FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Software Architecture by Component Selection

Hugo Ari Rodrigues Drumond

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Filipe Alexandre Pais de Figueiredo Correia

Co-supervisor: Hugo José Sereno Lopes Ferreira

July 20, 2019

Software Architecture by Component Selection

Hugo Ari Rodrigues Drumond

Mestrado Integrado em Engenharia Informática e Computação

July 20, 2019

Abstract

Software architecture is concerned with the high-level modelling of a software system and comprises the decisions and the rationale that led to a particular architectural solution. Although the rationale behind decisions is at the core of well architectured software much of this knowledge is still implicit, impromptu and not supported by software engineering processes and tools. Resulting in increased costs for change and architecture degradation, and poor decision re-usability. Furthermore, Component Selection involves selecting existing components that are suitable for some parts of the system instead of developing the whole system from scratch allowing architects and developers to focus on their team's areas of expertise which is normally associated with better quality products and reduced time to market. Component Selection is still a hard task in part due to the nonexistence of structured knowledge in most software projects. This is because, component consumers seek the functional, and non-functional aspects that resulted from specifications and architectural decisions to select an appropriate component given problem context. In order words, capturing knowledge is equally important for consumers. However, building an ontology to encode it is non-trivial since different component types may have distinct features and rationales for selection. Taking this into consideration, the goal of this dissertation is to build a conceptual framework that helps with architectural decision making in particular the selection of components by making use of structured knowledge. To accomplish this we investigated techniques and frameworks related to Architecture Knowledge Management (AKM), component selection, component comparison, data formats, and artificial intelligence. Whose critical output is a set of issues that culminated in an approach and a list of desired characteristics, a desiderata, that implementations should heed. The approach involves collecting features segmented by concern from Software Components present in repositories and exposing them through a service. In a sense building a knowledge base of software features that can assist component selection. Consequently, we implemented a framework that captures structured knowledge from features files present in GitHub repositories exposing it through a Representational State Transfer (REST) API. To evaluate it we discuss the implementation of each key principle of the desiderata, contrast it with the literature review, and exemplify its use through a prototype front-end application populated with Big Data feature information. In conclusion, the analysis of the desiderata in the evaluation chapter indicates that our approach and implementation better assist feature comparisons in component selection processes when compared to current approaches, but it still needs to be put to the test by researchers willing to conduct users studies with developers of components and its consumers. All in all, we established: a framework that assists producers and consumers of software in capturing, searching and comparing software features; a structured approach to capture project knowledge stored along side code; and lastly, a set of key characteristics, an implementation and a client side prototype for the comparison of software.

Keywords: Software architecture, Component Selection, Knowledge-Base

Resumo

A arquitetura de software preocupa-se com a modelação de alto nível de um sistema de software e compreende as decisões e a lógica que levaram a uma solução de arquitetura específica. Embora a lógica por detrás das decisões esteja no centro de software bem arquitetado, muito deste conhecimento ainda é implícito, improvisado e não é suportado por ferramentas e processos de engenharia de software. Resultando num aumento dos custos de mudança e degradação de arquitetura, e má reutilização de decisão. Além disso, a seleção de componentes envolve a escolha de soluções existentes que são adequadas para algumas partes do sistema, em vez de se desenvolver tudo a partir do zero, permitindo que arquitetos e programadores se concentrem nas suas áreas de especialização. Prática esta que normalmente está associada a produtos de melhor qualidade com menor tempo de comercialização. A seleção de componentes ainda é uma tarefa difícil, em parte devido à inexistência de conhecimento estruturado na maioria dos projetos de software. Tal ocorre porque os consumidores de componentes buscam pelos aspetos funcionais e não funcionais resultantes de especificações e decisões arquiteturais para selecionar um componente apropriado, dado o contexto do problema. Noutras palavras, captar conhecimento é igualmente importante para os consumidores. No entanto, construir uma ontologia para codificá-la não é trivial, pois, diferentes componentes podem ter características e justificações distintas para a seleção. Tendo isto em conta, o objetivo desta dissertação é construir uma framework concetual que auxilie na tomada de decisões arquitetónicas, nomeadamente, a seleção de componentes utilizando conhecimento estruturado. Com este fim em vista, investigámos técnicas e estruturas relacionadas com Gestão de Conhecimento de Arquitetura, seleção de componentes, comparação de componentes, formatos de dados, e inteligência artificial. Cujo resultado crítico é um conjunto de problemas que culminou numa abordagem e numa lista de características desejadas, um desiderata, que as implementações devem seguir. A abordagem envolve a colheita de características segmentadas por domínio de componentes de software presentes em repositórios e na exposição deste através de um serviço. De certo modo, construindo uma base de conhecimentos que auxilia na seleção de componentes. Consequentemente, implementámos uma framework que captura o conhecimento estruturado de ficheiros de características presentes em repositórios do GitHub, expondo-os numa API REST. Para avaliá-la, discutimos a implementação de cada princípio-chave do disederata, contrastamos com a revisão da literatura, e exemplificamos o seu uso através de um protótipo preenchido com informações sobre Big Data. Em conclusão, a análise do desiderata indica que a nossa abordagem e implementação melhoram as comparações de características nos processos de seleção de componentes quando comparadas com as abordagens atuais. No entanto, estas aindam precisam de ser testadas por investigadores dispostos a conduzir estudos com produtores de componentes e consumidores. De um modo geral, estabelecemos: uma estrutura que auxilia produtores e consumidores de software na captura, pesquisa e comparação de características de software; uma abordagem estruturada para capturar conhecimento de projetos armazenado ao lado do código; e, por último, um conjunto de características chave, uma implementação e um protótipo para a comparação de software.

Acknowledgements

First and foremost, I would like to thank my supervisor, Filipe Alexandre Pais de Figueiredo Correia, whose guidance and availability was decisive in the elaboration of this document. Secondly, I thank my co-supervisor, Hugo José Sereno Lopes Ferreira, for providing input when there were some early project deadlocks.

Last but not least, my sincere gratitude to my family for their support and unconditional love which has given me the strength to keep moving forward for as long as I remember. Also a big thank you to my friends for the many laughs and adventures we shared along the way.

Hugo Ari Rodrigues Drumond

"Whether you think you can,
or you think you can't – you're right."

Henry Ford

Contents

List of Figures

List of Tables

Acronyms

ADD Architecture Design Decision. 9, 14, 15, 16

ADDSS Architecture Design Decision Support System. 11

ADL Architecture Description language. xiii, 11, 12, 13, 16, 17

ADvISE Architectural Design Decision Support Framework. xiii, xiv, 13, 14, 15, 95

AKM Architecture Knowledge Management. i, 1, 10, 11, 16, 17, 19, 42

AKMS Architecture Knowledge Management System. xiii, xiv, 1, 10, 12, 13, 15, 16, 17, 32, 42, 71, 99, 101

API Application Programming Interface. 45, 47, 52, 53, 56, 57, 63, 64, 66, 67, 68

ArchiMate Architecture-Animate. 11

AREL Architecture Rationale and Elements Linkage. 11

ATAM Tradeoff Analysis Method. 15

CBAM Cost Benefit Analysis Method. 15

CoCoADvISE Reusable Architectural Decision Models for Quality-driven Decision Support. xiii, xiv, 15, 16, 70, 97

