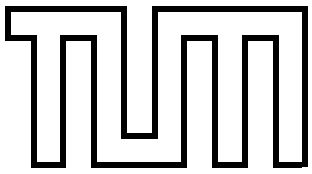


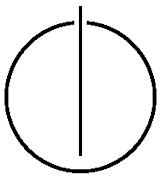
DEPARTMENT OF INFORMATICS

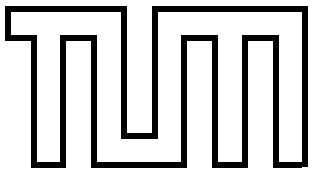
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Improving the Usability of an Integrated Decision Support System for Design Decision Making

Jiping Xu





DEPARTMENT OF INFORMATICS

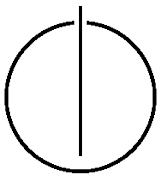
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Improving the Usability of an Integrated Decision Support System for Design Decision Making

Verbesserung der Benutzerfreundlichkeit eines integrierten Entscheidungshilfesystems für die Design-Entscheidungsfindung

Author:	Jiping Xu
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	M.Sc. Manoj Mahabaleshwar
Submission Date:	16.07.2018



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 16.07.2018 Jiping Xu

Acknowledgments

After an intensive period of six months, I would like to thank all the people who supported me during the course of this thesis.

Special thanks go to my advisor, Manoj Mahabaleshwar for his support and valuable guidance. He provided me with inspiring discussions and helped me to stay in the right direction of research.

In addition, I would like to thank Prof. Dr. Florian Matthes for the opportunity I was given to write my thesis at the chair of Software Engineering for Business Information Systems.

I would also like to thank my parents and friends for their continuous support and encouragement throughout the time.

Jiping Xu

Abstract

The importance of managing architectural knowledge has been highlighted in the past decade. Various models and tools have emerged to capture, share and use architectural knowledge. Architecture Management Enabler for Leading Industrial software (AMELIE) is an on-going research project for managing architectural design decisions in large software-intensive projects. This thesis analyzes how to ensure the usability of such a complex decision support system from the following perspectives: (a) end-user perspective – guiding end-users through performing their tasks, (b) developers perspective – providing documentation for extending and maintaining the system, and (c) system administrator – enabling continuous deployment by containerization. The result is three-fold. Firstly, a requirement analysis for architectural knowledge management and architectural decision making is performed. The generated requirements list provides guidelines to check whether a given decision support system fulfills the needs of its stakeholders. Secondly, comparing AMELIE against the requirement list reveals that some requirements do not fully function due to flaws in the user interfaces. Those flaws are identified by an expert review. A before and after comparison shows the usability improvement after removing the flaws. Thirdly, as AMELIE adopts a microservice architecture, all microservices are containerized for better development experience. An end-user manual and a developer documentation are created correspondingly to ease the use of the system.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**
1.1 Motivation 1
1.2 Research Questions 2
1.3 Outline 3

2 Background **4**
2.1 Architectural Design Decision 4
2.2 Decision Making Process 5
2.3 Decision Support System 6
2.4 Related Work 6
2.5 AMELIE 8
2.5.1 Microservice Architecture 9
2.5.2 Services 10

3 Requirements Analysis **14**
3.1 Architectural Knowledge Management Requirements 14
3.1.1 Requirements Elicitation 14
3.1.2 AKM Requirements Coverage 15
3.2 Architectural Design Decision Requirements 18
3.2.1 Requirements Elicitation 18
3.2.2 ADD Requirements Coverage 19
3.3 Practitioner Feedback 21
3.4 Researcher Feedback 22
3.5 Current Status of AMELIE 22
3.6 Final Results of Requirements Elicitation 23
3.6.1 Notation 23
3.6.2 AKM Requirements List 23
3.6.3 ADD Requirements List 27

4 Enhancing User Interfaces **30**
4.1 Expert Review Method . . . 30
4.2 Expert Review Process . . . 31
4.3 Results . . . 31
4.3.1 Projects . . . 31
4.3.2 Import . . . 33
4.3.3 Quality Attributes . . . 36
4.3.4 Architectural Elements . . . 39
4.3.5 Expertise Matrix . . . 40
4.3.6 Expert Recommender . . . 41
4.3.7 Design Decisions . . . 43
4.3.8 Design Decision Details . . . 46

5 Containerization **49**
5.1 Introduction . . . 49
5.1.1 Docker Container . . . 49
5.1.2 Docker Image . . . 50
5.1.3 Docker Compose . . . 51
5.2 Containerize AKRE-Server . . . 51
5.2.1 Create Dockerfile . . . 51
5.2.2 Create docker-compose file . . . 51
5.3 Shared Data Storage . . . 52
5.4 One-click Deployment . . . 53

6 End-user and Developer Documentation **55**
6.1 End-user Documentation . . . 55
6.1.1 Design Principles . . . 55
6.1.2 Implementation . . . 56
6.1.3 Result . . . 56
6.2 Developer Documentation . . . 56
6.2.1 API Documentation . . . 58
6.2.2 Machine Learning Model Configuration . . . 59

7 Conclusion and Future Work **61**
7.1 Conclusion . . . 61
7.2 Future Work . . . 62

List of Figures **64**

List of Tables **65**

Bibliography **66**

1 Introduction

1.1 Motivation

The importance of managing architectural knowledge has been highlighted in the past decade [7]. The research focus has shifted from architecture as an artifact to the process of how it is created. This leads to the perspective that software architecture is the result of a set of architectural design decisions [23]. Software architects face difficult design decisions every day. Documenting past design decisions addresses recurring problems in a similar context and avoids knowledge vaporization. It also supports stakeholders to reason about the architecture via tracing decisions to architecture and implementation [8]. However, despite the benefits of documentation, architects are concerned about the manual effort and cost of doing so [10]. The architectural knowledge evaporates and could not be reused. This calls for a system that can manage design decisions made in past and aid architects while making new decisions. Architecture Management Enabler for Leading Industrial software (AMELIE) is proposed to address the aforementioned problems.

AMELIE a meta-model based framework for managing architectural design decisions in large software-intensive projects [6]. It is a complex system built using the microservice architecture. Its microservices include SyncPipes, AKRE-Server, Document Clustering, Document Classifier, Amelietor and data storage. These microservices are developed by different people at different times and some of them are part of students theses. The diversity in the system leads to multiple usability problems. The word 'usability' here does not refer only to the usability of user interfaces (UI) but from a broad view of how usable the system is. Since there are two types of stakeholders for AMELIE: (1) end-users who want to get assistance from the system when making design decisions and (2) developers who want to develop and adjust the system according to their needs, the usability problems are identified and solved from these two different perspectives. Solving these problems improves the usability of AMELIE, which contributes to better management of architectural design decisions.

1.2 Research Questions

This thesis aims to improve the usability of AMELIE from multiple aspects by answering the following research questions:

RQ1: What are the requirements of a decision support system (DSS) for making design decisions?

Architectural knowledge management (AKM) has drawn much attention in the past. Different tools have been developed to support recording and managing architectural knowledge. But the shift from architecture itself to architectural design decisions (ADD) happened in the recent years. Compared to AKM, ADD is a rather new research topic with emerging requirements and tools. As there is little systematic research about the requirements of such a decision support system, it is important to understand what are the use cases of such a system in the first place. To ensure that the requirements are not only meaningful to scientific research but also useful for practitioners, employees from the industry partner of this project are interviewed for their feedback. The outcome is a list of AKM and ADD requirements. The resulting requirements are further labeled with their relevance to research areas, practical work, and AMELIE project.

RQ2: How to improve the usability of an existing DSS that supports design decision making?

The usability of a system is dependent on the role of its stakeholders. The different types of stakeholders are firstly identified: end-users and developers. Then multiple aspects of usability are examined for each type of stakeholders.

For end-users who want to use AMELIE to make design decisions, the usability is concerned with the usability of UIs, the understandability, and learnability of the system. The former one is addressed by performing an expert review on the UIs and resolving the identified issues in these UIs. The latter one is enhanced by providing a step-by-step end-user manual.

Developers are interested in extending the existing system or utilizing it according to their own needs. Understandability is crucial in both scenarios. Without proper documentation, developers have difficulties to extend and configure the system. Furthermore, as the system evolves over time, maintainability can become a severe issue that hinders usability. These problems are addressed by containerization of the system and a detailed developer documentation.

1.3 Outline

The next chapter discusses the foundation of this thesis. The terminology used in the thesis is explained and the related work is introduced. It also gives an overview of the AMELIE project.

In Chapter 3, a requirement analysis is performed based on a literature review. Then, stakeholders with different interests are interviewed for feedback. They also go through the implemented system to reflect on the current status of AMELIE according to the requirements.

Chapter 4 starts with an expert review of the UI of the system. Then the identified usability problems are solved by redesigning and reimplementing the problematic UI elements. The before and after comparison shows the changes and improvements.

Afterward, Chapter 5, addresses the usability from the perspective of development and deployment. Docker is introduced into the system and each microservice is containerized. The system is also containerized into one container with multiple services.

In Chapter 6, the creation of different types of documentation is discussed to address the needs of different stakeholders.

The last chapter summarizes the thesis and gives an outlook on the future work.

2 Background

This chapter serves as the foundation for understanding this thesis. The concepts of architectural design decisions, decision-making process, and decision support system are elaborated. In the second section, related work is discussed. Last but not least, as this thesis is based on the AMELIE project, a brief introduction to AMELIE is essential to the work in the following chapters.

2.1 Architectural Design Decision

Software architecture is the result of a set of architectural design decisions [23]. Design decisions are made by architects and developers in different phases of software lifecycle from design, implementation to maintenance. They play an important role not only in the overall quality of the architecture, but also in satisfying and balancing stakeholders' concerns [47].

Design Decision Description

A design decision can be expressed using the following attributes [23]: **rationale** is the reason behind the decision; **design rules** are the guidelines for making design decisions, while **design constraints** describe what is not allowed in the future of the design; **additional requirements** are cascading requirements that need to be satisfied to realize such a design decision.

Categories of Design Decisions

Considering the increasing amount of design decisions in a system over time, an ontology is helpful to categorize and manage different design decisions. In the work by Kruchten [27], design decisions are classified into three categories: property, existence and executive decisions.

- **Property decisions** are the ones concerned with design guidelines and constraints. They form the basis for making other design decisions.

- **Executive decisions** do not relate directly to the design elements. They are driven by the business environment and affect the development process and the organization.
- **Existence decisions** states the existence of architectural elements. They are further categorized into structural, behavioral and ban decisions. **Structural decisions** result in new elements in the system, such as a new component. **Behavioral decisions** deal with the interaction between components and fulfilling quality attributes. **Ban or non-existence decisions** lead to the removal of an element.

2.2 Decision Making Process

There are many benefits of capturing design decisions: lower costs of change [7] and reusability [25]. However, it is difficult to make good design decisions because the reasoning process of an architect can fail and the premises used for reasoning can be untrue [44]. Solving these problems requires understanding people's behaviors during the process of design decision making.

Tang et al. studied the psychological and cognitive aspects related to human thinking [44], including naturalistic and rational decision making, cognitive biases, group decision making, cognitive limitation and mental representation.

- **Naturalistic decision making** shows that people can make a decision based on the possible outcome. They do not need other options to compare with. **Rational decision making** describes how people use logic to make a choice between alternatives in an objective way.
- **Cognitive biases** is people's inability to make rational decisions. As architects make design decisions based on their skills and past experience, they can make sub-optimal decisions [46]. For example, they choose to use software solutions they are familiar with, but not the ones that best suit the situation.
- **Group decision making** involves conflicts during decision making. Group members have the tendency to minimize conflicts even if the result is not rational. However, group-thinking can also generate more ideas than individuals. Group members can compensate for each other's limitation in experience and expertise.
- **Cognitive limitation** in the context of design decision making refers to the limit of information, time and people's cognitive abilities [39]. Because of these limitations, people are not able to perform a thorough reasoning before they reach the conclusion.

- **Mental representation** is how an architect structures the decision mentally and transforms it into representation. People with different experience represent the same decision in different ways, which may lead to confusion and misunderstanding.

2.3 Decision Support System

Decision Support System (DSS) is an information system for supporting decision-making process. It is described as follows by Sprague [41]:

- The problem to be solved by a DSS is usually not so well structured and under-specified. It can be a high-level problem that managers typically face.
- The use of models or analytic techniques is combined with traditional data access and retrieval functions. The analysis provides insight into relevant information that guides or constraints the solution to the problem.
- Since a DSS provides support during decision making, it must be easy to use and provide interactive guidance.
- A DSS evolves as the environment and the problem to be solved change over time. It requires flexibility and adaptability that allows fast modification and extension [26].

2.4 Related Work

In this section, various tools that support design decision making are introduced. They represent the current research of architectural knowledge management as well as architectural design decisions.

Archium

The Archium tool aims at establishing traceability among various concepts, including design decisions, features, concerns, architecture descriptions, and implementations [23]. The relationship among these concepts is maintained during the complete lifecycle of the system, which means the data is always up-to-date. This helps an architect to better understand the current status of the architecture. All of these concepts can be expressed by the Archium language, which is an extension of the Java programming language.

The dependencies among design decisions can be visualized in Archium. An architect can navigate between different views to analyze potential outcomes of changing design decisions.

Architecture Design Decision Support System

Architecture Design Decision Support System (ADDSS) is a web-based tool for managing architectural design decisions [11]. It can capture, store, manage, and document architectural design decisions during different phases of the software lifecycle. Design decisions are associated with the development iteration, which enables an architect to view the evolution of architectural knowledge over time.

ADDSS provides links between design decisions and architectures for forward and backward traceability. The relationship among design decisions is managed through a dependency model.

Architecture Rationale and Element Linkage

Architecture Rationale and Element Linkage (AREL) is a UML-based tool to assist architects with creating and documenting architectural design with a focus on architectural design decisions and design rationale [43]. Design decisions are captured together with design concerns and design outcomes. Design concerns are the guidelines and constraints that influence the adoption of certain design decisions. Design outcomes are the result of applying design decisions. These three concepts are linked by the AREL causal model. In this way, it is possible to trace bi-directionally and analyze existing conflicts.

Knowledge Architect

The Knowledge Architect (KA) is a tool suite for capturing, sharing and managing architectural knowledge [32]. The tool suite consists of Knowledge Repository and five tools. Knowledge Repository stores the knowledge entities. The five tools are Document Knowledge Client, Knowledge Explorer, Knowledge Translator, Excel plug-in and Python plug-in. These tools enable the architect to extract architectural design decisions in different environments. The relationship between decisions can be visualized in the Knowledge Explorer. The Knowledge Translator is responsible for translating architectural knowledge (AK) from one domain model to another as the Knowledge Architect aims to provide a generic solution to AKM.

Process-based Architecture Knowledge Management Environment

Process-based Architecture Knowledge Management Environment (PAKME) is a web-based tool that aims to provide AKM support during the software architecting process [4]. It has four main services: knowledge acquisition, knowledge maintenance, knowledge retrieval and knowledge presentation. Knowledge acquisition is supported by the web client application. Architects can create new AK by entering them into web forms. A knowledge management component enables them to modify AK later. Existing knowledge can be retrieved through a search component. A representation component generates reports for analysis.

Architectural Decision Knowledge Wiki

Architectural Decision Knowledge Wiki (ADkwik) is a model-based collaborative system for architectural decision modeling and decision process support [37]. It leverages a rich user interface specially designed for collaboration in the form of comments, tags, and attachments. Users can access the system simply through a web browser.

The above-mentioned tools allow architects to capture design decisions (DDs) and its related concepts such as rationale, design rules, and design constraints. But they all follow a top-down approach, which means that architects have to first create knowledge manually in such tools to get decision support. Since they have their own working environment like MS Project, JIRA, EA, etc., for documenting architecture-related information, they are concerned with the extra effort and cost of documenting the same information for the second time in a different system [10]. This duplication of work is a burden and also leads to inconsistency among different information management systems.

That is why the AMELIE project aims to address the challenges of Archium, ADDSS, AREL, KA, PAKME and ADkwik in a bottom-up fashion, which means to import project data from external platforms into AMELIE automatically. Thus, architects no longer need to capture their design decisions in multiple systems separately. These aspects are further elaborated in the next section.

2.5 AMELIE

AMELIE is a meta-model based framework to support architects with managing architectural knowledge and making design decisions. It captures generic use cases for AKM tools such as capturing, storing and sharing AK; as well as organization-specific use cases within an industrial scenario such as enabling stakeholders to manage their

artifacts in different tools [5]. In this section, the architecture of this framework is introduced along with its individual components.

