. U. Khan, and J. Chen. CloudGenius: A Hybrid Decision Support Method for Automating the Migration of Web Application Clusters to Public Clouds. *IEEE Transactions on Computers*, 64(5):1336–1348, 2015.

[MTK+14] C. Manteuffel, D. Tofan, H. Koziolek, T. Goldschmidt, and P. Avgeriou. Industrial implementation of a documentation framework for architectural decisions. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 225–234, 2014.

[NP13] Marcin Nowak and Cesare Pautasso. Team Situational Awareness and Architectural Decision Making with the Software Architecture Warehouse. In Khalil Drira, editor, *Software Architecture*, pages 146–161, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Orb16] Stephen Orban. 6 Strategies for Migrating Applications to the Cloud. Available online at `https://medium.com/aws-enterprise-collection/6-strategies-for-migrating-applications-to-the-cloud-eb4e85c412b4` last visited on 2018-01-16, 2016.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[PSZ18] K. Plakidas, D. Schall, and U. Zdun. Context-aware application model facilitating architecture evolution, 2018.

[Saa90] Thomas L. Saaty. How to make a decision: The Analytic Hierarchy Process. *European Journal of Operational Research*, 48(1):9–26, 1990.

[SAP] SAP. SAP HANA Cloud. Available online at `https://www.sap.com/germany/products/hana.html`, last visited at 2018-01-19.

[SCZ+16] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[SD16] W. Shi and S. Dustdar. The Promise of Edge Computing. *Computer*, 49(5):78–81, 2016.

[Sie] Siemens. Mindsphere PaaS. Available online at `https://www.siemens.com/global/de/home/produkte/software/mindsphere.html`, last visited at 2018-01-19.

[SSJL12] Basem Suleiman, Sherif Sakr, Ross Jeffery, and Anna Liu. On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure. *Journal of Internet Services and Applications*, 3(2):173–193, 2012.

[SWLM03] Mikael Svahnberg, Claes Wohlin, Lars Lundberg, and Michael Mattson. A Quality-Driven Decision-Support Method For Identifying Software Architecture Candidates. *International Journal of Software Engineering and Knowledge Engineering*, 13(05):547–573, 2003.

[SZP07] Nelly Schuster, Olaf Zimmermann, and Cesare Pautasso. ADkwik: Web 2.0 Collaboration System for Architectural Decision Engineering. In *Nineteenth International Conference on Software Engineering and Knowledge Engineering (SEKE 2007)*, pages 255–260, Boston, USA, July 2007.

[TA05] J. Tyree and A. Akerman. Architecture decisions: demystifying architecture. *IEEE Software*, 22(2):19–27, 2005.

[TJH07] Antony Tang, Yan Jin, and Jun Han. A rationale-based architecture model for design traceability and reasoning. *J. Syst. Softw.*, 80(6):918–934, 2007.

[TSS+16] K. Telschig, N. Schöffel, K. B. Schultis, C. Elsner, and A. Knapp. SECO Patterns: Architectural Decision Support in Software Ecosystems. In *2016 1st International Workshop on Decision Making in Software ARCHitecture (MARCH)*, pages 38–44, 2016.

[WG14] Rainer Weinreich and Iris Groher. *A Fresh Look at Codification Approaches for SAKM: A Systematic Literature Review*, pages 1–16. Springer International Publishing, 2014.

[WKH16] B. M. R. Wilson, B. Khazaei, and L. Hirsch. Cloud adoption decision support for SMEs using Analytical Hierarchy Process (AHP). In *2016 IEEE 4th Workshop on Advances in Information, Electronic and Electrical Engineering (AIEEE)*, pages 1–4, 2016.

[WRH+12] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[WWSS17] K. Wehling, D. Wille, C. Seidl, and I. Schaefer. Decision Support for Reducing Unnecessary IT Complexity of Application Architectures. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 161–168, 2017.