COTS Commercial off-the-shelf. 21

DAMSAK Data Model for Software Architecture Knowledge. 11

DoDAF Department of Defense Architecture Framework. 11

DRIM Design Recommendation and Intent Model. 5

DRL Decision Representation Language. 5, 9

FDL fuzzy description logic. 21

HTML Hyper Text Markup Language. 16, 28, 29, 32, 43

HTTP Hyper Text Transfer Protocol. 23, 28, 29, 30

IBIS Issue Based Information Systems. 5, 8, 9

IRI Internationalized Resource Identifier. 23, 26

ISAA Integrated Issue, Solution, Artifact and Argument model. 5, 32

JSON JavaScript Object Notation. 32, 33, 34, 47, 49, 50, 51, 54, 55, 56, 64, 65

LADR Lightweight Architecture Decision Records. 17

MCDM Multiple-criteria decision-making. 1, 17, 21, 22, 57

NFR non-functional requirement. 10, 11, 13, 20, 21

OOP Object-oriented programming. 25

OSS Open-source software. 16, 21

OWL Web Ontology Language. 27, 31

PAKME Process-based Architecture Knowledge Management Environment. 11

PHI Procedural Hierarchy of Issues. 5

QOC Questions, Options, and Criteria. xiii, 5, 9, 13, 14, 15, 70

QuaDAI Quality-driven Product Architecture Derivation and Improvement. 15

RDF Resource Description Framework. xiii, 23, 24, 25, 26, 27, 28, 29, 30, 31, 33, 42, 43, 71

RDFa Resource Description Framework in Attributes. 32, 43, 71

RDFS Resource Description Framework Schema. xiii, 25, 27, 31

REST Representational State Transfer. i, 47, 56, 57, 66, 68

SPARQL SPARQL Protocol and RDF Query Language. 26, 27, 28

SysML Systems Modeling Language. 11

TOGAF The Open Group Architecture Framework. 11

UML Unified Modeling Language. xiii, 8, 9, 11, 16

URI Uniform Resource Identifier. 23, 28, 29, 30, 31, 42, 55

URL Uniform Resource Locator. 23, 46, 48, 50, 52, 55, 56, 64, 65

VbMF View-based Modeling Framework. xiii, 14, 15

XML eXtensible Markup Language. 23, 32, 33, 34

YAML YAML Ain't Markup Language. 33, 49, 51, 54, 55, 65

Glossary

MediaWiki Is a free open source wiki. It can be found at https://www.mediawiki.org/wiki/MediaWiki. xix

QuABaseBD pronounced as 'kbase-BeeDee' is "a Knowledge Base for Big Data Architectures and Technologies" that follows the taxonomy of (Gorton et al., 2015). It can be found at https://quabase.sei.cmu.edu/mediawiki/index.php/Main_Page. xiii, xv, 18, 19, 20, 42, 44, 61, 62, 64, 65, 66, 81, 82, 83, 84, 85, 86, 87, 88

Semantic MediaWiki Is an free open source extension to MediaWiki which adds concepts from the semantic web in order to give information structure. Effectively building knowledge management systems whose information can be queried, shared and linked in a machine and human readable way. It can be found at https://www.semantic-mediawiki.org/wiki/Semantic_MediaWiki. 19, 32, 64, 65

Chapter 1

Introduction

In this chapter we describe the motivations that lead to this thesis, and sum up the overall structure of the document. Section 1.1 presents related topics. Section 1.2 defines the problem and its causes. Section 1.3 describes the objective of the project as well as the motivation behind it. Section 1.4 elaborates on the main contributions of our work. Section 1.5 presents the structure and content of the document.

1.1 Context

Software architecture is concerned with the high-level modelling of a software system and comprises the decisions and the rationale that led to a particular architectural solution (Bosch, 2004; Jansen and Bosch, 2005). Although the rationale behind decisions is at the core of well architectured software much of this knowledge is still implicit, impromptu and not supported by software engineering processes and tools (Jansen and Bosch, 2005; Capilla et al., 2016). Resulting in increased costs for change and architecture degradation, and poor decision re-usability. As a way to solve this predicament, a new research interest emerged, AKM. Its tools assist in the decision making process by gathering and managing information paramount to the inception and evolution of a robust and backed architecture. Unfortunately AKM adoption is still not widespread mainly due to the costs of capturing the required information necessary to aid decision (Capilla et al., 2016). Furthermore most AKMSs: do not integrate well with current development processes; do not provide guidelines about which architecture knowledge is indeed important; and do not integrate well or at all with current tools and processes so as to not duplicate information. Because AKMSs are concerned with documenting decisions and their rationale they are linked to design rationale. It seeks argumentation-based structures to record decisions and their reasons as a way to address wicked problems, that is, difficult problems with incomplete, contradictory, and changing requirements.

1.2 Problem Definition

Component Selection involves contrasting a set of candidate components according to a set of criteria and picking one following a criteria input. It is often associated with Multiple-criteria decision-making (MCDM) systems and algorithms because it deals with the modelling of multiple conflicting criteria in decision making which is common in software selection — e.g. evaluating component origins, technology selection, etc. One of the reasons for the difficulty in Component Selection is the nonexistence of structured knowledge in most software projects. This is because architectural decisions are normally justified by combining problem context (data characteristics, granularity, ecosystem restrictions, etc) and non-functional vocabulary which is also an important factor in the selection of components. So it seems plausible to say that architecture knowledge capture is equally important for producers and consumers of software components. However, a wide encompassing and complex approach to capturing architecture knowledge might not be the most adequate since software developers would have to shift considerable development time to documentation activities which would also not benefit component consumers as they only need the current architectural decisions snapshot and not the full history that lead to it. As a result for the purpose of component selection the encoding of only the current characteristics and trade-offs of components is probably the best approach. Such would assist the construction of candidate component descriptions, Component Catalogues. Furthermore, components from different types have distinct features which makes efforts to build these taxonomies a hard problem. One additional concern is how to achieve and maintain a common model that is able to describe all software components while also describing trade-offs. The lack of specialised search engines and models for component selection means the search for software features must rely on generic keyword-based search engines which do not take advantage of model semantics. Indeed, it is current practice to use Web search engines, like Google, and keywords such as *versus* to find software comparisons in blogs, forums, developer communities, and so on. Fortunately, there are some sources of structured knowledge scattered all over the web such as the: Open Source Time Series DB Comparison[1], Knowledge Base of Relational and NoSQL Database Management Systems[2], Ultimate Time Series DB Comparison[3], Relational Database Management Systems Comparison[4], etc. However, in both cases information is hard to search and contrast because they hold different syntaxes and semantics. Moreover there is seldom a description of the architectural approaches taken and the rationale behind them which would indicate the scenarios and quality attributes that the system was designed for.

[1] https://docs.google.com/spreadsheets/d/1sMQe9oOKhMhIVw9WmuCEWdPtAoccJ4a-IuZv4fXDHxM/edit

[2] https://db-engines.com/en/systems

[3] https://tsdbbench.github.io/Ultimate-TSDB-Comparison/

[4] https://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems

1.3 Motivation and Goals

Using existing components that are suitable for some parts of the system instead of developing the whole system from scratch allows architects and developers to focus on their team's areas of expertise which is normally associated with better quality products and reduced time to market. However the process of selection is most often than not a laborious task since it is done in an ad-hoc way relying on individual experience, manual search, and only then comparison of features. Hence, the goal of this work is to provide an approach and develop a tool that helps with the selection of components by making use of structured knowledge. This would make the documentation of software features a byproduct of documented projects and not an external concern. In the long run such a formalism would also benefit architecture refactorings by providing a formal way to compare components with similar intents — e.g. the comparison between postgres and mongodb. In a distant future, the ultimate vision would be for an intelligent system to combine this information with load parameters and migration rules to change a component for another in a running environment while maintaining the functional behaviour of the system. An Architecture Refactoring Suggestion for component change could be triggered by determining under-performing components through the analyses of different metrics in a live environment.

1.4 Contributions

The main contribution of our work is a new way to capture structured knowledge in a way that fosters contributions and reuse of that information by other services. As a result the contributions of this dissertation are the following:

- A conceptual framework that assists producers and consumers of software in capturing the features of their software per domain, searching for appropriate components, and comparing them. As a side effect this would serve as a stepping stone towards Architecture Refactorings.

- A structured approach to project knowledge capture stored along side code that could possibly be extended to scenarios other than component selection — e.g. mapping of features to code through annotations which could be useful for building a dataset for machine learning purposes, whose aim would be to do the inverse, mapping code into features.

- A set of key characteristics described in a desiderata, an implementation of those principles in a service and a client side prototype for the comparison of software.