2.5.1 Microservice Architecture

AMELIE is built using the microservice architecture. Compared to the traditional monolithic architecture, microservices split the whole application into a set of single-purpose components. Each microservice is a small application with its own business logic. The communication between microservices is enabled through well-defined interfaces. Microservice ensures high cohesion within the service and loose coupling among the services.

Benefits of microservice architecture

- Better understandability of code: a microservice contains a clearly defined functionality, which is easier to understand and maintain.
- Independent development: several microservices can be developed independently by different teams. It is not necessary for them to know all the details of other services.
- Easier deployment: microservices can also be deployed independently. This enables continuous deployment and DevOps.
- Lower risk of change: a change in one microservice does not affect other services. New technologies can be easily adopted without risking the functionality of the whole system.
- Higher scalability: microservices can be scaled independently. The ones with higher demand can be allocated with more resources without affecting the others.

Drawbacks of microservice architecture

- Extra complexity as a distributed system: since all services are deployed independently, the whole application is now a distributed system. All challenges and risks of a distributed system need to be taken care of.
- Complicated testing: testing one service requires running all the services that it depends on. Otherwise, stubs need to be created to simulate the behavior of other services.
- Changes span over microservices: some changes may affect multiple services. In this case, the coordination between different services is a crucial task.

2.5.2 Services

Figure 2.1 depicts the architecture of AMELIE. It is shown that the whole system is decomposed into multiple smaller services, and the services communicate with each other through API calls. Microservice architecture makes it possible for the services to adopt different technologies such as different programming languages and frameworks. In this way, the services are not directly dependent on each other which reduces the complexity of the whole system.

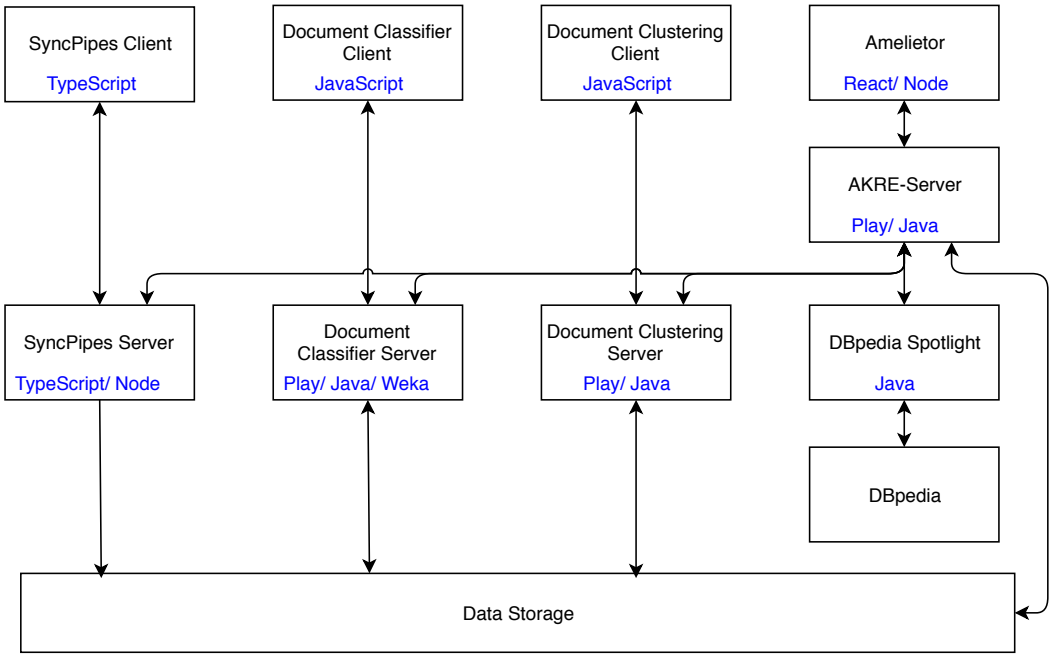


Figure 2.1: Microservice Architecture of AMELIE

The following is a brief introduction to each microservice.

SyncPipes

SyncPipes is a web application to import project data from external platforms into AMELIE. It has a server for data processing and a client for configuration. The server-side employs a bottom-up approach to extract issues automatically from issue management systems such as JIRA, GitHub and Enterprise Architect. Compared to other top-down approaches introduced in Section 2.4, it requires no extra effort from the end-users. They continue to manage the architecture and related issues in the tools they use before. SyncPipes takes care of mapping and transforming the original data to

the conceptual model of AMELIE. In top-down approaches, users need to input the issues and design decisions into the AKM tool themselves. This duplication of work is a burden and also leads to inconsistency among different information management systems.

Document Classifier

Document Classifier is a web application for identifying and classifying design decisions. Its working process is as follows. Firstly, a machine learning model is created using its web client. Users can define the data source, attributes, training method, and parameters. Then the trained model is saved into the database. After training, it is possible to see the graphical results in the web interface. This provides insights into the training parameters as well as the data source being used. Once the model is created, it can be used to detect and classify design decisions when importing a project. Classifying a design decision firstly checks whether an issue is a design decision. If it is, it is further categorized according to the ontology presented by Kruchten [27].

Document Clustering

Document Clustering is used to cluster similar design decisions. It is a web application built with a server and a client. Similar to Document Classifier, it adopts machine learning technologies. The clustering model is created through the web client. The user can choose different clustering algorithms and feature extractors. After training, the results are visualized for reviewing and adjustments. After the model is tuned, it is used during the importing phase of a project. Similar design decisions are clustered to enable reuse of past experience. When the architect makes a new design decision, he can look up how similar decisions that were made in the past.

Amelietor

Amelietor is the front-end application for end-users. It provides overview about existing design decisions addressing different quality attributes or architectural elements. Users can also get recommendations from experts specialized in a certain field. Detailed information about certain design decisions is available as well. Users are recommended with software solutions to an architectural element or alternative solutions to replace the current one.

AKRE-Server

AKRE-Server serves as the middleware between the front-end client, Amelietor, and internal services. It is developed with Play framework in Java. AKRE-Server handles all the requests from the client and dispatches them to corresponding services. This design enables a light-weight and fast client. When a request is finished, AKRE-Server notifies the client with the response sent back. The response is reflected in the client for further operations.

DBpedia Spotlight

DBpedia Spotlight[1] is an open source project for automatic annotation of DBpedia entities in natural language text [13]. It works in four steps: spotting, candidate selection, disambiguation, and filtering. Spotting identifies substrings that may be entities. Candidate selection finds DBpedia resources that are candidate meaning for those substrings. For each substring, the most likely candidate is decided during disambiguation. Finally, the results are filtered according to user-provided configurations.

DBpedia Spotlight service is called through RESTful APIs. The text to be annotated is sent as the request body and an array of annotated entities is returned. In the case of AMELIE, the summary and description of the design decision are sent to be annotated. The returned entities are highlighted in the text while the response data is further processed. The user clicks on an entity to see detailed information such as meta-data, software solutions, and alternatives.

Data Storage

Two types of data storage are utilized in AMELIE:

- **MongoDB** is a document-oriented NoSQL database which provides high flexibility of database schema. It stores configuration for other services and imported project data. The configuration includes pipeline settings for SyncPipes, machine learning models for Document Classifier and Document Clustering. They are created and modified in the individual client of each service. In Amelietor, the configuration data is used directly during the project importing phase. After a project is imported, various data is generated for analysis purpose, including its issues, design decisions, quality attributes, architectural elements, expertise and expert recommendation. This data is saved into MongoDB for further user operations.

[1] https://www.dbpedia-spotlight.org/

- **SocioCortex**[2] is an information management system for collaborative knowledge work. It serves as the issue management system for some projects such as AMELIE. The original information is stored on SocioCortex and can be retrieved for processing. For example, it can be used to train machine learning models. It is also possible to import projects from SocioCortex.

[2] http://www.sociocortex.com/

3 Requirements Analysis

Understanding the requirements of a system is crucial to the overall usability. In this chapter, a requirement analysis is performed based on a literature review. Since AMELIE is a decision support system, requirements focused on architectural design decisions are separated from general architectural knowledge management requirements. Then multiple practitioners and researchers are interviewed using the requirements lists. Their feedback provides insight into the different needs of industry and scientific research.

3.1 Architectural Knowledge Management Requirements

3.1.1 Requirements Elicitation

During the past decade, architectural knowledge management (AKM) has gained much attention. Several AKM models have been proposed to solve the problem of knowledge evaporation. Most of the models also have corresponding tools to evaluate the feasibility and usability of the model. Comparing these tools reveals both the basic requirements that are covered by most of them as well as special requirements according to the needs of the researchers.

Literature Selection

As the models develop different terms for similar concepts, the tools differ a lot from each other. As a result, it is difficult to compare the tools. Under this situation, Tang et al. [42] developed an evaluation framework defined by a set of ten criteria. These criteria are generated based on various AK activities during different stages of the software-lifecycle. During this research, five AKM tools were examined, namely ADDSS [11], Archium [23], AREL [43], The Knowledge architect [24] and PAKME [2]. The tools were compared against the framework to see whether they support the AK activities or not. These activities are considered to be functional requirements for AKM tools. The coverage of an activity by tools indicates the importance of a requirement.

Another systematic research was done by Capilla et al. [9] several years later than Tang. It is performed in a chronological way to find out the changing trend of AKM

tools. Seventeen AKM tools are categorized into three generations according to the development year. Then each generation is assessed separately. Based on the previous research by Feilkas et al. [17], list of eleven generic AK features is developed to compare the functionality coverage of the AKM tools. These features are also considered to be expected requirements of such tools.

Besides eliciting requirements from implemented tools, Liang et al. identified the actors of AKM tools and generated a list of use cases [33]. They are grouped into four categories: producing AK, consuming AK, knowledge management and intelligent support. A use case defines a scenario that the user interacts with the system, which also shows what functionality is the system supposed to provide. They are an important part of requirement analysis to guide the direction of tool development. After defining the use case, seven tools were examined to see which use cases they cover. The results showed that the basic use cases are covered by most of the tools, while more complicated ones remain as future research.

These three papers form the basis of AKM requirements. They are selected based on the wide time range they cover and different research methods they adopt.

Requirements Elicitation Process

Firstly, requirements are elicited from the three papers independently. The second step is to remove duplication. Since each of them contains basic requirements such as capturing and storing AK, many requirements overlap with each other. The description of overlapping requirements is merged while covering the differences between them. It is further revised to form a unified format. Besides the description of a requirement, since its appearance in different works of literature indicates its importance, a matrix is created to keep track of this information. Each paper is represented by a column. If a requirement is covered by one paper, the corresponding cell is marked.

3.1.2 AKM Requirements Coverage

Table 3.1 lists the requirements as well as their coverage of the aforementioned three papers. 'P1' is the systematic research by Tang et al. [42], 'P2' is the chronological review by Capilla et al. [9] and 'P3' represents the use cases from Liang et al. [33]. For each requirement, notation 'Y' in the cell means the requirement is covered by the specific paper, while '-' means not.

Table 3.1: AKM Requirements Coverage

No.	Requirement description	P1	P2	P3
1	Capturing and adding AK [9, 2, 28, 29]: elicit and capture AK from various resources. Add and store them into the knowledge repository.	-	Y	Y
2	Editing AK [2]: modify or delete AK elements in the repository.	-	-	Y
3	Searching and retrieving AK [42, 29] search through the existing AK using certain criteria (e.g. keywords and categories).	-	-	Y
4	Distilling AK [42]: distill specific knowledge from a system into general knowledge (e.g. architecture pattern) that can be reused in future systems.	-	-	Y
5	Reusing AK [42, 9, 45]: the architect reuses AK in another project context, e.g. reusing architectural design decisions from an old to a new project.	Y	Y	Y
6	Sharing AK [9, 33, 1, 18]: share knowledge with one or more actors of the system.	-	Y	Y
7	Tracing AK [3]: trace between various AK elements, e.g. design decisions and rationale.	-	-	Y
8	Integrating AK [42, 33]: integrate different types of information into concrete AK, e.g. integrate stakeholder requirements, system context, and technology constraints into system requirements.	-	-	Y
9	Applying general AK [42, 33]: use application-independent AK(static knowledge), e.g. architecture patterns, architectural styles to solve the problems at hand.	-	-	Y
10	Comprehending AK [42, 33]: learn and comprehend the AK, e.g. understand the rationale of a design decision.	-	-	Y
11	Storing AK [42, 9, 28]: store context information such as requirements and quality attributes; store general information such as design patterns and architectural styles.	Y	Y	Y
12	Maintenance support [34, 35, 31]: analyse the impact of changes in design decisions.	Y	Y	-
13	Implementation support [42, 12, 43]: trace the implementation artifacts to design decisions.	Y	Y	-
14	Synthesis support [42, 15, 29, 34]: provide multiple potential solutions with description and rationale.	Y	-	Y

15	Customizing AK [42, 9, 1]: provide different users with different levels of permissions.	Y	Y	-
16	Integration with other tools [42, 9, 36]: automatic or semi-automatic integration with others tools during different phases of software lifecycle.	Y	Y	-
17	Collaborative environment support [42, 9, 16]: support concurrent access from multiple users.	Y	Y	-
18	Configurable environment support [42, 9]: provide a comprehensive and tailorable knowledge representation to allow organizations to support and evolve their AK. Provide organization-, domain-, or project-specific meta-models and tools.	Y	Y	-
19	Versioning AK [28, 20]: create and manage different versions of the AK.	Y	Y	Y
20	Identifying stakeholder [45, 28]: the architect identifies a stakeholder according to certain criteria, e.g. who has the most "weight" on the architectural decisions.	-	-	Y
21	Notification about AK changes [15]: subscribe to specific AK elements, and subsequently get notified about changes to them.	-	-	Y
22	(Semi-) automatic AK enrichment [29]: generate AK content proactively, e.g. automatically distilling and interpreting AK from text without the users' intervention.	-	-	Y
23	Viewing AK and their relationships [9, 29]: view both AK elements and relationships e.g. the architectural decisions made and the relationships between these decisions.	-	Y	Y
24	Automated decision-making support [15, 29]: provide automated support for architects during the process of making decisions, e.g. recommend experts for a certain decision or recommend alternative solutions.	-	-	Y
25	Architectural evaluation support [42, 9, 45, 14]: analyze trade-offs and risks of each solution. Perform a critical evaluation of the AK, e.g. to make sure that requirements have been satisfied in the architecture design.	Y	Y	Y
26	Reasoning capability [9]: basic inference mechanism that can automatically detect inconsistencies, state new questions or perform some fuzzy reasoning.	-	Y	-
27	Translating AK [32]: translate the formal AK based on a given AK domain model into another domain model to facilitate reuse.	-	-	Y

28	Support trade-off analysis [42, 45]: analyze the architecture by trading off different quality attributes.	Y	-	Y
29	Cleaning up architecture [45]: the architect makes sure that all the dependencies of removed AK (e.g. the consequences of an architectural decision) have been removed as well.	-	-	Y
30	Support design maturity assessment [45]: the architect evaluates when the architecture can be considered as finished, complete, and consistent, e.g. verify whether a system conforming to the architecture can be made or bought.	-	-	Y
31	Risk analysis [45]: identify flaws in the architecture.	-	-	Y

It can be noticed that the two systematic reviews have similar coverage over requirements. The difference in tools examined does not introduce much new content. This can be caused by the continuity of tool development. However, the use cases presented by Liang et al. [33] suggest many new potential research directions.

3.2 Architectural Design Decision Requirements

3.2.1 Requirements Elicitation

Architectural design decisions (ADDs) are a subset of AK. A design decision contains not only the result itself but also the process behind it. It is difficult to manage ADDs because of its complex nature: relationships. For example, one decision can be caused by another one. In this case, making a new decision is not an independent task. The previous decision has to be taken into consideration and the effect needs to be assessed. This kind of decisions becomes an important concept in decision making, which is name as 'cause' or 'driver' according to different models. To model and manage these aspects, it is necessary to understand the requirements for a supporting tool.

Literature Selection

Compared to AKM, architectural design decision making is a rather new field. The new perspective to view software architecture as a set of ADDs [23] was brought up by Jansen et al. in the year 2005. Despite the short history of ADD research, there are several models and tools developed, including Kruchten's ontology [27], Archium [25] and AREL [43]. In the survey performed by Shahin et al. [38], nine models including the aforementioned ones are examined together with their tool support if applicable. The core functionality of each model/tool is summarized, which forms the basis of requirements elicitation.

Once ADDs are captured and stored, the next question is how to utilize them. Van Der Ven et al. [45] constructed a use case model based on industrial needs. These use cases clearly lay an emphasis on the decision-making process. They are categorized according to the actor of the use case, which can be either an architect, an architecture reviewer, a project manager or a maintainer. Furthermore, three layers of abstraction are used to describe the use cases: summary, user-goal, and sub-function. The summary is the high-level goal of the system, which gives an overview of the purpose of the system. User-goal focuses on users' interaction with the system and the tasks they want to perform. Sub-function is low-level details about system functionality. Considering that summary is too general, while sub-function contains too many details, the middle layer, user-goal, is more appropriate for our research purpose.