1.5 Document Structure

- Chapter 2 describes approaches to component selection, architecture knowledge management and other related works while also relating them to the problem at hand.

- Chapter 3 summarises current issues and presents our proposal.
- Chapter 4 describes the overall architecture of the system, the techniques and tools that were used to create the solution, its features, and how we can use and run the application.
- Chapter 5 evaluates whether the implementation follows the desiderata.
- Chapter 6 enumerates the main difficulties faced, presents the main contributions, indicates what future work could look like, and draws conclusions.

Chapter 2

Literature Review

In this chapter we present and discuss topics related to the goal we are trying to accomplish, to develop an approach and a tool that helps with the selection of components by making use of structured knowledge. Section 2.1, describes how architecture evolved in relation to the encoding of decisions and rationale in order to understand how these models can be used to aid component selection. Section 2.2 presents different ways to frame component selection problems. And Section 2.3 details possible data formats for the encoding of knowledge. Section 2.4 showcases different comparison websites. Building a tool that assists component selection by making use of structured information involves encoding vocabulary and its relationships, and sharing and reasoning about this body of knowledge. Henceforth different approaches to solve the above are presented.

2.1 Architecture Knowledge Management

Software architecture is complex, prone to erosion, and with high costs for change (Jansen and Bosch, 2005). According to (Jansen and Bosch, 2005) this is partly due to design decision knowledge about architecture being implicit and not having a first-class representation. The lack of explicitness results in valuable knowledge being lost especially when experts depart raising overall project cost (Capilla et al., 2016). The first systems that tried to handle problems related to decision encoding where built around the 70s and the 80s and used concepts from Design Rational (Capilla et al., 2016). In a general sense rationale is:

> "The reasons or intentions that cause a particular set of beliefs or actions"
> *Cambridge Dictionary*

Hence Design Rationale is concerned with the explicit encoding of decisions, and the reasons behind those decisions that constitute the design of an artefact (Jarczyk et al., 1992). Its approaches, also called argumentation-based models, are studied and used in many fields because its

goal is ubiquitous; to understand, maintain, and improve design by building a collection of structured knowledge about decisions and its reasons. Some of which are the: Toulmin model (Toulmin, 1958); Issue Based Information Systems (IBIS) (W Kunz, 1970); Procedural Hierarchy of Issues (PHI) (McCall, 1991); QOC (MacLean et al., 1991); Decision Representation Language (DRL) (Lee, 1991); Design Recommendation and Intent Model (DRIM) (Pena-Mora et al., 1993); Win-Win Spiral Model (Boehm and Kitapci, 2006); and more recently the Integrated Issue, Solution, Artifact and Argument model (ISAA) (Zhang et al., 2013). The Software Engineering Community has adapted and extended some of these methods to document the rationale behind software design decisions, requirements specifications, and other approaches that combine rationale and scenarios to elicit and refine requirements (Tang et al., 2006). As can be seen in Figure F.1, design rationale can be useful to project managers, members of the development team, maintainers, analysts, new stakeholders, and current and future architects. In our opinion, designers of other systems, what we call component consumers, too find system rationale and constraints particularly important because they can leverage this information to better determine how well a component might fit into their system — e.g. VoltDB uses statement-based replication for their implementation of replication logs which is prone to inconsistency when faced with non-deterministic operations; however it does not constitute a problem since the system was designed uniquely for deterministic transactions (no functions such as NOW() that get dispatched to replicas in DML statements) (Kleppmann, 2017).

According to (Tang et al., 2006) early references to the importance of using a Design Rationale approach to Software Engineering can be found in (Parnas and Clements, 1985) and (Potts and Bruns, 1988), however it was only with the work from (Perry and Wolf, 1992) that a foundation was established for the evolving Software Architecture Community. It states that software architecture is a set of processing, data, and connecting elements holding constraints and relationships among each other that follow from a careful reasoning process that ought to be a first-class citizen in architectural descriptions (Perry and Wolf, 1992).

$$SoftwareArchitecture = \{Elements, Form, Rationale\}$$

Furthermore, they crystallise and relate architectural concepts and terms as a way to build an unifying basis for understanding and sharing. Some of these are: architectural views; architectural styles; requirements; architecture; design; implementation; architecture specifications; problems of use and reuse; etc.

Some important works in the beginning of the 2000s stated the importance of design rationale (Len Bass, Paul Clements, 2003) and the limitations of current approaches (Bosch, 2004), and made efforts to build new ontologies (Kruchten, 2004) and tools (Jansen and Bosch, 2005). Until then efforts to guide the management and use of rationale information such as the IEEE 1471-2000 standard (IEEE Architecture Working Group, 2000) and the Views and Beyond approach (Clements et al., 2002) had some flaws (Tang et al., 2006). The first made architectural terms explicit, stated what information architectural descriptions should contain, and conceived

a conceptual model of architectural description where relations to defined terms were made. Although they state that rationale is a part of architectural descriptions, no information for the storing, sharing, or manipulation of this knowledge was shown. The later emphasises the importance for rationale capturing, clarifies what rationale information is, and states a way to document rationale through the linkage of factors to issues to design decisions called Global Analysis, see Figure 2.1. However the previous topics are only touched very briefly and the document does not demonstrate a concrete example of a rationale toolkit for Software Architecture and a process that makes use of it, show alternative ways to document and use rationale, or indicate how this knowledge could be shared between communities.

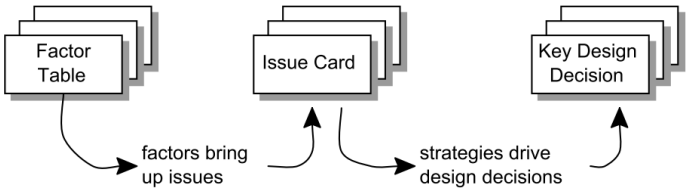

Figure 2.1: Global Analysis Flow
Source: (Clements et al., 2002)

Global analysis is made up of two parts, analysis and rationale. It is an architectural design activity done incrementally as the system's architecture evolves due to new factors and the refinement of strategies into new issues (Clements et al., 2002; Hofmeister et al., 2005), and is used in the Siemens approach to architecture design as the first design activity to back each view (Clements et al., 2002). First, all characteristics that affect the architecture of the system are taken into account by analysing the environment and encoding them into factors. These represent: environment constraints such as time to market and resources; system requirements such as important features, quality attributes and technological constraints; and their flexibility, changeability and impact (Clements et al., 2002). The Table 2.2 shows an example of a factor table.

Factor	Flexibility/Changeability	Impact
O4.2 Schedule Feature Delivery		
Features are prioritised	Negotiable	Moderate impact on the schedule
T2.1 Domain-specific Hardware Probe Hardware		
Hardware to detect and process signals	Upgraded every 3 years as technology improves	Large impact on image acquisition and processing components
P1.1 Features Acquisition Types		
Acquire raw signal data and convert into images	New types of acquisitions may be added every 3 years	Affects UI, acquisition performance, and image processing

Figure 2.2: Example of a Factor Table
Source: (Hofmeister et al., 2005)

The major architectural design characteristics of a system stem from factors that have low flexibility, high changeability, or that affect many components (Clements et al., 2002). This leads

us to the second concept, Issue Cards. Their purpose is to assist architects in identifying key problems which come about from strong factors, and in identifying a solution in the form of strategies and related approaches to solve the problem (Hofmeister et al., 2005), see Figure 2.3.

Issue: Easy Addition and Removal of Acquisition Procedures

There are many acquisition procedures. Implementation of each feature is quite complex and time consuming. There is a need to reduce complexity and effort in implementing such features.

Influencing Factors
O4.1: Time to market is short.
O4.2: Delivery of features is negotiable.
P1.1: New acquisition procedures can be added every three years.
P1.2: New image-processing algorithms can be added on a regular basis.
...

Solution
Define domain-specific abstractions to facilitate the task of implementing acquisition and processing applications.
Strategy: Use a flexible pipeline model for image processing.
Develop a flexible pipeline model for implementing image processing. Use processing components as stages in the pipeline. This allows the ability to introduce new acquisition procedures quickly by constructing pipelines using both old and new components.
Strategy: Introduce components for acquisition and image processing.
...
Strategy: Encapsulate domain-specific image data.
...

Related Strategies
See also **Encapsulate domain-specific hardware.**

Figure 2.3: Example of an Issue Card
Source: (Hofmeister et al., 2005)

Issue cards were inspired by design patterns and pattern languages (Hofmeister et al., 2005) and in an analogous way: list what factors affect it and if needed how so; discusses a general solution and identifies a set of strategies to solve the problem which can be in the form of architectural styles, architectural patterns, design tactics, design guidelines, constraints, or other approaches (Clements et al., 2002) along with a explanation about its impact on the system — e.g. strengths and weaknesses in the form of quality attributes; and, points to alternative solutions.

This process culminates into key design decisions which represent the actual implementation of strategies from Issue Cards. In essence, it is a table which maps rationales in the form of strategies, present in Issue cards, to design decisions see Figure 2.4.

Design Decision	Rationale
Decompose the Exporting component into ImageCollection, Comm, and Export components. ImageCollection and Comm handle the domain-specific image data	Strategy: Encapsulate domain-specific image data

Figure 2.4: Example of a Decision Table
Source: (Hofmeister et al., 2005)

In this way, Global Analysis provides rationale capture, linkage between architecture design requirements and strategies, and traceability through the the linkage of factors to issues to design decisions (Clements et al., 2002). To conclude, it adapts concepts from design rationale namely IBIS (W Kunz, 1970) to aid architectural processes through the use of issues, positions, and arguments present in Issue Cards in the form of issues, strategies and rationale (Hofmeister et al.,

2005). There seems to be shortcomings in Global Analysis such as the lack of trade-off, composability and dependency descriptions for decisions and strategies when defining a solution in an Issue Card. Which possibly arises from the fact that the model lacks first-class structures to describe this knowledge. Nevertheless, it could prove useful for our work as a means to define a collection of stories written by component consumers with semi-structured text associated to a particular component — e.g. a collection of justifications for the use a datastore. That information could live alongside software code acting as documentation for component consumers, and be consumed by a service which presents use case stories for particular software; thereby assisting consumers searching for appropriate software. To take advantage of such a structure the UML model below, Figure 2.5, would have to be extended and implemented in a backend which would receive stories from repositories or through a web app.

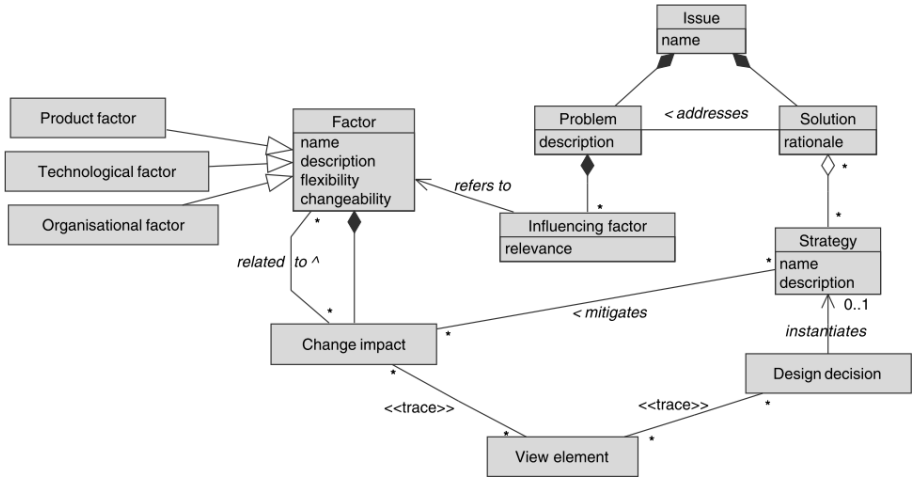

Figure 2.5: UML model of Global Analysis artefacts
Source: (Hofmeister et al., 2005)

There is a problem however, methods such as IBIS (W Kunz, 1970), QOC (MacLean et al., 1991), and DRL (Lee, 1991) are not widely accepted by practitioners because they need to know the end results of the design and the related intents and rationale to get there (Capilla et al., 2016). Also there is the danger of separating architectural models from their rationale which inevitably leads to heaps of textual information which are difficult to use (Conklin, 1991).

To deal with the lack of first class representation for Architecture Design Decisions (ADDs), and their cross-cutting and intertwining — i.e. affecting multiple components and connectors — description of software architecture Bosch suggested the creation of decision-centric Architecture Knowledge systems (Bosch, 2004). In it he states that architecture research on modelling of components and connectors, the first stage of software architecture, has matured and disseminated to the industry (Bosch, 2004). And that further research on first-class representation of design decisions, the second stage of software architecture, is needed to deal with design erosion and the difficulty in changing the architecture of software systems (Bosch, 2004). Their evidence suggests that knowledge vaporisation of information concerning domain analysis, architectural

styles, selected patterns, and other design decisions is the main culprit in architectural unsoundness (Bosch, 2004). In sum, he found the following problems and promoted further work on the: first-class presentation of ADDs; cross-cutting and intertwined nature of design decisions among components, connectors and even themselves; high cost of change because of the above; design rules and constraints violation; and, removal of obsolete design decisions (Bosch, 2004). They also identified the many parts that constitute design decisions in a rigorous manner:

- Restructuring effect: design decisions have great impact on software architecture because they involve addition, removal, splitting or merging of components (Bosch, 2004). The accumulated restructuring to system design due to a plethora of decisions is not easy to understand, thus a notation and language to describe design decisions is paramount (Bosch, 2004).

- Design rules: design decisions can impose rules that some or all components must follow such as a particular way of performing a task (Bosch, 2004).

- Design constraints: define what the system and its parts may not do (Bosch, 2004).

- Rationale: is the output of a careful analysis into the functional and non-functional requirements (NFRs) to achieve the best design (Bosch, 2004).

From this moment on several AKMSs and models were built with support for one or more of the following main use cases for Architecture Knowledge (De Boer et al., 2007; Capilla et al., 2016): **sharing** of goals, requirements, problems, system behaviour, contexts such as assumptions, constraints, risks, trade-offs, and so on (Capilla et al., 2016); **compliance** to set constraints, perform dependency, consistency, impact and quality requirements analysis as the architecture changes during its lifetime (Capilla et al., 2016); **discovery** of design questions, design alternatives, behaviours and scenarios through the knowledge encoded in the system in the form of business and technical contexts, architecture views, and so on (Capilla et al., 2016); and **traceability**, the ability to navigate forwards from requirements, decisions, and implementation or backwards to assist in system understanding, impact analysis, design assessment, and designer maintenance tasks such as reviewing architecture changes (Capilla et al., 2016).

Although there are many AKMSs there is still no consensus on: a common meta-model / data-model to describe Architecture Knowledge (Capilla et al., 2016); and, what type of knowledge is valuable to which purpose (Capilla et al., 2016). As a consequence Component Selection assisted by Architecture Knowledge is still a non-trivial problem with no established solution.

The first step in order to build a tool for AKM is to define a data-model. ISO/IEC/IEEE 42010 (ISO/IEC/IEEE, 2011), described in summary in appendix B, is the successor of the IEEE 1471-2000 standard (IEEE Architecture Working Group, 2000) and its purpose is to standardise terminology and models to describe how architecture descriptions of systems are organised and expressed (ISO/IEC/IEEE, 2011).