Besides two pieces of research from the broad view of managing ADDs, the Archuim tool [25] focuses on a smaller but highly prioritized group of use cases. Nine out of twenty-seven use cases are selected based on their occurrences in interview reports [25].

These three papers form the basis of ADD requirements. They are selected in a way to cover both research and industrial needs.

Requirements Elicitation Process

The process is similar to the one described in Section 3.1: elicit requirements independently, remove duplication and then fill in coverage matrix.

3.2.2 ADD Requirements Coverage

Table 3.2 lists the requirements as well as their coverage of the aforementioned three papers. 'P1' is the survey from Shahin et al. [38], 'P2' represents the list of use cases by Van Der Ven et al. [45] and 'P3' is requirements of tool Archium by Jansen et al. [25]. For each requirement, notation 'Y' in the cell means the requirement is covered by the specific paper, while '-' means not.

Table 3.2: ADD Requirements Coverage

No.	Requirement description	P1	P2	P3
1	Support decision import and export [37]: allow information exchange between different systems.	Y	-	-
2	Searching design decisions [32]: search and filter design decisions by role, phase and scope attributes.	Y	-	-

3	Architectural decision recovery [45]: the architect reconstructs decisions with their associated rationale from an existing or 3rd party system.	-	Y	-
4	Support complex representation of architectural decisions [42, 9, 45]: use ontologies to represent the complex nature of architectural decisions, as well as their dense inter-dependencies.	-	Y	-
5	Architectural decision visualization[45, 30]: visualize architectural decisions and their relations from a number of different viewpoints, depending on the stakeholders' concerns.	Y	Y	-
6	Identifying architectural driver [45]: identify which architectural drivers have the most influence on the design of the system.	-	Y	-
7	Affected stakeholders identification [45]: identify which stakeholders will be affected by changes in decisions.	-	Y	-
8	Retrieving architectural decision [45, 25, 32, 43]: given the architectural model (or a part of it), trace back to the architectural decision it is based on. Provide the architectural drivers and the rationale of the decisions.	Y	Y	Y
9	Support stakeholder-specific overview [45]: identify key architectural decisions and unresolved concerns for a specific stakeholder.	-	Y	-
10	Check implementation against architectural decisions [25]: check how is the implementation in line with the architectural decisions, so as to know where in the development process people ignore or disregard the made decisions.	-	Y	-
11	Pattern detection of architectural decision dependencies [45, 25]: identify patterns in the graphs of decisions that can be interpreted in a useful fashion, or lead to guidelines for the architects. For example: decisions being hubs ("Godlike" decisions), circularity of a set of decisions, and decisions that gain weight over time and are thus more difficult to change or remove.	-	Y	Y
12	Completeness check [45, 25]: check if the requirements are all sufficiently covered by the decisions that are taken.	-	Y	Y
13	Consistency check [45, 25, 38]: check if the current set of architectural decisions is internally consistent. Check if the chosen alternatives have inconsistent consequences on the architectural model.	Y	Y	Y

14	Architectural decision consequence analysis [45, 25, 43]: the main consequences of a decision are the changes in the model when a decision is executed. In addition, new decision topics can be introduced.	Y	Y	Y
15	Superfluous architectural decision detection [45, 25]: check decisions that overlap (i.e. are redundant), e.g. parts of the decisions describe the same, or decisions do not affect the current architectural model at all (i.e. are unnecessary).	-	Y	Y

Since the papers selected here have different perspectives, the resulted requirements do not overlap with each other as much as in Table 3.1. But requirements number 8: retrieving architectural decision, 13: consistency check and 14: architectural decision consequence analysis are still covered by all of the papers, which indicates their high relevance for both researches and industrial needs.

3.3 Practitioner Feedback

To make a tool that is practical and usable for practitioners, three employees working in the architecture definition department in a large industry company were interviewed. Interviewee A is a software architect, interviewee B is a project leader and interviewee C is a project manager. Each of them has more than ten years of experience in the field of software engineering. They are provided with the lists of requirements which are presented in Table 3.1 and Table 3.2 and asked to label whether each of the requirements is relevant to their work. This process is done individually to avoid the influence on each other.

After receiving all three responses, the results are firstly compared against each other. Each of them ticked almost all requirements from the provided, although many of the requirements were not in their initial proposal. This can result from the fact that users do not know what they want until they are provided with it. It should be noted that they are only provided with requirements, not a system fulfilling the requirements. This leads to a less accurate expectation from the system.

On the other hand, they did not mention any new requirements that were not covered in the lists. This indicates that the requirements capture most of the aspects of decision support for software architecture.

Then the responses are merged in the following way: for requirements with three identical responses, the result is decided directly; for those with different responses, the majority is chosen. No requirements contain three different responses. The result gives insight into industrial needs. However, it is important to realize that such a list

can be too broad for a single system. According to different usage scenarios, some requirements need to be modified or neglected.

3.4 Researcher Feedback

Although all requirements are first gathered from studies and researches, they can still differ from the research area of researchers at the sebis chair[1]. To understand this difference, one researcher from the chair is provided with the lists and asked to label whether each requirement is relevant. The result shows that fundamental and highly prioritized requirements are within the research scope, while the advanced ones are not taken into consideration yet. This insight can guide future research directions as well as reflect on the current research interests.

3.5 Current Status of AMELIE

AMELIE has been under development for several years. As a prototype for supporting architectural design decision making, new services are gradually developed and integrated. Since AMELIE evolves over time, it is important to keep track of the system status for continuous development and maintenance. At the time point of this thesis, a check over the functionality of AMELIE concludes the current status and can be used to guide further steps of research.

To achieve a more objective result, both the three practitioners from the industry partner and the researcher at sebis are asked to perform an evaluation. The practitioners are given access to the system to evaluate the requirements against the tasks they can perform. The researcher is one of the developers of AMELIE, which indicates a deep knowledge of the system. To note down the results, the same requirements lists are provided and they are asked to label whether a requirement is already fulfilled in AMELIE. This process is done individually to avoid influences on each other.

Once receiving the four responses, they are firstly compared. Interviewee B, C, and the researcher have similar opinions on the current status of the system. Most of the basic requirements are labeled with yes, while advanced ones are either partially supported or not supported. However, interviewee A gives a very different result. For requirements that are labeled as supported by other interviewees, they are mostly labeled as partially supported or not supported by interviewee A. The reason for this can be the interviewee's strictness about evaluation. On the other hand, as a software architect, interviewee A has more experience in dealing with design decisions. His

[1] https://wwwmatthes.in.tum.de/pages/t5ma0jrv6q7k/sebis-Public-Website-Home

evaluation is based on the assistance he could get for the work of an architect, which can be demanding for a research prototype.

In the end, the results are merged based on a majority rule, which is mostly decided by interviewee B, C, and the researcher.

3.6 Final Results of Requirements Elicitation

3.6.1 Notation

For AKM and ADD, a list of requirements is consolidated based on the process described in Section 3.1 and 3.2. They are further extended with practitioner feedback, researcher feedback and the current status of AMELIE. The final list consists of five columns: requirement number, description, industry partner, sebis, and AMELIE. To improve the readability of wide tables, the abbreviation 'SM' is used to represent the industry partner, 'SB' for 'sebis' and 'A' for 'AMELIE'.

Column 'SM' shows whether a requirement is relevant and helpful to practitioners' work; column 'SB' means whether a requirement is relevant to the research area at the chair; column 'A' means whether the functionality described in the requirement is already implemented in AMELIE. There are three possible values for each cell: 'Y' means relevant or required, 'P' means partially relevant, and '-' means irrelevant or unimportant.

3.6.2 AKM Requirements List

The AKM requirements list consists of thirty-one requirements. They are ordered based on their priorities, which is decided in the following way: the more papers mention one requirement, the higher priority it has. It can be noticed that high-priority requirements are mostly relevant to both the industry partner and sebis and are more probable to have already been implemented.

However, some requirements are desired but not implemented, for example, AKM-28: support trade-off analysis. This can be due to the scope limit of the system. Trade-off analysis can be useful to identify the strength and weakness of an architecture, but an automatic analysis introduces many other new requirements. Some other implemented requirements are relevant to practitioners but not to sebis. A potential reason for this phenomenon is the gap between practical needs and scientific research directions.

Table 3.3: AKM Requirements Relevance

No.	Requirement description	SM	SB	A
1	Capturing and adding AK [9, 2, 28, 29]: elicit and capture AK from various resources. Add and store them into the knowledge repository.	Y	Y	Y
2	Editing AK [2]: modify or delete AK elements in the repository.	Y	Y	Y
3	Searching and retrieving AK [42, 29]: search through the existing AK using certain criteria (e.g. keywords and categories).	Y	Y	Y
4	Distilling AK [42]: distill specific knowledge from a system into general knowledge (e.g. architecture pattern) that can be reused in future systems.	Y	Y	P
5	Reusing AK [42, 9, 45]: the architect reuses AK in another project context, e.g. reusing architectural design decisions from an old to a new project.	Y	Y	Y
6	Sharing AK [9, 33, 1, 18]: share knowledge with one or more actors of the system.	Y	Y	Y
7	Tracing AK [3]: trace between various AK elements, e.g. design decisions and rationale.	Y	Y	P
8	Integrating AK [42, 33]: integrate different types of information into concrete AK, e.g. integrate stakeholder requirements, system context, and technology constraints into system requirements.	Y	Y	Y
9	Applying general AK [42, 33]: use application-independent AK(static knowledge), e.g. architecture patterns, architectural styles to solve the problems at hand.	Y	-	Y
10	Comprehending AK [42, 33]: learn and comprehend the AK, e.g. understand the rationale of a design decision.	Y	Y	Y
11	Storing AK [42, 9, 28]: store context information such as requirements and quality attributes; store general information such as design patterns and architectural styles.	Y	-	Y
12	Maintenance support [34, 35, 31]: analyze the impact of changes in design decisions.	Y	-	P
13	Implementation support [42, 12, 43]: trace the implementation artifacts to design decisions.	Y	Y	P
14	Synthesis support [42, 15, 29, 34]: provide multiple potential solutions with description and rationale.	Y	Y	Y

15	Customizing AK [42, 9, 1]: provide different users with different levels of permissions.	Y	-	Y
16	Integration with other tools [42, 9, 36]: automatic or semi-automatic integration with others tools during different phases of software lifecycle.	Y	Y	Y
17	Collaborative environment support [42, 9, 16]: support concurrent access from multiple users.	Y	Y	Y
18	Configurable environment support [42, 9]: provide a comprehensive and tailorable knowledge representation to allow organizations to support and evolve their AK. Provide organization-, domain-, or project-specific meta-models and tools.	Y	Y	Y
19	Versioning AK [28, 20]: create and manage different versions of the AK.	P	Y	Y
20	Identifying stakeholder [45, 28]: the architect identifies a stakeholder according to certain criteria, e.g. who has the most "weight" on the architectural decisions.	Y	Y	Y
21	Notification about AK changes [15]: subscribe to specific AK elements, and subsequently get notified about changes to them.	P	Y	Y
22	(Semi-) automatic AK enrichment [29]: generate AK content proactively, e.g. automatically distilling and interpreting AK from text without the users' intervention.	Y	Y	Y
23	Viewing AK and their relationships [9, 29]: view both AK elements and relationships e.g. the architectural decisions made and the relationships between these decisions.	P	-	-
24	Automated decision-making support [15, 29]: provide automated support for architects during the process of making decisions, e.g. recommend experts for a certain decision or recommend alternative solutions.	Y	Y	P
25	Architectural evaluation support [42, 9, 45, 14]: analyze trade-offs and risks of each solution. Perform a critical evaluation of the AK, e.g. to make sure that requirements have been satisfied in the architecture design.	Y	-	P
26	Reasoning capability [9]: basic inference mechanism that can automatically detect inconsistencies, state new questions or perform some fuzzy reasoning.	Y	-	-
27	Translating AK [32]: translate the formal AK based on a given AK domain model into another domain model to facilitate reuse.	P	-	-

28	Support trade-off analysis [42, 45]: analyze the architecture by trading off different quality attributes.	Y	-	-
29	Cleaning up architecture [45]: the architect makes sure that all the dependencies of removed AK (e.g. the consequences of an architectural decision) have been removed as well.	Y	-	-
30	Support design maturity assessment [45]: the architect evaluates when the architecture can be considered as finished, complete, and consistent, e.g. verify whether a system conforming to the architecture can be made or bought.	Y	-	-
31	Risk analysis [45]: identify flaws in the architecture.	Y	-	-

Compare this table with Table 3.2, it is clearly noticed the requirements with a high priority are implemented at an early stage. These requirements include fundamental functionality such as capturing, adding, editing, searching and sharing AK. They form the basis upon which AKM can be performed.

Other requirements address how to manage and benefit from AK. For example, AKM-5 describes reusing ADDs from an old project to a new one. In an industrial setting, practitioners tend to build applications with similar technologies in their respective fields. As a result, similar scenarios and design decisions can occur repeatedly. Providing them with the decisions they made in the past helps them to reflect on past experiences and analyze current potential consequences. The same applies to scientific research. However, it is not sufficient for practitioners and researchers to only know the result, the decision that was made. Since the contexts are usually not exactly the same, the rationale behind a decision has a significant influence on the final result. This requires traceability between design decisions, rationale and other AK elements as stated in AKM-7. The rationale of a design decision can only be interpreted correctly together with its context. So AKM-8 requires integrating different types of information into the system, including stakeholder requirements, system context, and constraints.

In addition to the requirements that are either fundamental or crucial to an AKM tool, more advanced requirements are also covered in the list. Starting from number twenty-six, they have not drawn much attention yet, but they can give some idea about the future work. For example, AKM-26 requires an inference mechanism for automatic inconsistency detection or question creation. Once such an artificial intelligence is involved in an AKM tool, architects are freed from examining potential contradictions as they are automatically detected in a more efficient way. Considering that inconsistencies can reside between decisions, this inference mechanism is able to detect dependencies of decisions. This fulfills AKM-29 which describes how to clean up an architecture. As a software evolves over time, some design decisions are changed or removed. If their dependencies remain in the system, they could cause confusion since architects do not

know whether it is safe to remove them or not. From this perspective, cleaning up the architecture improves the maintainability of a software.

3.6.3 ADD Requirements List

The ADD requirements list consists of fifteen requirements. The decrease in number is caused by the overlapping scope of these two topics. Since ADD is a subset of AK, the general use cases are already covered in the previous section. In this section, only those requirements that focus on ADD are discussed.

Considering the fact that not all three papers are systematic reviews, the priorities cannot be decided in the same way as AKM requirements. As a result, the complexity of a requirement is chosen as the ordering criteria. This helps to check whether the basic requirements are covered by the system or what advanced requirement can be the next step of research.

Table 3.4: ADD Requirements Relevance

No.	Requirement description	SM	SB	A
1	Support decision import and export [37]: allow information exchange between different systems.	Y	Y	Y
2	Searching design decisions [32]: search and filter design decisions by role, phase and scope attributes.	Y	Y	P
3	Architectural decision recovery [45]: the architect reconstructs decisions with their associated rationale from an existing or 3rd party system.	Y	Y	Y
4	Support complex representation of architectural decisions [42, 9, 45]: use ontologies to represent the complex nature of architectural decisions, as well as their dense inter-dependencies.	Y	Y	Y
5	Architectural decision visualization[45, 30]: visualize architectural decisions and their relations from a number of different viewpoints, depending on the stakeholders' concerns.	Y	Y	Y
6	Identifying architectural driver [45]: identify which architectural drivers have the most influence on the design of the system.	Y	Y	Y
7	Affected stakeholders identification [45]: identify which stakeholders will be affected by changes in decisions.	Y	Y	P

8	Retrieving architectural decision [45, 25, 32, 43]: given the architectural model (or a part of it), trace back to the architectural decision it is based on. Provide the architectural drivers and the rationale of the decisions.	Y	Y	Y
9	Support stakeholder-specific overview [45]: identify key architectural decisions and unresolved concerns for a specific stakeholder.	Y	Y	Y
10	Check implementation against architectural decisions [25]: check how is the implementation in line with the architectural decisions, so as to know where in the development process people ignore or disregard the made decisions.	Y	-	-
11	Pattern detection of architectural decision dependencies [45, 25]: identify patterns in the graphs of decisions that can be interpreted in a useful fashion, or lead to guidelines for the architects. For example: decisions being hubs (”Godlike” decisions), circularity of a set of decisions, and decisions that gain weight over time and are thus more difficult to change or remove.	Y	Y	-
12	Completeness check [45, 25]: check if the requirements are all sufficiently covered by the decisions that are taken.	P	-	P
13	Consistency check [45, 25, 38]: check if the current set of architectural decisions is internally consistent. Check if the chosen alternatives have inconsistent consequences on the architectural model.	Y	-	P
14	Architectural decision consequence analysis [45, 25, 43]: the main consequences of a decision are the changes in the model when a decision is executed. In addition, new decision topics can be introduced.	Y	-	P
15	Superfluous architectural decision detection [45, 25]: check decisions that overlap (i.e. are redundant), e.g. parts of the decisions describe the same, or decisions do not affect the current architectural model at all (i.e. are unnecessary).	P	-	-

The fundamental requirements are all relevant to both the industry partner and sebis chair. For example, as described in ADD-4, ontologies are used to express the complex nature of ADDs as well as their inter-dependencies. Considering the complicated relationships a design decision has with other AK elements, a careful modeling is essential to utilizing design decisions. And ADD-6 emphasizes the identification of the architectural driver for a design decision. It is considered to be a basic requirement as

the architectural driver is the key to understand a design decision.