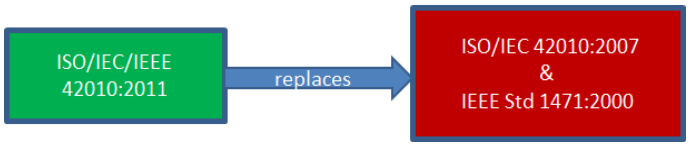

Figure 2.6: New Architecture Description Standard ISO/IEC/IEEE 42010
Source: http://enterprise-strategy-architecture.blogspot.com/2011/11/understanding-isoiecieee-420102011.html

Over the years there have been other attempts such as Process-based Architecture Knowledge Management Environment (PAKME) (Babar et al., 2005), Data Model for Software Architecture Knowledge (DAMSAK) (Babar et al., 2006), the meta-model behind Architecture Design Decision Support System (ADDSS) (Capilla et al., 2006), Architecture Rationale and Elements Linkage (AREL) (Tang et al., 2007), and the various models underpinning Knowledge Architect (Jansen et al., 2009) (Capilla et al., 2016). As there is too much diversity and no uniform approach to cover all architecture knowledge use cases it is best (and advised) to develop custom solutions following existing meta-models more adapted to those cases (Capilla et al., 2016). Afterwards using those implementations to validate empirically the underlying meta-model (Capilla et al., 2016). Recently researchers have used and adapted the IEEE 42010:2011 meta-model (ISO/IEC/IEEE, 2011) to develop new AKM Tools or to evaluate previous ones (Capilla et al., 2016). Our solution could make use of the IEEE 42010:2011 meta-model (ISO/IEC/IEEE, 2011) to create a simplified tool adapted to Component Selection. There are also some architecture frameworks and ADLs that follow this standard and that could possibly be looked into and adapted. An architecture framework is a set common of practices within a domain for building, analysing and using architecture descriptions, some examples of IEEE 42010:2011 (ISO/IEC/IEEE, 2011) architecture frameworks are: Kruchten's "4+1" view model (Kruchten, 1995), Siemens' 4 views method (Christine Hofmeister, Robert Nord, 1999), Department of Defense Architecture Framework (DoDAF)[1], The Open Group Architecture Framework (TOGAF)[2], among others[3]. An ADL is a graphical or / and textual description of a software system in terms of elements and theirs relationships[4] and can be of three types: box and line informal drawings, formal ADL, and UML-based notation (Malavolta et al., 2013); some examples of ones compatible with the IEEE 42010:2011 meta-model (ISO/IEC/IEEE, 2011) are: Rapide[5], Wright[6], Systems Modeling Language (SysML)[7], Architecture-Animate (ArchiMate)[8], etc[3] (ISO/IEC/IEEE, 2011).

An ADL could be implemented or used to describe the interactions of the elements that constitute a system to favour component selection. Through the analysis of the structure and relations

[1] https://dodcio.defense.gov/library/dod-architecture-framework/
[2] https://en.wikipedia.org/wiki/The_Open_Group_Architecture_Framework
[3] http://www.iso-architecture.org/ieee-1471/afs/frameworks-table.html
[4] https://www.todaysoftmag.com/article/2241/architecture-description-languages
[5] http://complexevents.com/stanford/rapide/
[6] http://www.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/rallen_thesis.htm
[7] https://www.omg.org/spec/SysML/About-SysML/
[8] http://pubs.opengroup.org/architecture/archimate3-doc/

described in the ADL artefact behaviours and NFRs could possibly be inferred and checked. Also Architectural Styles, Patterns and Tactics could be a first class citizen in the language and act as an abstract concept (abstract class) that could be realised (implemented) into the artefact to form well understood relationships between elements. The combination of the above would allow a component consumer to understand the intricacies of the system under analysis for selection while at the same time having a textual and / or graphical representation which could be composed with their own ADL artefact to determine the qualities of the composition — e.g. like an import of a library in a programming language where well defined interfaces are exposed and intricacies hidden. The ADL could possibly be annexed or linked to structured documentation to form a simple source of Architecture Knowledge which could live alongside code in a repository. However, formal ADLs are rarely part of software life cycles due to the lack of documentation, tooling, extensibility, and wide focus (Malavolta et al., 2013). Because of this UML is seen as the successor to existing ADLs (Malavolta et al., 2013), see Figure 2.7. As a matter of fact UML has gotten closer to an ADL, "ADL research of the 1990s directly influenced the definition of UML 2.0" (Malavolta et al., 2013). Furthermore there are many UML extensions tailored towards specific concerns in software engineering[9] and others[10]. Unfortunately there are still analysis limitations in ADLs especially in terms of extra-functional expressiveness (Malavolta et al., 2013). As a consequence it is probably ill advised to extend or build an ADL for component selection within our time-frame.

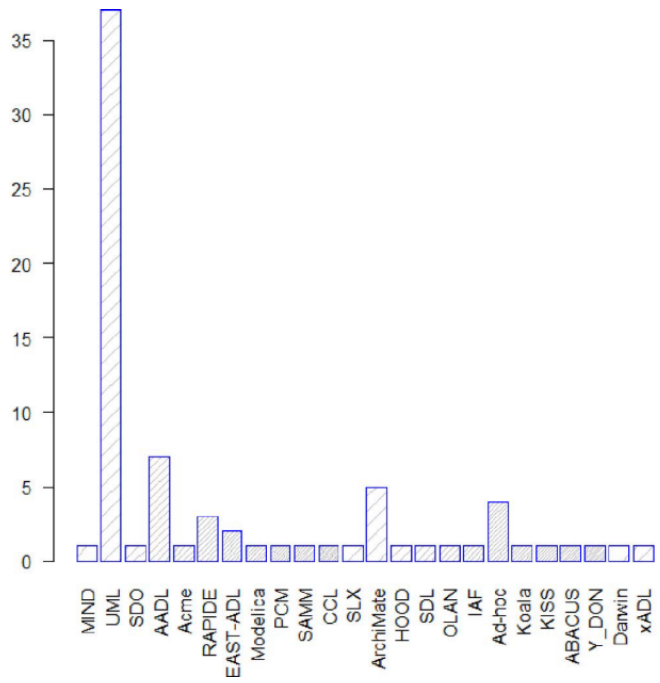

Figure 2.7: ADLs used by the study population
Source: (Malavolta et al., 2013)

[9] https://www.omg.org/spec/category/software-engineering/
[10] https://www.omg.org/spec/

AKMSs built on top of these approaches to make the management of decisions and rationale a first class citizen. There are three generations of AKMSs (Capilla et al., 2016) as can be seen in Figure 2.8.

Tool	CAP	MGM	SHA	DOC	EVO	REU	REA	UCT	COL	PER	ASS
1st Generation											
RAT	X	P	--	P	--	--	X	--	X		X
Archium	X	P	--	X	--	--	P	--	--	--	--
PAKME	X	X	P	X	X	--	--	--	P	--	--
ADDSS	X	X	P	X	X	--	--	--	P	--	--
AREL	X	P	--	X	X	--	--	--	X	--	P
2nd Generation											
Eagle	X	X	X	X	P	--	--	--	X	X	--
ADkwik	X	X	X	X	P	X	--	--	X	--	--
SEURAT	X	P	--	P	--	--	P	--	--	--	X
KA	X	X	--	X	--	--	--	--	X	P	P
ADDM	X	X	X	X	P	X	--	--	X	X	--
ADDSS 2.0/2.1	X	X	P	X	X	P	--	--	P	P	--
3rd Generation											
SAW	X	X	X	X	--	--	P	X	X	--	P
ADvISE	X	X	X	X	--	X	P	--	X	--	X
Decision Architect	X	X	X	X	X	--	--	--	X	P	P
RGT	X	P	X	X	X	--	--	--	X	--	X

Legend: *CAP (Capture), MGM(Management), SHR(Share), DOC(Document), EVO(Evolution), REU(Reuse), REA(Reasoning), UCT(Uncertainty), COL(Collaborative), PER(Personalization), ASS(Assessment), X (Capability supported), P(Capability partially supported), --(Capability not supported)*

Figure 2.8: Comparison between AKMSs
Source: (Capilla et al., 2016)

Work on the tools from the first generation (2004-2006) was done with little knowledge of each other and focused on the ways to capture and represent problem knowledge — e.g. using templates list of attributes like (Tyree and Akerman, 2005) which can be seen as an extension of Global Analysis — (Capilla et al., 2016). The second generation (2007-2010) brought sharing capabilities by using wikis and web based tools and personalization to extend features to specialised groups of users (Capilla et al., 2016). And the third generation (2011-2014), focused on collaboration for concurrent work, reuse which often consists in the retrieval of captured design decisions or design patterns, fuzzy decision-making, and assessment capabilities (Capilla et al., 2016).