The latter half of the list presents advanced requirements that are out of the current research scope. ADD-12: completeness check and ADD-13: consistency check focus on the analytic view of design decisions. These tasks are mainly performed by architects instead of the tool because of their complexity, but they are still partially covered in the implementation of AMELIE. Though they are not directly supported by the system, users can get relevant information from the system to assist in their manual checking. The same applies to ADD-14, which requires automatic consequence analysis.

It is noticeable that the focus on utilizing ADDs ranges from making decisions to analyzing decisions. While the former one is considered to be more important and relevant at the moment, the latter one provides more insight and will be highlighted in future research.

4 Enhancing User Interfaces

User experience is the overall experience of end-users' interaction with the system. An important part of user experience is the usability of a system, which includes the ease of learning, the efficiency to use and the satisfaction while interacting with UI [22]. A positive user experience helps users to fulfill their needs and keeps them loyal to the product, which is crucial to the success of the product.

In this chapter, usability issues in AMELIE project are identified using a usability-inspection method. They are then solved during the implementation phase, which is described in the last section.

4.1 Expert Review Method

A design review is a usability-inspection method to identify potential usability problems in a system [19]. There are two main types of methods: heuristic evaluation and expert review. Heuristic evaluation requires the reviewer to check the compliance of the system with a set of heuristics such as Jakob Nielsen's 10 Usability Heuristics. Although this method is commonly used in academic researches, it can be too rigid at times. Since user interfaces have various goals, fulfilling the heuristics does not guarantee meeting the needs of clients. In other words, the assessment should be dependent on the context of the user interfaces. In addition to heuristic evaluation, expert reviews are another good alternative.

An expert review is carried out by a user experience expert from the perspective of end-users to identify usability issues. The emphasis is laid on the experience and expertise of the reviewer. The expert usually not only checks the compliance with heuristics but also other guidelines and related researches. This makes the assessment more flexible and more customized according to the specific context of the system.

Considering the user-centered design of AMELIE, an expert review is chosen as the method to review the UIs.

4.2 Expert Review Process

The first step of an expert review is to understand the end-users of the system. In AMELIE, end-users are architects and developers who seek assistance in making design decisions. Afterward, their typical goals are also identified. Then the expert review is performed based on the workflow of end-users. Usability issues are gathered along the way. These issues also reflect one or more system requirements proposed in Chapter 3. In the end, a list of issues is created. This forms the basis for user experience improvement.

4.3 Results

The user experience issues are grouped based on their corresponding UI pages including projects, import, quality attributes, architectural elements, expertise matrix, expert recommender, design decisions, and decision details page. They are also available on the GitHub page of the Amelietor [1]. For each issue, the following structure is used: **Description** is the spotted issue from the expert review, **Related requirements** shows the unmet requirements according to the requirement lists, and **Solution** includes the implementation to solve the issue as well as how this improves the usability of the system.

4.3.1 Projects

On this page, end-users are able to browse all projects. They can search for a specific project using its name, description or category. After finding the project, clicking on the project redirects them to the details of that project.

Issue 1: Incomprehensible data table

Description Figure 4.1 shows the initial implementation of the list of projects in a data table. It consists of the name, description, category, and issue count of all projects. The data table is difficult to understand because of the following reasons. The checkboxes in the first column are very misleading. Checkboxes usually indicate that users can select the item and perform some actions, which is not true in this case. What's more, different types of elements are mixed in the '# issues' column. It contains both the total number of issues and the state of project import.

[1] https://github.com/sebischair/amelietor

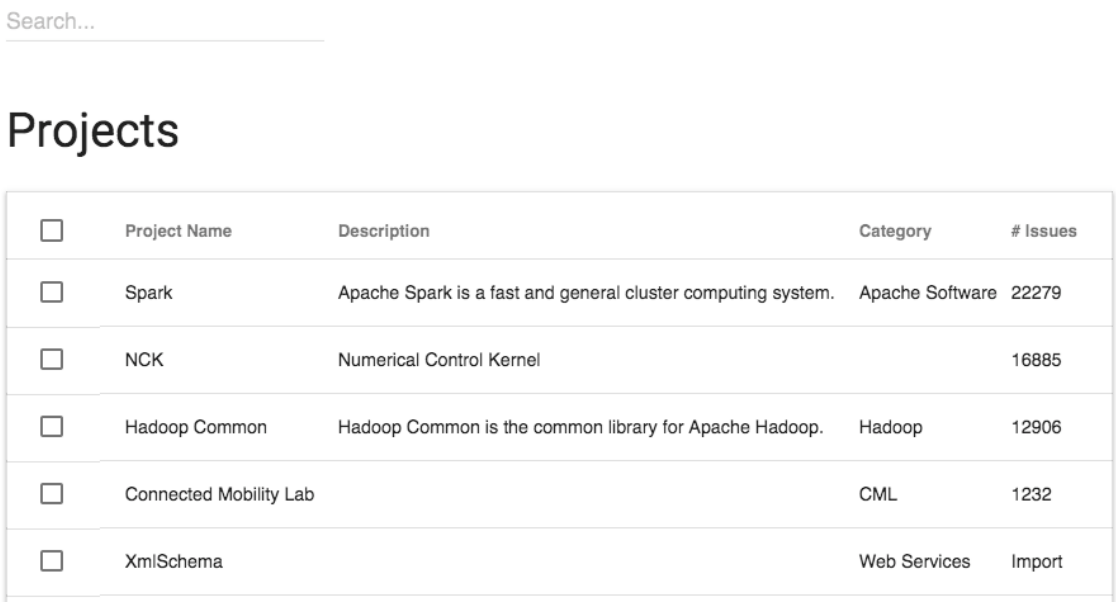


Figure 4.1: Initial State of Projects

Related requirements

- AKM-1: Capturing and adding AK [9, 2, 28, 29]: elicit and capture AK from various resources. Add and store them into the knowledge repository.
- ADD-1: Support decision import and export [37]: allow information exchange between different systems.

This page is the starting point of any user action. With the aforementioned problems, it is difficult for users to figure out where and how to import a project, which fails to meet these two requirements.

Solution

1. Move the search field closer to the form according to the principle of proximity. Proximity means to arrange related UI elements together and unrelated elements apart [40]. It indicates which elements are connected and gives users visual hint about their relationships.
2. Remove misleading checkboxes and import buttons
3. Replace input cursor with a pointer when hovering over a column header

4. Enable clicking on a row to open the project
5. Provide a longer description to ease searching

The misleading elements on the page are either removed or replaced to follow the design principle of consistency and standards. This reduces the learning effort for users. Users can utilize their past experience without confusion.

Issue 2: Low performance

Description The performance of the data table is pretty poor. The scrolling delay is around one second, which is unacceptable for web applications.

Solution

1. Replace data table component with Material-UI library
2. Add pagination to the table

This issue is caused by the library in use, namely react-mdl[2]. This library has been deprecated since March of 2017. Library deprecation can cause compatibility and performance problems as it is not maintained anymore. In this case, a new library is chosen to replace the old one. After comparing and researching different alternatives, Material-UI[3] is selected because of high quality of its components and fast release cycles. Material-UI is a react library that follows the material design principles from Google. It provides most of the commonly used components like tables, lists, and buttons. After replacing the component, pagination is added to allow faster data loading. Users can configure the size of pagination according to their needs. Now data is loaded gradually as users go through other pages, which reduces the waiting time.

Before & After Comparison

Figure 4.2 shows the changes after solving the aforementioned issues. The new UI preserves the basic design while improving the understandability of the UI elements.

4.3.2 Import

When users open a project that is not imported yet, they will be redirected to the import page. A step-by-step guide instructs them about how to import the project and process all the design decisions.

[2]https://github.com/tleunen/react-mdl
[3]https://github.com/mui-org/material-ui

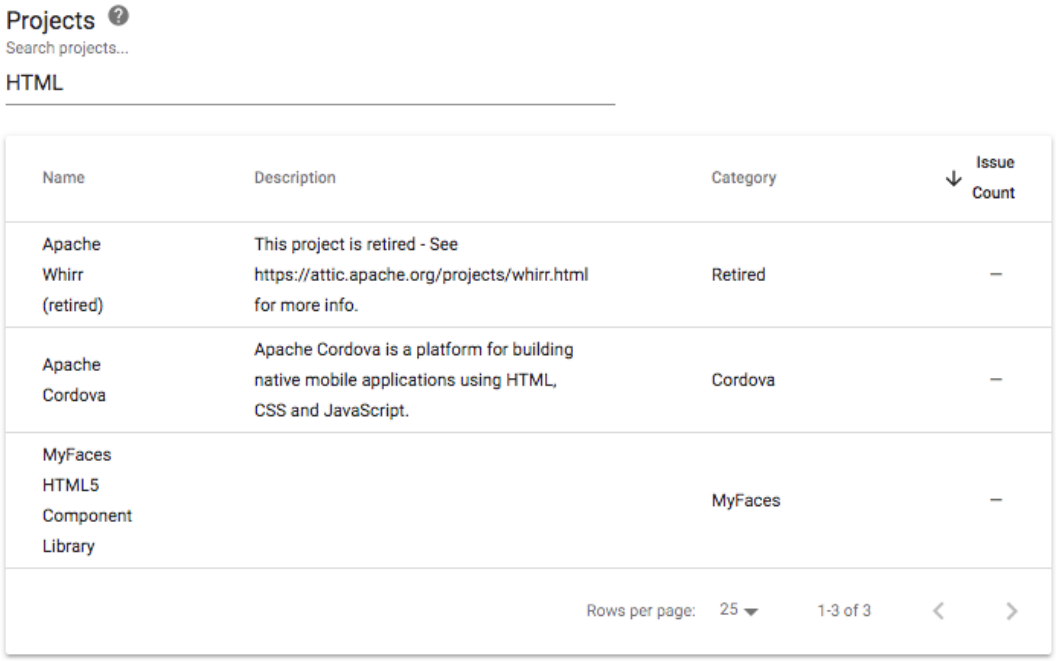


Figure 4.2: Redesigned Data Table of Projects

Issue 3: Confusing steps and action buttons

Description The steps numbering is unclear. In Figure 4.3, it is visible that a step 1.1 exists between step 1 and 2, which only provides status update. This is not a sub-step of the first action. Secondly, the action buttons also cause confusion. They are labeled with nouns like 'SyncPipes', not verbs. Buttons should indicate an action to perform. A noun does not provide information about what is supposed to happen after clicking on the button.

Related requirements

1. AKM-1: Capturing and adding AK [9, 2, 28, 29]: elicit and capture AK from various resources. Add and store them into the knowledge repository.
2. AKM-11: Storing AK [42, 9, 28]: store context information such as requirements and quality attributes; store general information such as design patterns and architectural styles.
3. AKM-16: Integration with other tools [42, 9, 36]: automatic or semi-automatic integration with others tools during different phases of software lifecycle.

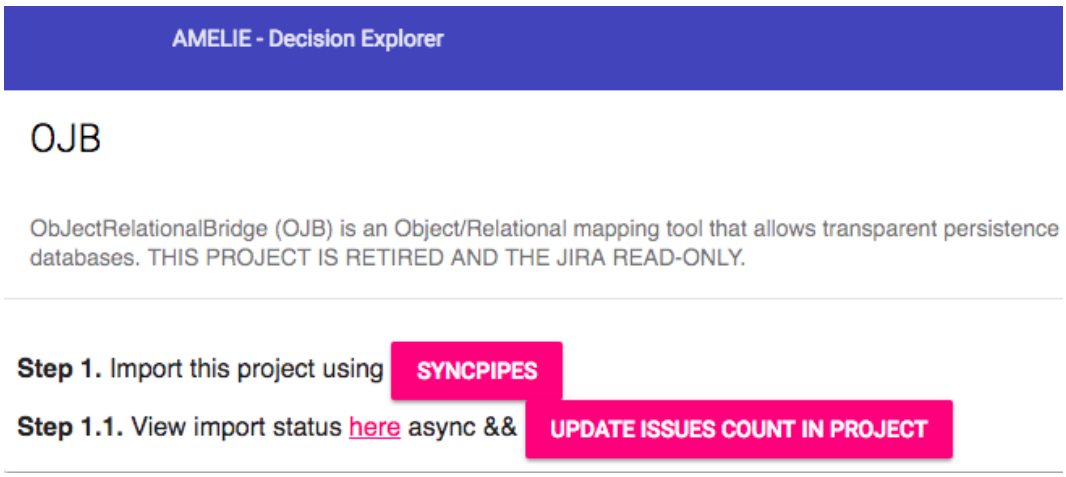


Figure 4.3: Initial State of Import

The import functionality is implemented in the SyncPipes project, which is integrated into AMELIE. This requirement is considered to be important as it increases the efficiency of architects. They no longer need to enter the same information twice, once in the issue management system and once in the AKM tool.

Solution

1. Replace plain steps with a stepper component
2. Rename buttons with corresponding verbs
3. Enable continuing from previous importing step

The whole stepping logic is reimplemented with the stepper component. In the old stepper, there was no step state. Each step has a conditional logic to decide whether it is displayed or not. This approach is error-prone as the condition changes. With a stepper component, all logic is maintained within the stepper. This provides a reliable state for the current step. The steps are also simplified to two steps with clearer information.

Issue 4: Manual state checking in a second system

Description In step 1.1, users have to go to another website to check the import status. This must be done manually, which requires users' attention while waiting for the results.

Solution

1. Remove manual status checking step
2. Pull pipeline status from SyncPipes server proactively
3. Provide current status information
4. Advance the stepper automatically when the previous one is done

The manual status checking is replaced by an automatic status update. Now users do not need to use a second web application and wait for the result. The stepper updates the status proactively and displays a loading message. Once the import is finished, the stepper automatically advances to the next step.

Before & After Comparison

Figure 4.4 shows the changes on project import page. The stepper is completely redesigned to form a logical flow of actions.

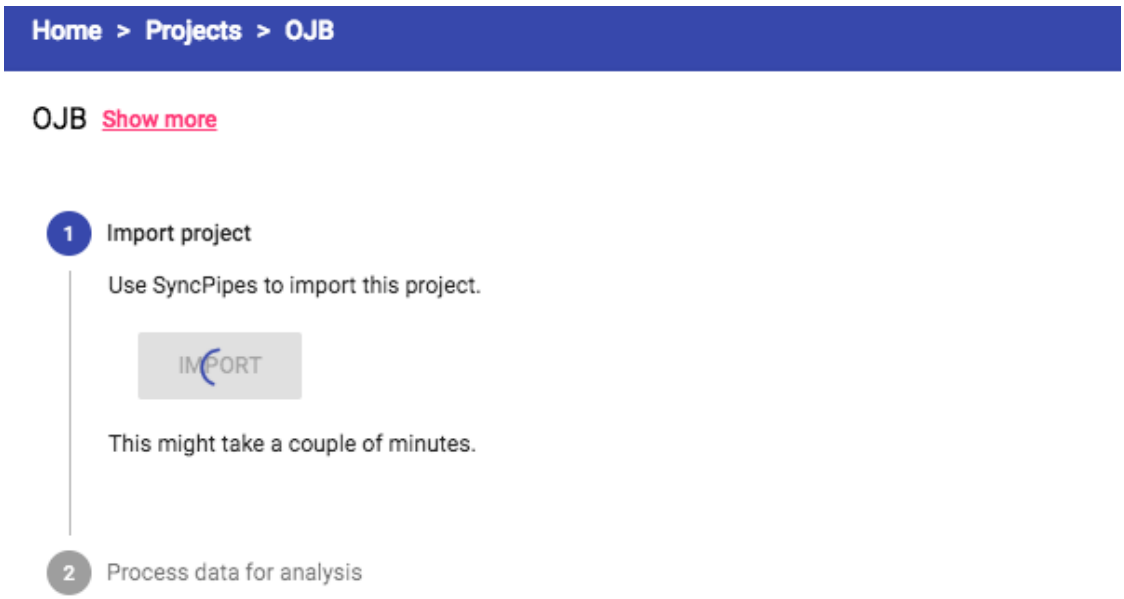
4.3.3 Quality Attributes

Quality attributes include non-functional requirements such as performance, security, availability, and usability. They can be addressed by design decisions to improve the overall quality of the system. Missing quality attributes indicate potential problems.

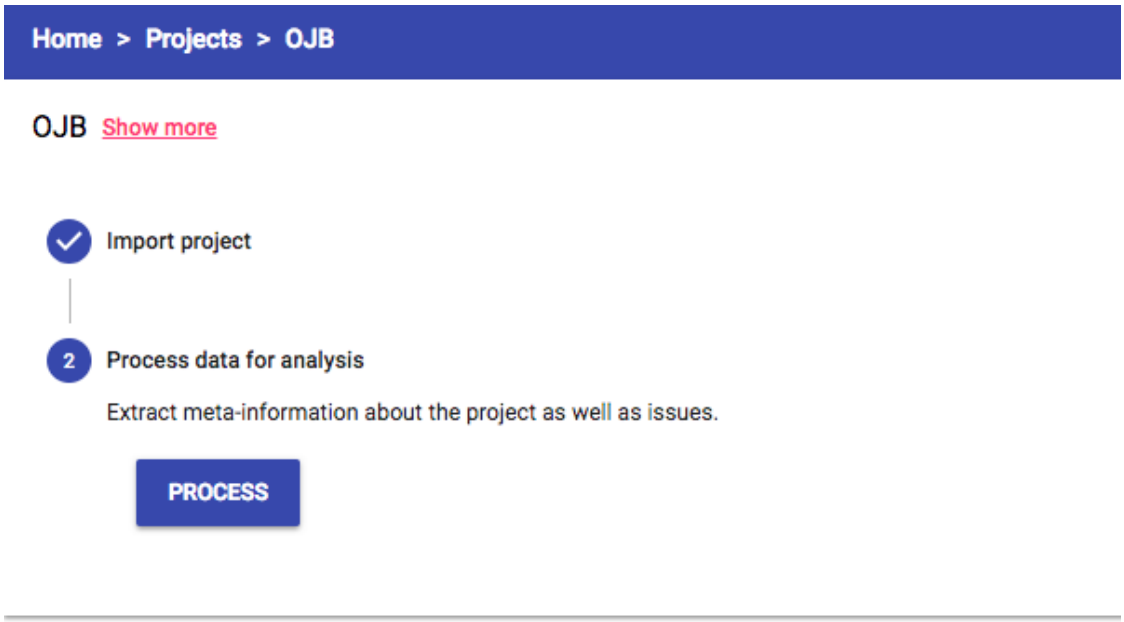
The bar chart in Figure 4.5 shows how many design decisions address each quality attribute. The design decisions are further categorized into behavioral, structural and non-existence decisions. In addition, a time slider provides a chronological view of all design decisions.

Issue 5: Visibility problem of the bar chart

Description The stacked bar chart has a fixed height, which makes it not readable on small screens as shown in Figure 4.5. This figure is a full screenshot, but only the upper part of the bar chart is visible. When there is no data, the text 'no data' is displayed in the middle of the empty SVG element. This text information is not visible even on medium-sized screens. Besides the visibility problem, the cursor does not indicate the click action on the chart by turning into a pointer. It is possible to click on the bars and get redirected to the page of corresponding design decisions, yet there is no visual hint about this possible action.



(a) Import the Project



(b) Process Imported Issues

Figure 4.4: Redesigned and Reimplemented Import Stepper

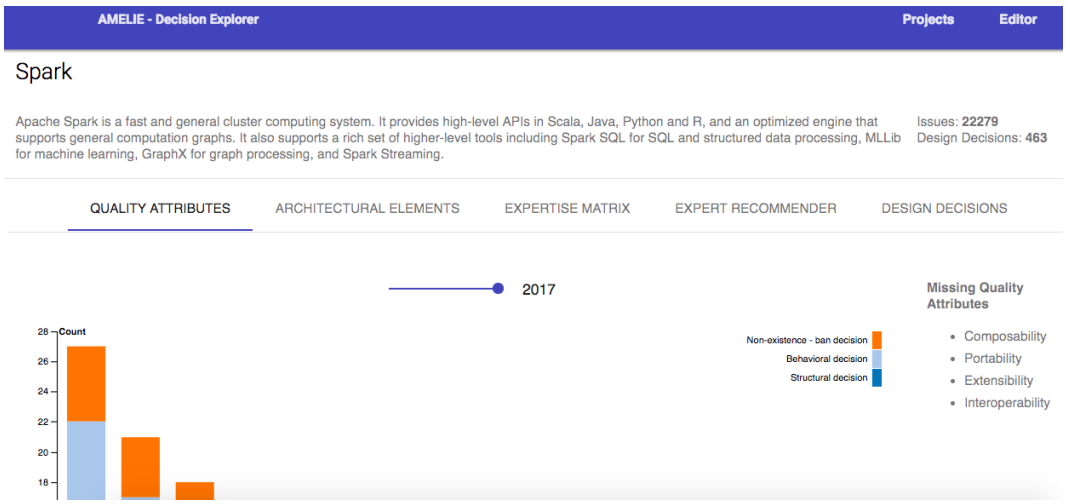


Figure 4.5: Initial State of Quality Attributes

Related requirements

1. ADD-2: Searching design decisions [32]: search and filter design decisions by role, phase, and scope attributes.
2. ADD-5: Architectural decision visualization[45, 30]: visualize architectural decisions and their relations from a number of different viewpoints, depending on the stakeholders' concerns.

Visualization enables users to digest a huge amount of data in an easily understandable way. This chart also allows searching for design decisions based on their year and categories.

Solution

1. Replace deprecated library components
2. Reduce the fixed height of the image
3. Add pointer cursor to bars

The image generator requires a fixed height container, which makes it hard to have a responsive design for the chart. However, after experimenting with several different situations, it was found that a smaller fixed height is satisfactory in most of the cases.

4.3.4 Architectural Elements

Architectural elements include architectural styles, methods, design patterns, and software and hardware systems [6]. They are annotated for each design decision by the DBpedia Spotlight component and visualized on this page. A time slider provides a chronological view of design decisions grouped by architectural elements.

Issue 6: Visibility problem of the bubble chart

Description This issue is similar to issue 5 with a different perspective of visualization. The bubble chart has a fixed height which is barely visible on small and medium-size screens as shown in Figure 4.6. Although this figure is a full screenshot, only the upper part of the chart is visible. When there is no data, the explanation text is off the screen which gives users no clue of the current situation.

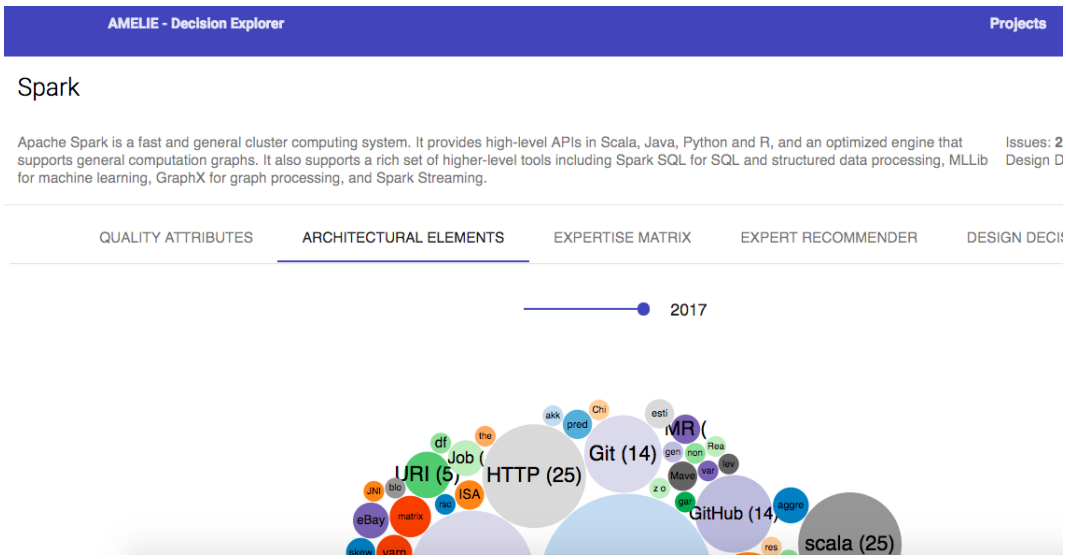


Figure 4.6: Initial State of Architectural Elements

Related requirements

1. ADD-2: Searching design decisions [32]: search and filter design decisions by role, phase, and scope attributes.
2. ADD-5: Architectural decision visualization[45, 30]: visualize architectural decisions and their relations from a number of different viewpoints, depending on the stakeholders' concerns.

Solution

1. Replace deprecated library components
2. Adjust SVG size
3. Add pointer cursor to circles

4.3.5 Expertise Matrix

In an expertise matrix, rows represent experts and columns correspond to architectural elements. A cell within the matrix indicates the expertise of an architect with respect to an architectural element. Searching for an expert will show all the fields the expert has dealt with while searching for architectural elements suggests all experts who can solve those issues that address corresponding architectural elements.

Issue 7: Oversized matrix

Description The huge matrix overflows its container in both horizontal and vertical directions. Figure 4.7 shows that scrolling the matrix results in scrolling the whole page, including the navigation bar and tabs.

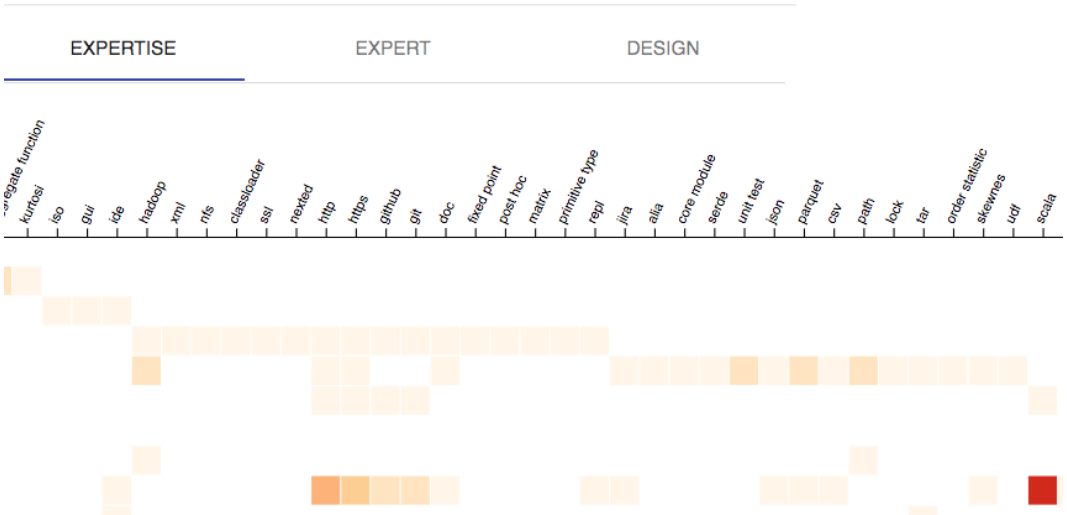


Figure 4.7: Initial State of Expertise Matrix

Related requirements

1. AKM-24: Automated decision-making support [15, 29]: provide automated support for architects during the process of making decisions, e.g. recommend experts for a certain decision or recommend alternative solutions.

Although the support is not automated yet, users can get recommendations for experts by manual searching.

Solution

1. Replace deprecated library components
2. Enable scrolling only on the matrix element

The improvements made here are straightforward based on the problems identified.

4.3.6 Expert Recommender

Unlike the expert matrix, expert recommender provides recommendations for design decisions. Each design decision has several expert candidates. Every candidate is assigned an expert score, which indicates their past experience in solving similar problems.

Issue 8: Problematic recommendation column

Description As shown in Figure 4.8, for each design decision, five experts are displayed by default. The redundant padding between expert names expands the height of a row. Thus each row has little content compared to its size. Additionally, the meaning of expert scores is unclear.

Solution

1. Reduce padding between expert names
2. Collapse experts and display two by default. Expand when clicking on 'show all' button.
3. Add tooltip to explain the meaning of expert scores.

A denser layout enables users to browse more data without the need to scrolling. And two alternatives are usually good enough for people to make decisions.

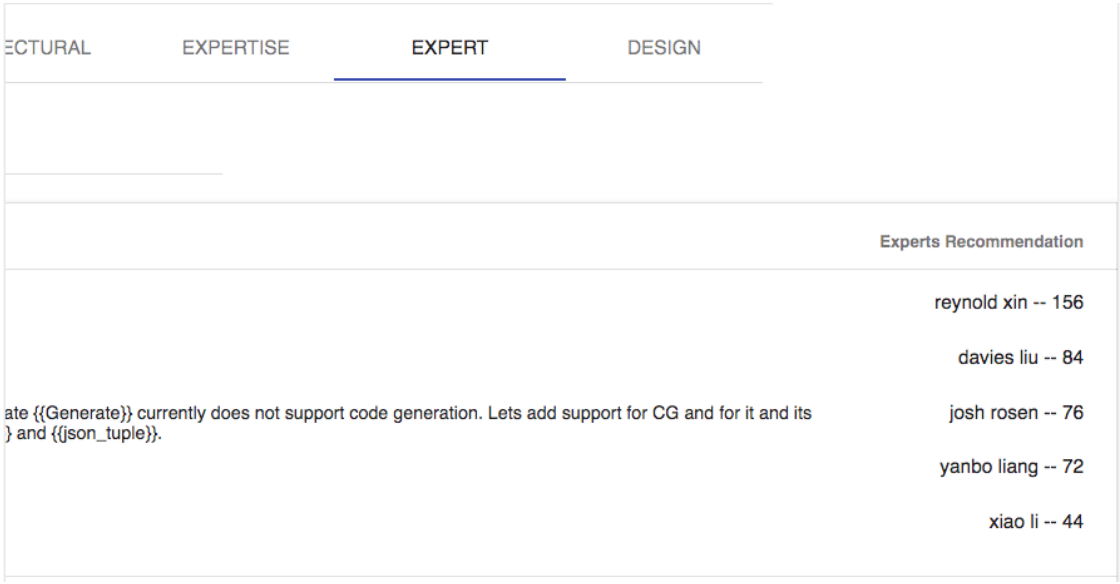


Figure 4.8: Initial State of Expert Recommender

Issue 9: Low performance

Description Similar to issue 2, the performance of the data table is poor.

Related requirements

1. AKM-24: Automated decision-making support [15, 29]: provide automated support for the architect in the process of making decisions, e.g. recommend experts for a certain decision or recommend alternative solutions.

Solution

1. Replace data table component with Material-UI library
2. Add pagination to the table

Before & After Comparison

Figure 4.9 shows the improved expert recommender page. Compared to the old design, extra information of experts is collapsed to provide a condensed view.

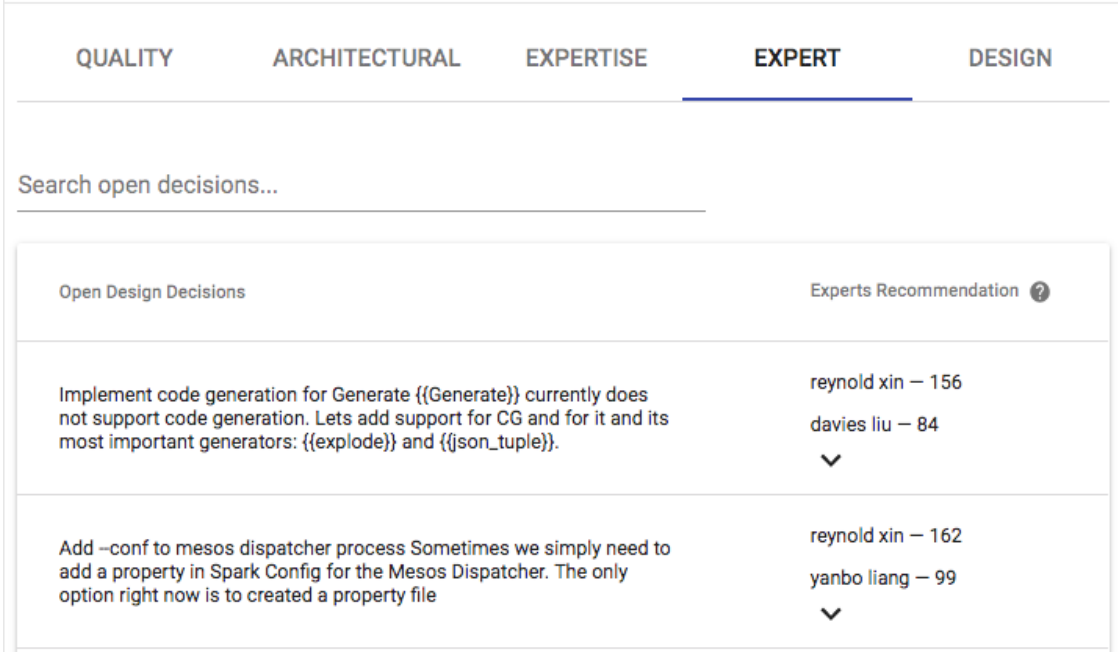


Figure 4.9: Redesigned Data Table of Expert Recommendations

4.3.7 Design Decisions

All design decisions of the project are displayed in a data table. Users are able to search and filter design decisions according to their needs.