The features that are most important for component selection apart from basic functionality like capturing (CAP) and management (MGM) of knowledge, are reuse (REU), some kind of reasoning (REA) to guide the user towards the right component, sharing (SHA), relation to NFRs (ASS), and if possible collaborative (COL) mechanisms to promote contributions.

From the Figure 2.8 and the Tables E.1 from appendix E, we observe that the only tool that supports those requirements is ADvISE[11] which is a Eclipse-based tool that supports modelling of ADLs through the use of QOC (MacLean et al., 1991) and fuzzy decision making (Lytra et al., 2013). Its purpose is to assist decision making for reusable architecture decisions at different levels of abstraction to achieve low cost documentation of rationale in a semi-automated fashion[11], see Figure 2.9.

[11] https://swa.univie.ac.at/Software_Architecture/research-projects/architectural-design-decision-support-framework-advise/

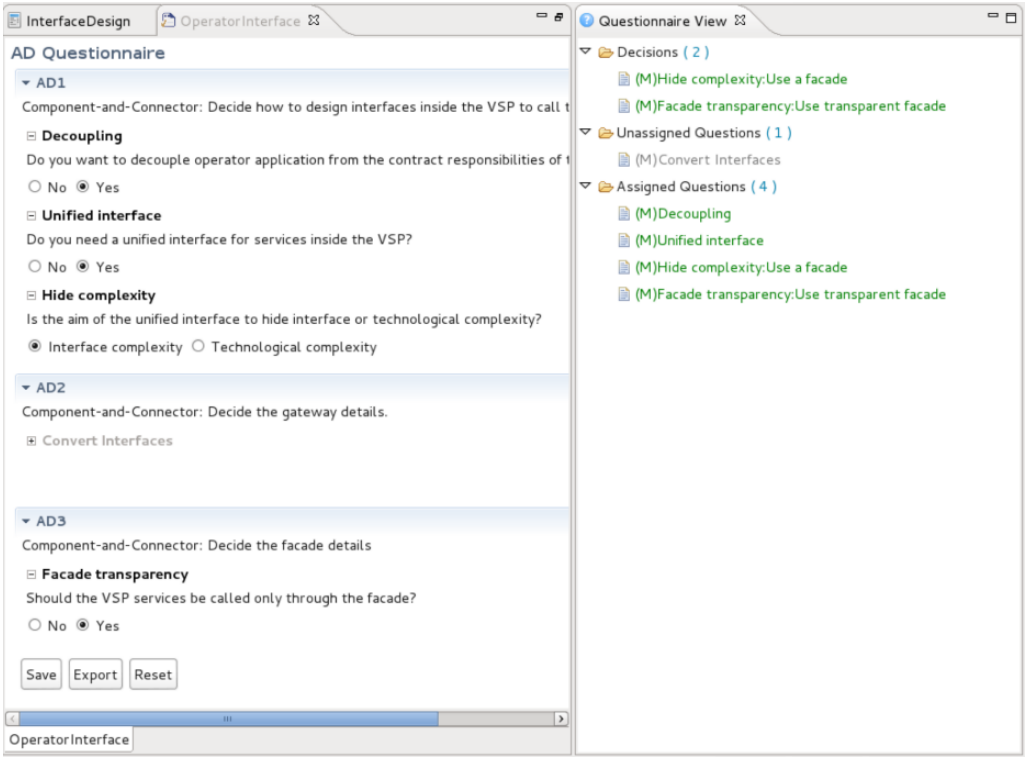

Figure 2.9: ADvISE
Source: (Lytra et al., 2013)

In appendix C Figure C.1 which describes the ADvISE meta-model we can see that a decision is a collection of questions which have options or answers that may trigger further questions and decisions; and that options establish links to solutions which are a collection of patterns. Also, the model can be reasoned about by writing forces and rules in the fuzzy logic layer. It is effectively an extension of QOC (MacLean et al., 1991), seen in Figure 2.10, with support for inference.

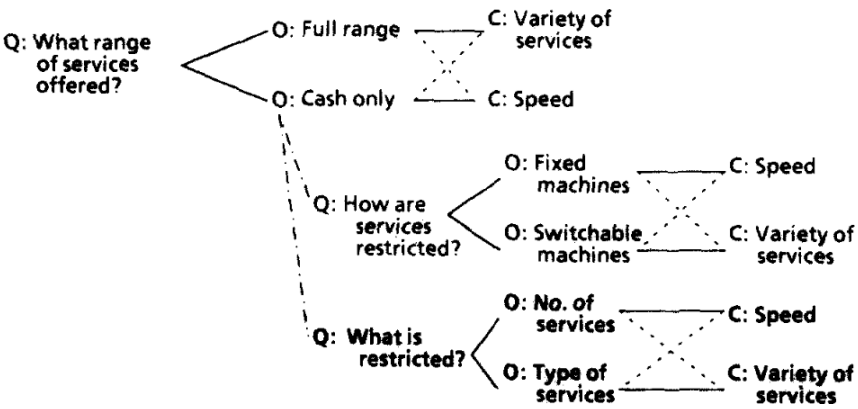

Figure 2.10: QOC example
Source: (MacLean et al., 1991)

In essence, ADvISE allows for the creation of once per domain reusable ADD using the *Model Editor* to generate Questionnaires editable through the *Questionnaire Editor Tool* as a way to make concrete ADDs (Lytra et al., 2013); thereby automatically generating architectural decision documentation from the answers to the questionnaires (Lytra et al., 2013). It can also be integrated with VbMF to keep architectural decisions and designs consistent and traceable to each other[11] (Lytra et al., 2013), see Figure 2.11.

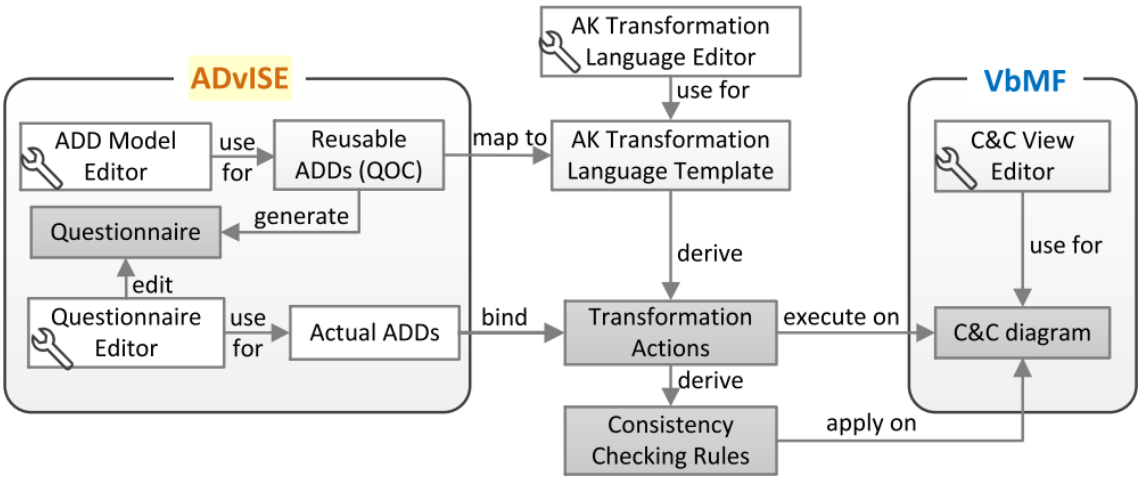

Figure 2.11: ADvISE and VbMF integration
Source: (Lytra et al., 2013)