Issue 10: Incomprehensible design decisions data table

Description This data table has similar usability problems as issue 1: confusing checkboxes in the first column and wrong cursor type, which is visible in Figure 4.10.

Solution

1. Remove misleading checkboxes
2. Replace input cursor with a pointer when hovering over a column
3. Enable clicking on a row to open the project
4. Provide a longer description to ease searching

The misleading elements on the page are either removed or replaced to follow the design principle of consistency and standards.

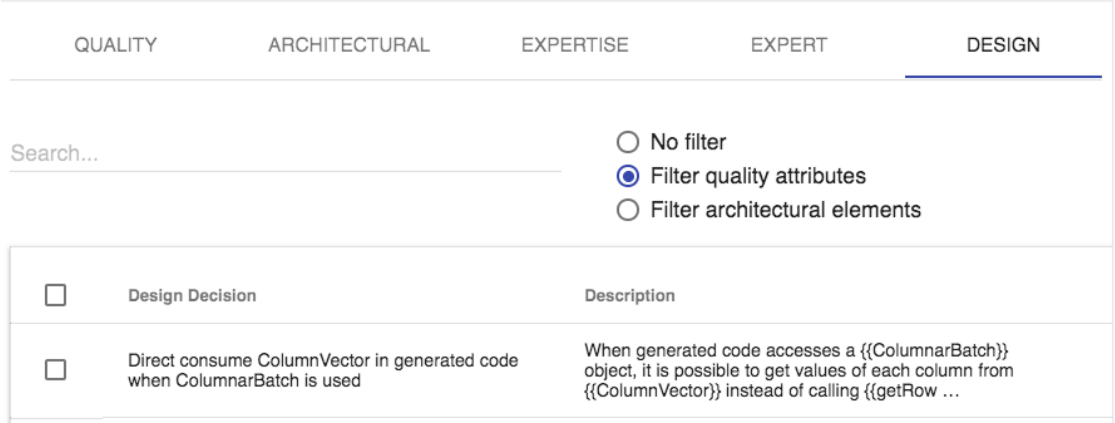


Figure 4.10: Initial State of Design Decisions

Issue 11: Low performance

Description Similar to issue 2, the performance of the data table is poor.

Solution

1. Replace data table component with Material-UI library
2. Add pagination to the table

Issue 12: Little control over filtering options

Description The current filters can only filter with/without quality attributes or architectural elements as shown in Figure 4.10. This provides little value for users. The more useful scenario would be to filter different values. For quality attributes, users should be able to select multiple required quality attributes. For architectural elements, a search field is sufficient. On the other hand, the filters are currently positioned in a bad way which makes end-users clueless.

Related requirements

1. ADD-2: Searching design decisions [32]: search and filter design decisions by role, phase and scope attributes.

The newly designed filtering options enhance users' control over search options. Once attributes like role, phase and scope are associated with design decisions, it is feasible to adapt the filter for them as well.

Solution

1. Redesign filtering options
2. Implement new filters

The old filters are replaced with three new filtering options. The first one is a search field for design decision descriptions. The second one is a drop-down menu for selecting quality attributes. The third one is a search field for architectural elements. Only results that fulfill all requirements are displayed. This provides higher flexibility for users. They can either search with any requirement or pinpoint specific design decisions accurately.

Before & After Comparison

Figure 4.11 shows the changes of Design Decisions page. The filters are redesigned while the data table is reimplemented.

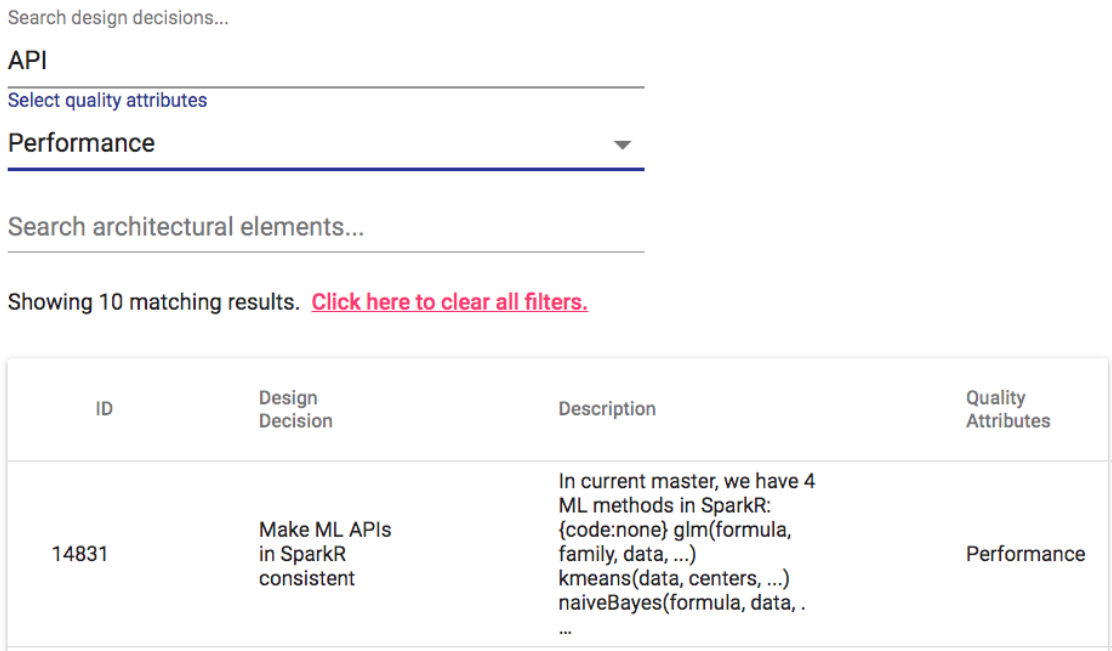


Figure 4.11: Advanced Filtering Options for Design Decisions

4.3.8 Design Decision Details

Clicking on a design decision redirects users to the design decision details page. Users can view the description of a decision and annotate the architectural elements within the text. For an architectural element, its meta-information is provided together with software solutions and alternatives. Similar decisions made in the past are also displayed. This enables users to reuse the knowledge and solve the current design concern.

Issue 13: Mispositioned action button

Description Figure 4.12 shows that the annotation button has a fixed position directly above the text, which makes part of the text unreadable. It is also mixed with the progress bar. When users click on the button, it gets replaced by the progress bar. The disappearance of the button can cause confusion for users.

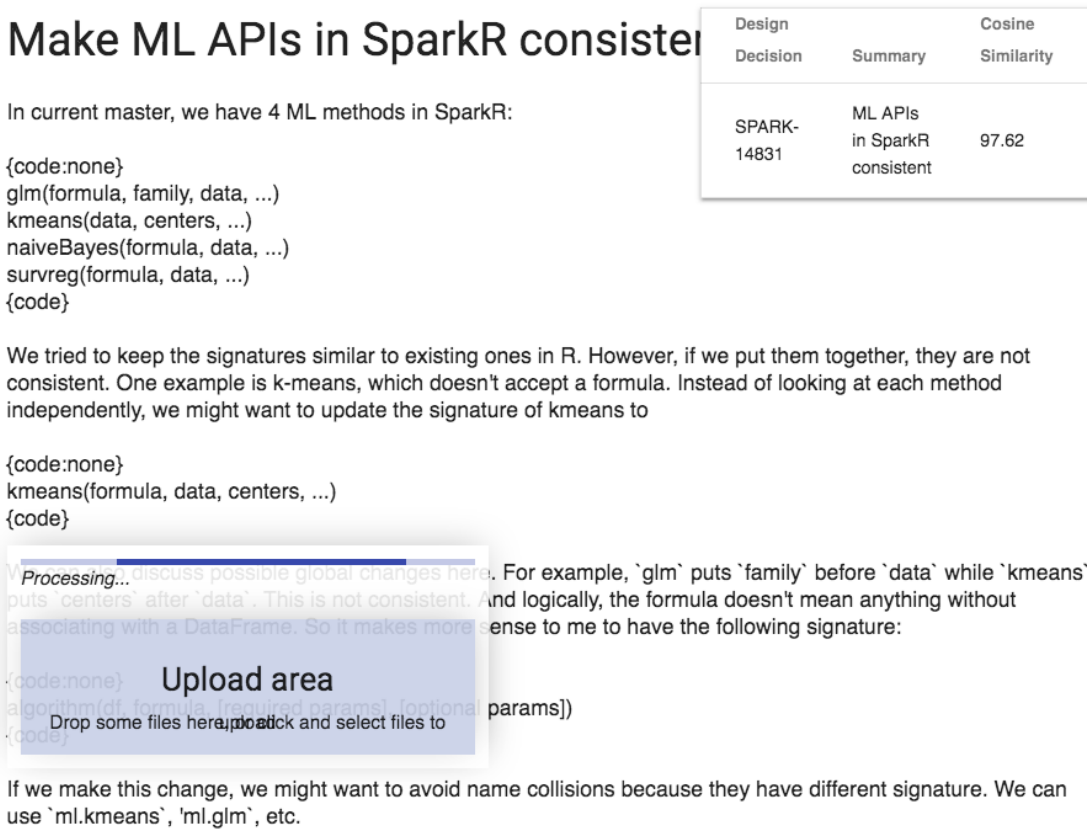


Figure 4.12: Initial State of Decision Details

Related requirements

1. AKM-1: Capturing and adding AK [9, 2, 28, 29]: elicit and capture AK from various resources. Add and store them into the knowledge repository.

Once the text is annotated, users can click on an architectural element to see meta-information, software solutions and alternatives. They can also add new content or delete an old one. All changes are saved in the database and can be retrieved later.

Solution

1. Create footer action bar for 'annotate' button
2. Separate button and progress bar

The main focus is to create a dedicated component for action buttons. This improves extendability of adding new functions and buttons later. The separation also makes the meaning of each component clearer.

Issue 14: Long text being cut off

Description The container of architectural element information expands according to the length of the content. For some elements, the meta-information can be quite lengthy, which stretches over the whole page. In general, it is enough to provide a short preview of the context. Users can decide whether they would like to continue and read the full content. The problem is more serious with software solutions and alternatives information. There is too much padding between the elements, which makes the list too long to read.

Related requirements

1. AKM-24: Automated decision-making support [15, 29]: provide automated support for the architect in the process of making decisions, e.g. recommend experts for a certain decision or recommend alternative solutions.

Information retrieved from the DBpedia Spotlight component is processed to provide software solutions as well as alternative solutions. Users are provided with further information about each solution so that they can compare and trade-off before making a decision.

Solution

1. Collapse lengthy meta information. Click on 'see more' to expand the card.
2. Reduce padding between list items

A condensed text card enables users to get more information without the need to scroll.

Before & After Comparison

Figure 4.13 shows the changes of Decision Details page. The action bar is redesigned to improve understandability. The text cards are condensed for a better readability.

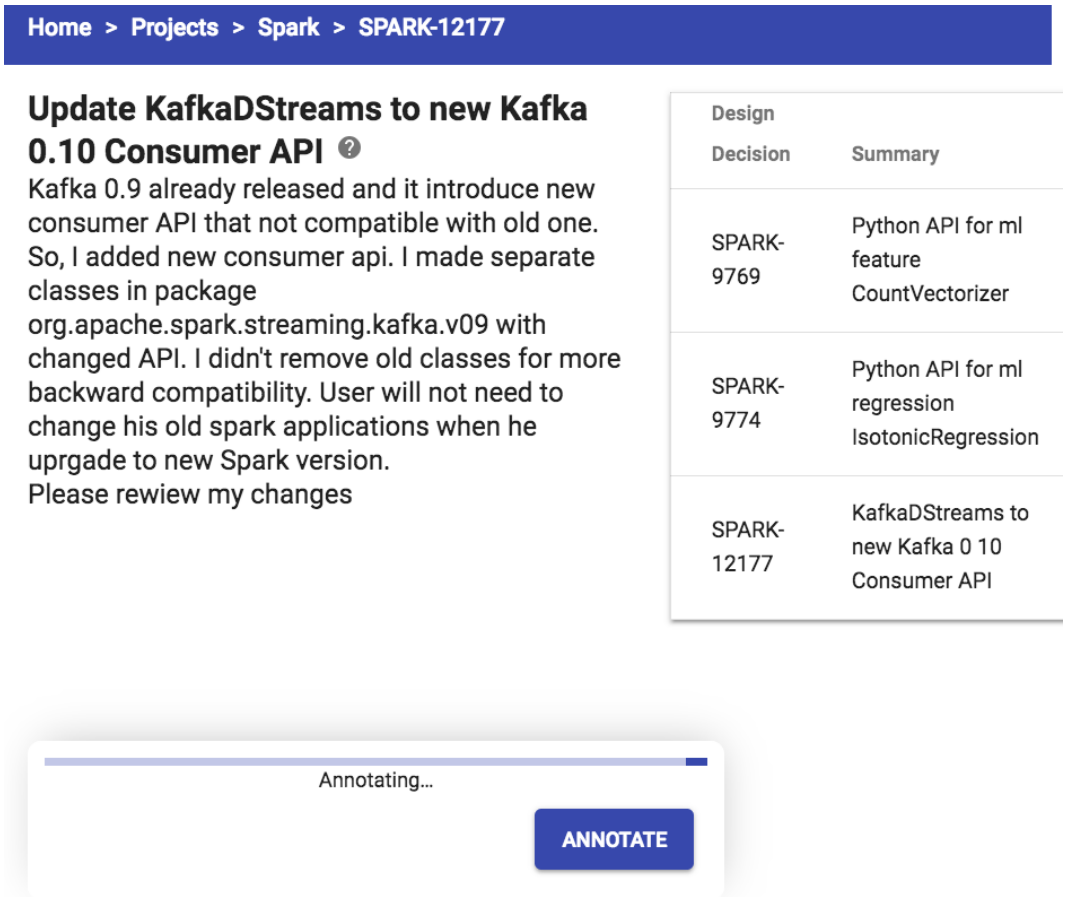


Figure 4.13: Redesigned Action Bar and Layout

5 Containerization

AMELIE consists of several microservices which are developed separately with different technologies. Deploying them one by one demands to learn each of them and is time-consuming. The same problem exists for developers who want to customize and extend AMELIE. In a collaborative environment, it is even more troublesome to configure all microservices under different set-ups. Containerization is a solution to these problems.

5.1 Introduction

Docker is introduced in this section.

5.1.1 Docker Container

Docker [1] is a containerization technology that enables the creation of containers. Containers package an application with all its dependencies like libraries and then ship it as a whole. This happens at the application layer, which is different from virtual machines as shown in Figure 5.1. Virtual machines (VMs) are an emulation of a computer system. They reside at the physical layer and provide the functionality of a physical computer. Since each VM includes its own operating system, running multiple VMs requires a lot of resources. This is a severe problem, especially for microservice architecture. Each microservice needs to be deployed to a dedicated VM. Scaling microservices leads to performance overhead. But with docker container, a microservice is only packed with its dependencies. Multiple containers run on the same hosting operating system and share resources, which makes it light-weight and faster.

On the other hand, Docker containers are based on open standards, which means they can run on various operating systems: Windows, Mac, Linux distributions and other operating systems. Developers no longer need to read types of different documentation to set up their working environment. Once the setup file is created, they can use it to have the exact same development environment. This helps to keep a unified and stable environment for each developer and saves time as well. Docker container also makes deployment easier and faster. A Dockerfile is sufficient to deploy the container to cloud providers.