Even though ADvISE seems to be the most suitable AKMS for component selection it lacks support for sharing and collaboration among different work groups, in fact in (Weinreich and Groher, 2016) ADvISE is said to miss sharing functionality — i.e. not a web app and lacks a mechanism to expose the collected knowledge in eclipse to the web for wide consumption and reuse. However, QOC (MacLean et al., 1991) as shown in (Lytra et al., 2013) can be useful to guide consumers towards a solution, in our case a component, through a series of choices which could be built collaboratively in a web app for a given domain — e.g. how to choose a datastore. In effect, (Lytra et al., 2015) follows an approach similar to the above but more guided by the synergies and trade-offs of quality attributes which could further clarify component selection since they are key in architectural design, see appendix D. Distancing itself from most AKMSs, seen above in Figure 2.8, which rely on extensive ADD mechanisms to avoid knowledge vaporisation (Bosch, 2004), but moving closer to Software Architecture Evaluation Methods such as Tradeoff Analysis Method (ATAM) (Clements et al., 2002), Cost Benefit Analysis Method (CBAM) (Kazman et al., 2001), Attribute Driven Design (Bass et al., 2002), and Quality-driven Product Architecture Derivation and Improvement (QuaDAI) (González-Huerta et al., 2013) that try to judge architectural decisions described through quality attributes in accordance to non-functional goals and scenarios (Lytra et al., 2015), see Figure 2.12 and 2.13.

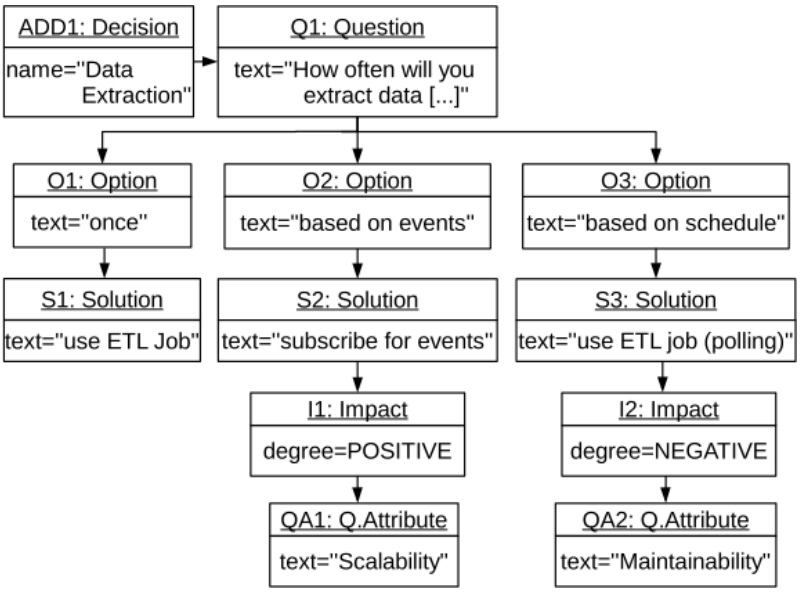

Figure 2.12: CoCoADvISE exemplary model
Source: (Lytra et al., 2015)

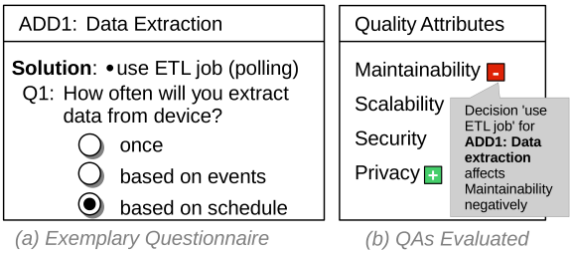

Figure 2.13: CoCoADvISE questionnaire
Source: (Lytra et al., 2015)

Unfortunately it is still not clear which AKMS approaches and knowledge better describe particular architectural tasks (Capilla et al., 2016), one of which is component selection — i.e. there are many AKMS approaches (Weinreich and Groher, 2016), see section E.1, with almost no experiments (Tofan et al., 2014; Capilla et al., 2016) even though research on knowledge-based architecture documentation has increased over the last decade (Ding et al., 2014a); also how can we extract knowledge from these systems to aid component consumers if almost no developers use them to record ADD (Capilla et al., 2016)? See Figure E.1 for some reasons why that is the case.

The relation between Component Selection and AKM is growing in importance because the inclusion of Open-source software (OSS) components into commercial software systems (Franch et al., 2013) and others is on the rise. Alas, most OSS do not have any architecture documentation especially those that are not related to industry, research or that have big teams (> 10 elements) (Ding et al., 2014b). When they do document they prefer Hyper Text Markup Language (HTML) (70.4%) followed by Pictures (20.4%), Wikis (8.3%), PDF (5.6%), Word (3.7%), and PPT (1.9%)

but with few artefacts (<= 3, 84.6%) focusing primarily on Model (98.1%), System (97.2%), Mission (91.7%), Environment (43.5%), Stakeholder (41.7%), Concern (33.3%), Rationale (18.5%), View (7.4%), Viewpoint (0%), Library Viewpoint (0%) elements and preferring natural language descriptions (88.9%) over diagrams (41.7%), UML (17.6%) , and lastly formal ADL (0%), see Figure 2.14 (Ding et al., 2014b).

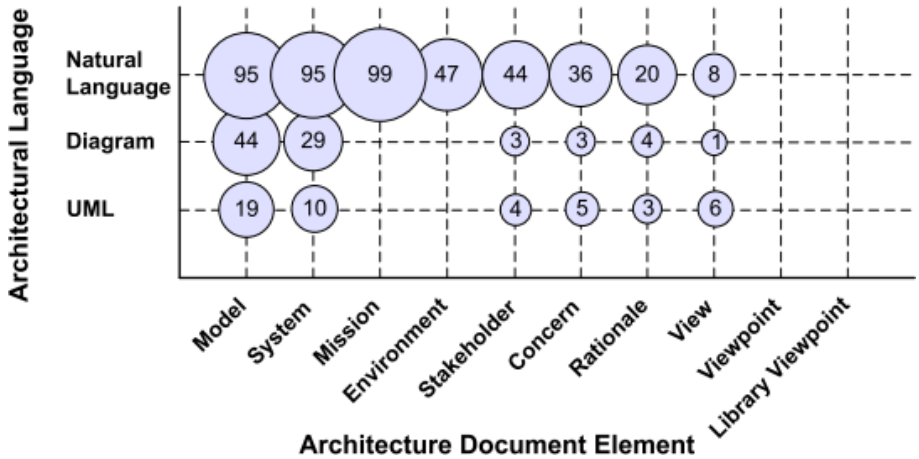

Figure 2.14: Distribution of OSS projects from SourceForge, Google Code, Github, and Tigris over ADL and architecture document elements
Source: (Ding et al., 2014b)

There are simpler alternatives to full-fledged AKMSs like Lightweight Architecture Decision Records (LADR)[12] and others[13,14]. LADR is of the opinion that well written code and tests is a form of documentation and that design decisions should be recorded alongside code instead of wikis, websites or other external tools. Command-line tools such as adr-tools[15] assist this process. However, these approaches are even worse than the ones seen above in regards to component selection because their selection depends on many criteria that are not captured; from the architectural design decisions developers opted for which might be related to a plethora of concerns such as desirable quality attributes in certain contexts, constraints because of other decisions, business or company tech know-how, among others to company and community support, licenses, popularity, relation to my particular problem space, etc.

There have also been attempts to extract architecture knowledge from developer communities (Soliman et al., 2018), source-code (Shahbazian et al., 2018; Mirakhorli and Cleland-Huang, 2016), and documentation (Slankas and Williams, 2013) using artificial intelligence techniques. Additionally, machine learning has been used to assist knowledge base curation (Gorton et al., 2017). But it is still a challenge that needs further researcher in the fields of software engineering and artificial intelligence (Gorton et al., 2017).