[1] https://www.docker.com/

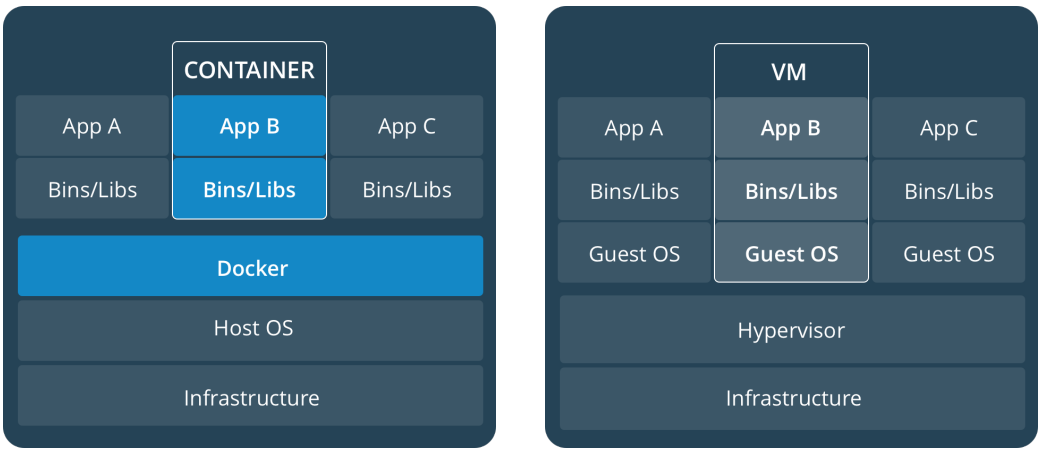


Figure 5.1: Container Architecture vs. Virtual Machine Architecture [21]

To sum up, docker improves the efficiency of microservice execution, development, and deployment. These aspects are considered to be usability for developers.

5.1.2 Docker Image

Docker containers are instantiated from docker images. A docker image is defined by a Dockerfile, which contains multiple layers to create the image. There are several commonly used instructions in a Dockerfile:

- FROM: specifies the base image which serves as the ground layer of the image. A Dockerfile must start with a 'FROM' instruction.
- RUN: executes a command and creates a commit.
- CMD: specifies the command to run within the container. It is usually used as the entry point to start the application.
- EXPOSE: specifies the ports the container listens on.
- COPY: copies files and directories to the filesystem of the container.
- VOLUME: creates a mount point to share files and directories between the local filesystem and the container.

Usually, it is not necessary to write a Dockerfile from scratch. Many Dockerfiles are published online for applications developed in various programming languages and frameworks. They are sufficient to run the container with some customization.

5.1.3 Docker Compose

A microservice architecture requires to run multiple services at the same time. Docker compose is a tool to manage running multiple containers simultaneously. All services are defined in a docker-compose YAML file. A service can contain following configurations:

- build: specifies the path to build the service.
- image: specifies the remote image to be used to start the container.
- ports: maps the host ports to container ports.
- depends_on: defines the dependency between different services. Start a service will automatically start the services it depends on first.
- environment: defines environment variables.

5.2 Containerize AKRE-Server

Containerizing a microservice requires two steps. Firstly, an image to run the service needs to be created. Secondly, other services that communicate with this service need to be specified in the docker-compose file.

5.2.1 Create Dockerfile

AKRE-Server is built with Play Framework written in Java. The first layer is an image for Java applications. Such an image can be found on Docker Hub, a central place for docker registries. Then Play Framework is fetched and installed in a new layer. Afterward, the code of the application is copied to the filesystem of the container. By defining the entry point to execute the code, the application will be started once the container runs.

For applications developed with the same technology, a dockerfile can be reused as they run in the same environment.

5.2.2 Create docker-compose File

A docker-compose file defines multiple services that are necessary to run the application. It starts by defining the docker-compose version, which is related to the docker engine version. The current major version of docker-compose is 3. Then each service is

specified individually. AKRE-Server is built from the code directly. To run AKRE-Server, MongoDB is necessary. This service uses the official image published on Docker Hub. By defining that AKRE-Server depends on MongoDB, running this docker will start MongoDB first.

Listing 5.1: docker-compose for AKRE-Server

```
1  version: "3"
2  services:
3    akre:
4      tty: true
5      build: ./
6      ports:
7        - "9000:9000"
8      depends_on:
9        - mongo
10   mongo:
11     image: "mongo:3.6.2"
12     ports:
13         - 27017:27017
```

5.3 Shared Data Storage

Dockers for other microservices are created in the same way as described in the previous section. Each docker can run independently and be replaced by the normal stand-alone application. However, when all of them run in docker and want to communicate with each other, one problem arises: they share a database. Since AMELIE has already been running for some time, a database dump is created to restore data in dockers. So far each docker restores data separately, which means a database dump is included in the build. A database dump can take up to hundreds of megabytes. It is also problematic to keep consistency among all these copies.

To solve this problem, a dedicated container is created for restoring the database. It has two services: mongo and mongoforamelie. Service mongo serves as the database and service mongoforamelie is responsible for restoring data into mongo.

The restoration process is defined in the dockerfile as shown in listing 5.2: line 1 specifies the container runs on MongoDB. Line 3 creates a directory for database dumps, and line 4 to 7 copies the dumps from the current directory to the container's filesystem. Line 9 to 16 restores all the databases from the dumps to the container. After this command is finished, container mongoforamelie will stop automatically. The

other container, mongo, will keep running will all the data loaded.

Listing 5.2: Restore Database from Dump Files

```
1  FROM mongo
2
3  RUN mkdir -p /data/archive
4  COPY archive/amelie_oss.gz /data/archive/amelie_oss.gz
5  COPY archive/docclassifier.gz /data/archive/docclassifier.gz
6  COPY archive/docclustering.gz /data/archive/docclustering.gz
7  COPY archive/syncpipes.gz /data/archive/syncpipes.gz
8
9  CMD mongorestore --host mongo --port 27017 --drop --gzip && \
10     --archive=/data/archive/amelie_oss.gz --db amelie_oss; && \
11     mongorestore --host mongo --port 27017 --drop --gzip && \
12     --archive=/data/archive/docclassifier.gz --db docclassifier; && \
13     mongorestore --host mongo --port 27017 --drop --gzip && \
14     --archive=/data/archive/docclustering.gz --db docclustering; && \
15     mongorestore --host mongo --port 27017 --drop --gzip && \
16     --archive=/data/archive/syncpipes.gz --db syncpipes;
```

This container is included as a new service in other dockers, which eases the process of data restoration. In addition, now all the data is kept in one place. Updating data involves only one container. The other microservices can pull the new image and they are all in synchronization.

5.4 One-click Deployment

Each microservice image is published to Docker Hub individually to accelerate development cycles. When a microservice is updated, the developer can simply push the new image again without affecting other services. Other developers can then pull and run the new container. This greatly reduces build time in comparison to rebuilding the whole system. It also increases the flexibility for continuous testing.

Although the images are published separately, they can easily form the whole system to enable one-click deployment. Before containerization, deployment is time-consuming and error-prone. Every microservice requires configuration for its environment and needs to be deployed one by one. Now with docker, a single container can serve as the central place for all these services. All the docker-compose files from the microservices are merged in the following way: containers no longer get built from local files, but rather use remote images; the ports, environment variables, and other configuration

stay the same; dependencies are added to ensure the starting sequence. Running the command 'docker-compose up' on this container will start all the services. By deploying this container to a cloud provider results in a running system without extra effort.

6 End-user and Developer Documentation

A documentation is meant to help users to understand an application. There are two types of users: end-users and developers. End-users want to utilize the application to achieve their goals, while developers want to maintain and extend the application according to their needs.

6.1 End-user Documentation

End-users employ the application to realize their goals. They are not interested in learning the technical details of the application. So an end-user documentation should show the necessary steps to accomplish their tasks. It should also be as concise as possible at the same time, as end-users would like to use the application immediately instead of reading long textual information.

Since AMELIE has a well-defined user workflow, a user tour is sufficient to help users with the application. A user tour is a visual step by step guide. It walks users through the app and provides contextual help, such as introducing features or highlighting important information. This improves the user's experience and the overall usability of the application.

6.1.1 Design Principles

Providing a user a guide does not mean going through all the elements in the user interface. There are several principles to design a tour guide.

First, a tour guide should not be a barrier between users and the application. Users would like to get their work done in an as short amount of time as possible without spending too much effort learning the application. They do not have the patience to go through long instructions and learn everything about the application right at the beginning.

Second, the information provided by the tour guide should be contextual. Contextual means to provide information that is relevant to the current state of the user. This principle keeps the information short but relevant.

Based on these two principles, the tour guide is designed according to the current page the user is on. Only the important and necessary information is included in the

guide. It does not require the user to go through the whole application on the first launch, but rather gives tips when the user visits certain pages.

6.1.2 Implementation

React Joyride[1] is a simple yet powerful React library for creating in-app tours. It provides necessary customization over properties, step content, styles, and callback. With Joyride, tours can be implemented individually within components.

For a tour, the steps and their corresponding content are firstly defined as an array. As the tour starts automatically when an end-user visits the page for the first time, a visiting state is saved in the local storage. When the component is mounted, it checks whether the tour should be triggered. During the tour, an end-user can decide whether it is desired to skip the tour or go back to previous steps. After the first visit, it is still possible to access the tour via an information icon.

6.1.3 Result

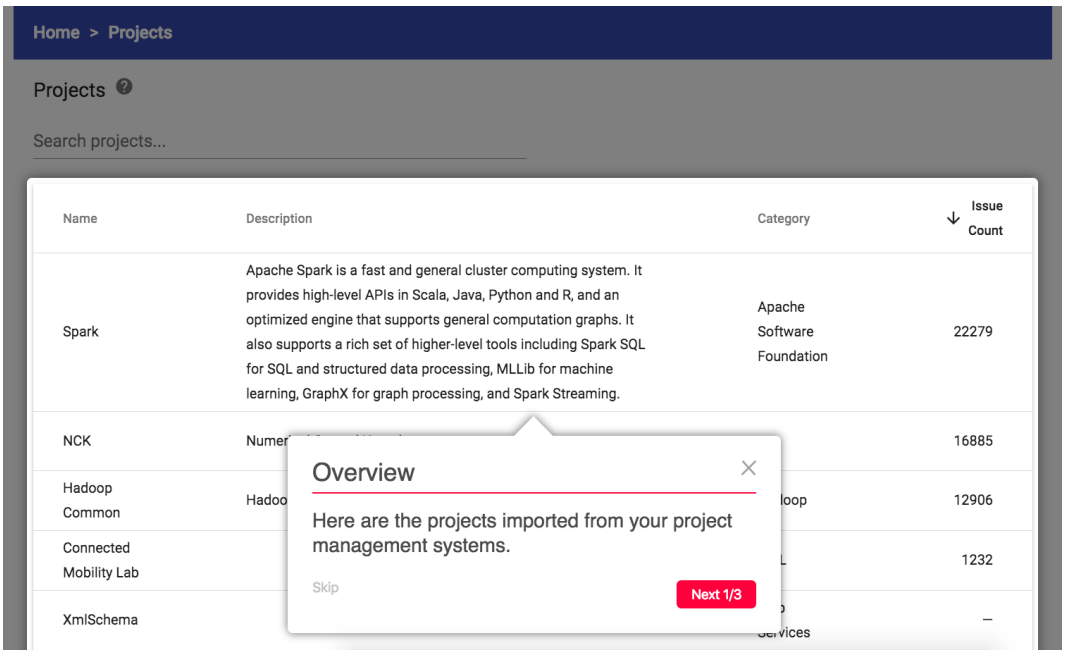
Figure 6.1 shows the implemented guided tour on the Projects page of AMELIE. The tour starts automatically when an end-user accesses this page for the first time. The element to be introduced is highlighted with a pop-up window which contains the guidance. An end-user can click on 'Next' to advance or 'Back' to return to the previous back, or skips the tutorial upon finding the information unnecessary. Closing the tour window interrupts it, and it can be continued by clicking on a red dot indicating the next step. If the tour is finished and the user wants to review it, clicking on the info icon next to the page title 'Projects' will restart the whole tour.

6.2 Developer Documentation

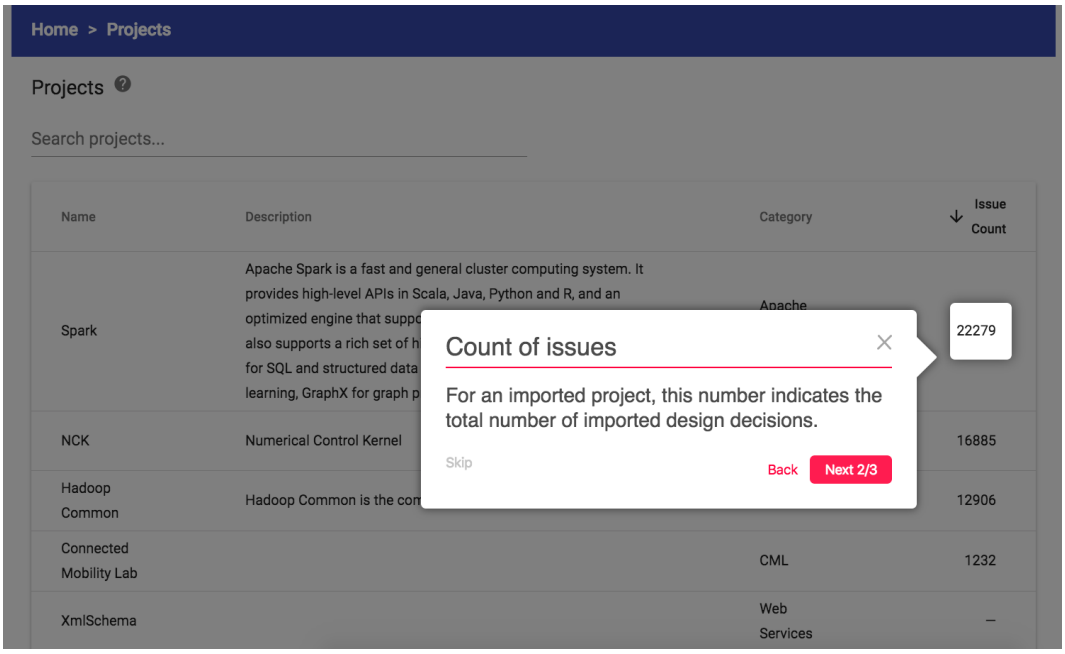
AMELIE is an open source research project, which means it can be customized and extended by developers according to their interests. Extending the application requires knowledge about its interfaces, especially for a complex system with multiple microservices. This information is provided by an API documentation. It contains detailed information about APIs, such as the purpose of the function, example requests, and responses.

On the other hand, AMELIE uses machine learning techniques to cluster and classify design decisions. There are pre-defined machine learning models that are ready to use, but it is also possible to create new models and tune them. Since creating machine

[1] https://github.com/gilbarbara/react-joyride



(a) Step One



(b) Step Two

Figure 6.1: A Step-by-step Guided Tour on Projects Page

learning models is a non-trivial task, extra documentation is created for Document Clustering and Document Classifier.

6.2.1 API Documentation

Postman[2] is an API testing tool. It has a graphical user interface to construct a request, send it to the server, and read the response. This interaction enables a developer to understand how an API works. It also generates an API documentation automatically in the form of a web page. The sent requests and their responses can be saved as examples for the documentation.

API documentation is created for AKRE-Server[3], Document Clustering[4] and Document Classifier[5]. All external APIs that are responsible for communication with other services are documented in the following steps:

1. Specify request type and URL: environment variables are used to enable testing in different environments.
2. Add parameters: parameters can also be stored in variables.
3. Create request body for POST requests.
4. Send request and save the response as an example: providing an example response in the documentation informs developers about what they should expect from an API.
5. Add API description: description gives more information about the purpose of the API, the expected request type, and response type.

After the API requests are created, they are grouped into different folders according to their functionalities. This eases the navigation when the documentation includes a large number of APIs. In the end, a web page is generated automatically by Postman and is published online for public access.

Figure 6.2 shows the generated documentation for AKRE-Server. On the left side is the navigation folders for easy access to specific parts of the documentation. The middle panel is the documentation of an API, including its title, URL, description, headers and example body if applicable. On the right side, an example of requests and responses is provided to give developers a direct impression of what the system expects

[2]https://www.getpostman.com/
[3]https://documenter.getpostman.com/view/4318985/RW84rWWK
[4]https://documenter.getpostman.com/view/4318985/docclustering/RW86LA17
[5]https://documenter.getpostman.com/view/4318985/docclassification/RW86LAES

and what it returns. With such a documentation, developers can find information about specific endpoints they want to connect to.

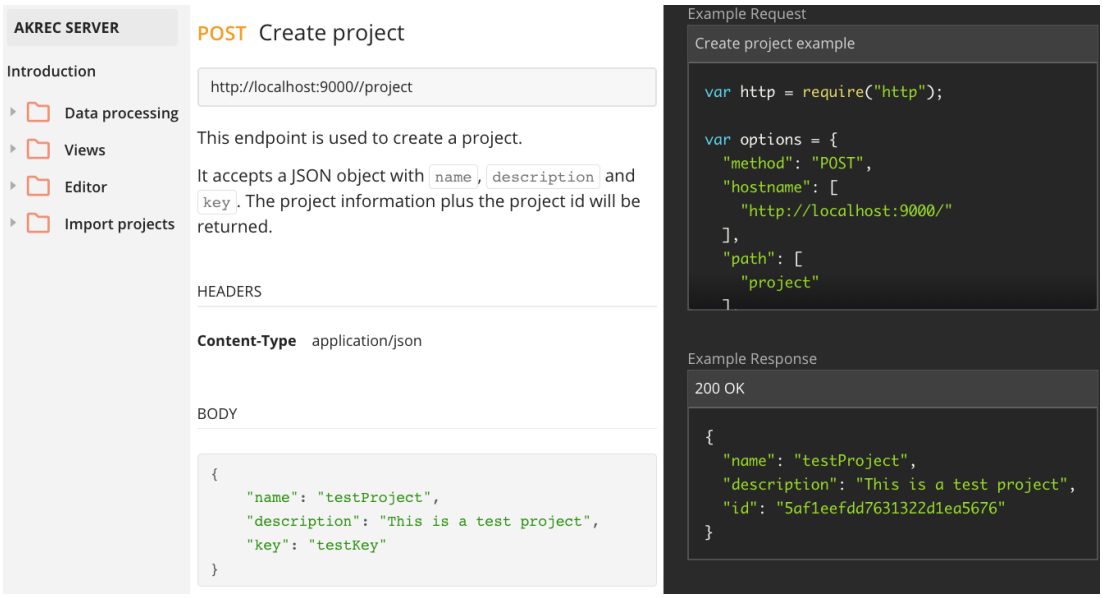


Figure 6.2: API Documentation for AKRE-Server

6.2.2 Machine Learning Model Configuration

Document Clustering and Document Classifier use machine learning techniques to process design decisions. Design decisions are categorized into structural, behavioral and non-existence decisions. Similar decisions are clustered together so that decisions made in the past can guide the current situation. These two microservices are invoked while importing a project into AMELIE. When an end-user imports a project in the client of AMELIE, these models are invisible. The end-user should not know the technical details behind the scene. It is the job of a machine learning expert to create, configure and manage the models.

Both of these two services have their own client application for this task. However, creating and tuning a machine learning model is a non-trivial task. It includes selecting the base project, machine learning attributes, algorithms and many other options. To ease this process for experts, a step-by-step instruction is created for Document Clustering[6] and Document Classifier[7]. It demonstrates the whole process of creating

[6]https://github.com/sebischair/DocClusteringWorkbench/wiki
[7]https://github.com/sebischair/DocClassification/wiki

an example model.

Since the instruction manuals are specific to these two services, they are kept in the individual GitHub repositories as Wiki pages. This keeps the technical details from end-users while providing an easy access for developers who visit the repositories for development instructions.

7 Conclusion and Future Work

This chapter concludes the whole thesis and proposes potential future steps.

7.1 Conclusion

The usability of a system is not only the usability of its user interface but also other aspects such as understandability, learnability, extensibility, and maintainability. This thesis presents several approaches to improve different aspects of the usability of a decision support system named AMELIE.

In Chapter 3, a requirement analysis was firstly performed based on a literature review. Considering the scope of the system, the requirements were divided into two parts according to their focus: architectural knowledge management and architectural design decision making. Then, practitioners and researchers were interviewed to understand the relevance of such requirements. AMELIE was also examined to figure out what requirements are covered by the system and what is not. The resulting lists of requirements showed that most of the requirements are considered relevant to practitioners. Some advanced requirements were not in the research scope at the chair, nor were they implemented in AMELIE, but they are potential future research directions.

Some implemented requirements did not fully function due to usability flaws in the user interfaces. In Chapter 4, an expert review was conducted to identify potential usability problems. They were solved by redesigning and reimplementing problematic UI elements. The before and after comparison showed the changes and improvements in UIs.

In Chapter 5, the usability of AMELIE was addressed by containerizing its microservices. Setting up the environment for all microservices one by one could be a tedious and error-prone process. With Docker, the environments were managed within the container, which ensured a unified set-up among all developers. To further simplify the deployment process, all microservices were merged into one container. Deploying this container led to a running application with just one click.

The last chapter presented multiple types of documentation created to improve the usability, extensibility, and maintainability of AMELIE. The documentation consisted of two parts: end-user guide and developer's documentation. The end-user guide

was incorporated into the UI in the form of a user tour. It provided informational and contextual help. API documentation was created with the tool Postman to help developers further work on the system. Extra instructions about creating and modifying machine learning models were written in Wiki pages of GitHub repositories.

7.2 Future Work

In the future work, the usability of AMELIE will be further enhanced from the following aspects.

Firstly, the identified requirements with high priorities should be implemented. According to Table 3.3, AKM-4: distilling AK and AKM-7: tracing AK are both highly prioritized yet partially supported by the system. Distilling AK means to transform project-specific knowledge into general knowledge that can be applied under other circumstances. This helps architects to discover new architectural styles or patterns that they were not aware of before. Patterns describe reoccurring problems as well as their solutions. The recognition of a pattern enables utilizing best practices to solve the new problem. The other requirement, AKM-7 demands traceability between design decisions, rationale, and other AK elements. Since design decisions are dependent on each, one design alone is not sufficient to understand the broad view of the overall architecture. Traceability establishes the connections between design decisions for retrospective analysis. What's more, traceability also helps to understand the rationale of a design decision. It can only be interpreted correctly together with its context, including stakeholder requirements, system context, and constraints.

Besides the highly prioritized requirements, there are many advanced requirements that have not drawn much attention yet. AKM-26 describes the reasoning capability of an AKM tool. It is an inference mechanism for automatic inconsistency detection or question creation. Integrating artificial intelligence into the tool leads to various benefits as well as challenges. On one hand, it lowers the cost to analyze a large number of design decisions. The analysis can also be performed more frequently to avoid critical conflicts in the end. On the other hand, it is challenging to codify human knowledge into machine rules. Developing a stand-alone prototype before integrating it into the whole system is suggested for the sake of tuning and testing.

The advanced requirements with a focus on ADD emphasize the analytic view of design decisions, such as ADD-12: completeness check, ADD-13: consistency check, and ADD-14: architectural decision consequence analysis. Currently, these tasks are mainly performed by architects instead of the tool because of their complexity. Similar to AKM-26, these requirements provide insight into the architecture of a software to create a complete and consistent architecture in the end. They should be involved in

the stand-alone prototype for analysis purposes.

Last but not least, in order to fulfill these requirements during the development, practitioners must be interviewed regularly for feedback whether the system provides the necessary functionality. As new requirements and technologies emerge, it will be challenging to adjust the existing system to incorporate them.

List of Figures

2.1 Microservice Architecture of AMELIE 10

4.1 Initial State of Projects 32
4.2 Redesigned Data Table of Projects 34
4.3 Initial State of Import 35
4.4 Redesigned and Reimplemented Import Stepper 37
4.5 Initial State of Quality Attributes 38
4.6 Initial State of Architectural Elements 39
4.7 Initial State of Expertise Matrix 40
4.8 Initial State of Expert Recommender 42
4.9 Redesigned Data Table of Expert Recommendations 43
4.10 Initial State of Design Decisions 44
4.11 Advanced Filtering Options for Design Decisions 45
4.12 Initial State of Decision Details 46
4.13 Redesigned Action Bar and Layout 48

5.1 Container Architecture vs. Virtual Machine Architecture 50

6.1 A Step-by-step Guided Tour on Projects Page 57
6.2 API Documentation for AKRE-Server 59

List of Tables

3.1 AKM Requirements Coverage 16
3.2 ADD Requirements Coverage 19
3.3 AKM Requirements Relevance 24
3.4 ADD Requirements Relevance 27

Bibliography

[1] M. A. Babar and I. Gorton. A tool for managing software architecture knowledge. In *Sharing and Reusing Architectural Knowledge-Architecture, Rationale, and Design Intent, 2007. SHARK/ADI'07: ICSE Workshops 2007. Second Workshop on*, pages 11–11. IEEE, 2007.

[2] M. A. Babar, I. Gorton, and B. Kitchenham. A framework for supporting architecture knowledge and rationale management. In *Rationale Management in Software Engineering*, pages 237–254. Springer, 2006.

[3] M. A. Babar, A. Northway, I. Gorton, P. Heuer, and T. Nguyen. Introducing tool support for managing architectural knowledge: an experience report. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 105–113. IEEE, 2008.

[4] M. A. Babar, X. Wang, and I. Gorton. Pakme: a tool for capturing and using architecture design knowledge. In *9th International Multitopic Conference, IEEE INMIC 2005*, pages 1–6. IEEE, 2005.

[5] M. Bhat, K. Shumaiev, A. Biesdorf, U. Hohenstein, M. Hassel, and F. Matthes. Meta-model based framework for architectural knowledge management. In *Proccedings of the 10th European Conference on Software Architecture Workshops*, page 12. ACM, 2016.

[6] M. Bhat, K. Shumaiev, and F. Matthes. Towards a framework for managing architectural design decisions. In *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, pages 48–51. ACM, 2017.

[7] J. Bosch. Software architecture: the next step. In *European Workshop on Software Architecture*, pages 194–199. Springer, 2004.

[8] G. Buchgeher and R. Weinreich. Automatic tracing of decisions to architecture and implementation. In *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*, pages 46–55. IEEE, 2011.

[9] R. Capilla, A. Jansen, A. Tang, P. Avgeriou, and M. A. Babar. 10 years of software architecture knowledge management: practice and future. *Journal of Systems and Software*, 116:191–205, 2016.

[10] R. Capilla, F. Nava, and C. Carrillo. Effort estimation in capturing architectural knowledge. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 208–217. IEEE Computer Society, 2008.

[11] R. Capilla, F. Nava, S. Pérez, and J. C. Dueñas. A web-based tool for managing architectural design decisions. *ACM SIGSOFT software engineering notes*, 31(5):4, 2006.

[12] E. J. Conklin and K. Yakemovic. A process-oriented approach to design rationale. *Human-Computer Interaction*, 6(3):357–391, 1991.

[13] J. Daiber, M. Jakob, C. Hokamp, and P. N. Mendes. Improving efficiency and accuracy in multilingual entity extraction. In *Proceedings of the 9th International Conference on Semantic Systems*, pages 121–124. ACM, 2013.

[14] R. C. De Boer, R. Farenhorst, P. Lago, H. Van Vliet, V. Clerc, and A. Jansen. Architectural knowledge: getting to the core. In *International Conference on the Quality of Software Architectures*, pages 197–214. Springer, 2007.

[15] R. Farenhorst, R. Izaks, P. Lago, and H. Van Vliet. A just-in-time architectural knowledge sharing portal. In *Software Architecture, 2008. WICSA 2008. Seventh Working IEEE/IFIP Conference on*, pages 125–134. IEEE, 2008.

[16] R. Farenhorst, P. Lago, and H. Van Vliet. Effective tool support for architectural knowledge sharing. In *European Conference on Software Architecture*, pages 123–138. Springer, 2007.

[17] M. Feilkas, D. Ratiu, and E. Jurgens. The loss of architectural knowledge during system evolution: an industrial case study. In *2009 IEEE 17th International Conference on Program Comprehension (ICPC 2009)*, pages 188–197. IEEE, 2009.

[18] G. Fischer and J. Otswald. Knowledge management: problems, promises, realities, and challenges. *IEEE Intelligent systems*, 16(1):60–72, 2001.

[19] A. Harley. Ux expert reviews. URL: `https://www.nngroup.com/articles/ux-expert-reviews/` (visited on 07/01/2018).

[20] P. Haumer, K. Pohl, K. Weidenhaupt, and M. Jarke. Improving reviews by extended traceability. In *Systems Sciences, 1999. HICSS-32. Proceedings of the 32nd Annual Hawaii International Conference on*, 10–pp. IEEE, 1999.

[21] D. Inc. What is a container. URL: `https://www.docker.com/what-container` (visited on 06/28/2018).

[22] W. Iso. 9241-11. ergonomic requirements for office work with visual display terminals (vdts). *The international organization for standardization*, 45:9, 1998.

[23] A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Software Architecture, 2005. WICSA 2005. 5th Working IEEE/IFIP Conference on*, pages 109–120. IEEE, 2005.

[24] A. Jansen, T. de Vries, P. Avgeriou, and M. van Veelen. Sharing the architectural knowledge of quantitative analysis. In *International Conference on the Quality of Software Architectures*, pages 220–234. Springer, 2008.

[25] A. Jansen, J. Van Der Ven, P. Avgeriou, and D. K. Hammer. Tool support for architectural decisions. In *Software Architecture, 2007. WICSA'07. The Working IEEE/IFIP Conference on*, pages 4–4. Ieee, 2007.

[26] P. G. Keen. Decision support systems: a research perspective. In *Decision Support Systems: Issues and Challenges: Proceedings of an International Task Force Meeting*, pages 23–44, 1980.

[27] P. Kruchten. An ontology of architectural design decisions in software intensive systems. In *2nd Groningen workshop on software variability*, pages 54–61. Citeseer, 2004.

[28] P. Kruchten, P. Lago, and H. Van Vliet. Building up and reasoning about architectural knowledge. In *International Conference on the Quality of Software Architectures*, pages 43–58. Springer, 2006.

[29] P. Lago, R. Farenhorst, P. Avgeriou, R. C. de Boer, V. Clerc, A. Jansen, and H. van Vliet. The griffin collaborative virtual community for architectural knowledge management. In *Collaborative Software Engineering*, pages 195–217. Springer, 2010.

[30] L. Lee and P. Kruchten. A tool to visualize architectural design decisions. In *International Conference on the Quality of Software Architectures*, pages 43–54. Springer, 2008.

[31] Z. Li, P. Liang, and P. Avgeriou. Application of knowledge-based approaches in software architecture: a systematic mapping study. *Information and Software Technology*, 55(5):777–794, 2013.

[32] P. Liang, A. Jansen, and P. Avgeriou. Collaborative software architecting through architectural knowledge sharing. *Collaborative Software Engineering (CoSE)*:343–368, 2008.

[33] P. Liang and P. Avgeriou. Tools and technologies for architecture knowledge management. In *Software Architecture knowledge management*, pages 91–111. Springer, 2009.

[34] A. MacLean, R. M. Young, V. M. Bellotti, and T. P. Moran. Questions, options, and criteria: elements of design space analysis. *Human–computer interaction*, 6(3-4):201–250, 1991.

[35] B. Ramesh and M. Jarke. Toward reference models for requirements traceability. *IEEE transactions on software engineering*, 27(1):58–93, 2001.

[36] W. C. Regli, X. Hu, M. Atwood, and W. Sun. A survey of design rationale systems: approaches, representation, capture and retrieval. *Engineering with computers*, 16(3-4):209–235, 2000.

[37] N. Schuster. Adkwik–a collaborative system for architectural decision modeling and decision process support based on web 2.0 technologies. *Stuttgart Media University*, 2007.

[38] M. Shahin, P. Liang, and M. R. Khayyambashi. A survey of architectural design decision models and tools. *Technical Report SBU-RUG-2009-SL-01, Sheikh Bahaei University & University of Groningen*, 2009.

[39] H. A. Simon. The sciences of the artificial. *Cambridge, MA*, 1969.

[40] M. Smith. The principles of graphic design: how to use proximity effectively. URL: `http://www.edgee.net/the-principles-of-graphic-design-how-to-use-proximity-effectively/` (visited on 07/01/2018).

[41] R. Spraque. A framework for the development of decision support system. *Management Information System Quarterly*, 4(4):1–26, 1980.

[42] A. Tang, P. Avgeriou, A. Jansen, R. Capilla, and M. A. Babar. A comparative study of architecture knowledge management tools. *Journal of Systems and Software*, 83(3):352–370, 2010.

[43] A. Tang, Y. Jin, and J. Han. A rationale-based architecture model for design traceability and reasoning. *Journal of Systems and Software*, 80(6):918–934, 2007.

[44] A. Tang, M. Razavian, B. Paech, and T.-M. Hesse. Human aspects in software architecture decision making: a literature review. In *Software Architecture (ICSA), 2017 IEEE International Conference on*, pages 107–116. IEEE, 2017.

[45] J. S. Van Der Ven, A. Jansen, P. Avgeriou, and D. K. Hammer. Using architectural decisions. In *Second International Conference on the Quality of Software Architecture (Qosa 2006)*, 2006.

[46] H. van Vliet and A. Tang. Decision making in software architecture. *Journal of Systems and Software*, 117:638–644, 2016.

[47] O. Zimmermann. Architectural decisions as reusable design assets. *IEEE software*, 28(1):64–69, 2011.