[12] https://www.thoughtworks.com/radar/techniques/lightweight-architecture-decision-records
[13] https://news.ycombinator.com/item?id=18874707
[14] https://news.ycombinator.com/item?id=19098926
[15] https://github.com/npryce/adr-tools

2.2 Component Selection

Component selection is complex and is seldom done in a structure way. It is a huge time sink since it is in practice an unstructured exploratory task where the relevant information is mostly hidden, out of date and lacking formalism (Gorton et al., 2015). In Subsection 2.2.1 we present and discuss an AKM approach in the form of a semantic wiki for the selection of big data systems. Subsection 2.2.2 describes a formal framework built using the Semantic Web Stack which can be generalised to components. Subsection 2.2.3 relates component selection to MCDM and discusses the difficulty in acquiring knowledge to feed such systems.

2.2.1 Quality Architecture at Scale for Big Data

In (Gorton et al., 2015), a dynamic knowledge base, *QuaABaseBD*, and a detailed feature taxonomy for designing big data systems with scalable database systems was constructed (Gorton et al., 2015). The taxonomy was derived from the authors experience in evaluating databases for big data systems and takes into account core architectural characteristics, the data model and query capabilities (Gorton et al., 2015).

Since the main objective of our work is to find a structured way to model component selection (Gorton et al., 2015) aids in determining the usefulness of a more technical approach to selection rather than relying only on common software quality parameters such as reliability, performance, security, consistency and so on. To validate the utility of the taxonomy they classified 9 database systems and encoded them in *QuaABaseBD*. Initially this task was done by the authors and then offloaded to graduate students who located and gathered features by reviewing the documentation of each database system (Gorton et al., 2015). According to (Gorton et al., 2015) the approaches that database systems follow in order to achieve certain quality goals can be distinguished by the data model and the data distribution architecture they support. Because of this, architectural and data characteristics of applications are greatly affected by database technology (Gorton et al., 2015). Having this in mind they structured the feature taxonomy as shown in Table 2.1:

Table 2.1: Organisation of features from (Gorton et al., 2015) updated with information from QuABaseBD

Categories	Features				
Data Architecture A.1	Data Model A.1.1	Query Languages A.1.2	Consistency A.1.3		
Software Architecture A.2	Scalability A.2.1	Data Distribution A.2.2	Data Replication A.2.3	Security A.2.4	Administration and Management A.2.5

All of the features above are further divided into a set of sub-features each of them having a set of allowed values (Gorton et al., 2015) as can be seen in the Tables of Appendix A. Since quality

assessment is futile without context the ontology uses other concepts such as scenarios, tactics and quality attributes to describe respectively architectural problems, approaches to solve those problems, and the advantages and disadvantages of using those approaches in terms of quality. Features integrate into the ontology by supporting certain tactics (Gorton et al., 2015). In essence, a tactic is a design decision that attempts to handle an architectural problem which has positive or negative impact on certain quality attributes and that can be realised by implementing certain features. The QuABaseBD ontology is presented visually in Figure 2.15:

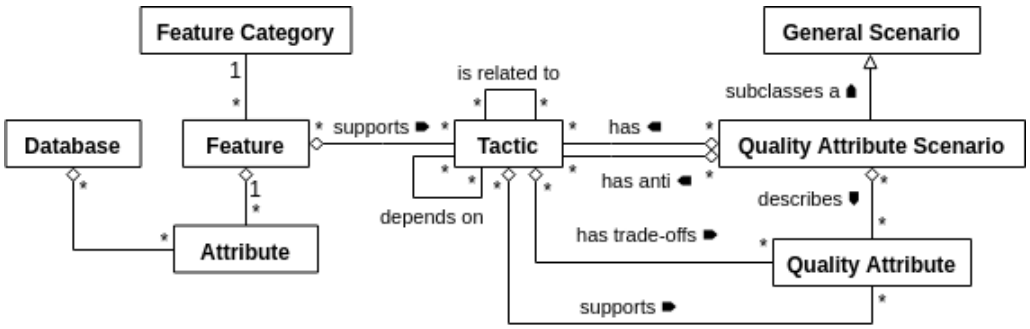

Figure 2.15: Logical structure of the QuABaseBD ontology

As can be seen in Figure 2.15 there are two sections with different intents separated by the software architecture concept tactics. The one to the right of tactics inclusive is meant to serve as a growing collection of concepts and terms needed to understand and reason about database technologies — i.e. architecturally significant requirements, quality attribute trade-offs, and how design tactics can solve certain architectural requirements (Gorton et al., 2015). The other side represents the actual features that databases implement following (Gorton et al., 2015)'s taxonomy as can be seen in appendix A. According to (Gorton et al., 2015) the linkage between these two sides through the features-tactics relationship is novel and allows architects and developers to reason about the features that are needed in order to achieve certain quality attributes. Moreover it can be used to compare the impact that different implementations of tactics (group of features) have on system qualities (Gorton et al., 2015). Their work relates to AKM and in a similar fashion to early developments on this field such as (Kruchten, 2004) builds a graph of related design alternatives and relationships for decision encoding. But designed to describe distributed databases instead of specific software projects (Gorton et al., 2015).

QuABaseBD, the knowledge base that implements the ontology, is encoded through the Semantic MediaWiki platform which builds upon concepts from the semantic web to give structure to wiki information. Through a combination of forms and templates it allowed them to represent novel domain knowledge through a medium that users are accustomed to (Gorton et al., 2015). The knowledge-base can be searched through three means:

- "Explore Software Design Principles", where one can browse for databases that support a certain quality attribute (Availability, Consistency, Performance, Scalability, Security) by

analysing Quality Attribute Scenarios and the Tactics that handle these.

- "Explore Database Technologies and Features", where it is possible to query per quality attribute for database features and to explore the features and tactics that a given database supports.
- "Explore Architecture Tactics for Big Data Systems" where you can browse for databases that implement certain tactics in order to satisfy some quality attribute scenario having a positive or negative impact on certain quality attributes.

To conclude the ontology was robust enough to encode all the functionally of the nine evaluated databases (Gorton et al., 2015). Also the QuABaseBD content is of high quality since it is intended to be validated by experts on each database through a systematic process (Gorton et al., 2015). However, almost no contributions were made since 18:59, 13 April 2016 as of 15:30, 21 January 2018. Which probably means that the choice of platform was not ideal, or the ontology was too low-level and consequently unable to be used by the average developer.

Nonetheless, the generalisation of the ontology for other component types could prove useful to ease component producers with decision recording through the vocabulary of scenarios and tactics while also providing consumers with structured data for the selection of components. Furthermore, pushing the component descriptions more closely to developers could also act as a way to increase contributions. More concretely, it would ease: the elaboration of selection criteria for component selection since the model's features are connected indirectly to scenarios and quality attributes, and the screening of candidate products because key characteristics are well segmented and externalised from software documentation enabling fast comparisons. Such would inform a more theoretical phase in the evaluation of software just before a more practical one using benchmarking techniques which is not the focus of this work — e.g. as done in (Klein et al., 2015a,b).

2.2.2 A formal Framework

In (Di Noia et al., 2018) an ontology based approach is used to define a theoretical framework and a semi-automated tool capable of: gathering structured information about design patterns and the families they belong to, as well as capturing the relationships between NFRs and design patterns. Even though (Di Noia et al., 2018) focuses on finding appropriate architectural design patterns through fuzzy modelling of NFRs its modelling ideas are useful for our study. Because only the end product is different, components instead of architectural design patterns, the modelling principles ought to be similar. This formalisation, first proposed in (Di Noia et al., 2015), makes it possible for architects to build a knowledge base of concepts and relations that aid in the architectural design patterns decision making process. Such is equally important in component selection since the criteria and rationale for selection lives mainly in the head of architects and lacks formal structure (Jansen and Bosch, 2005; Capilla et al., 2016; Di Noia et al., 2018).

The theory behind (Di Noia et al., 2015) and (Di Noia et al., 2018), mathematical fuzzy logic, is an extension of classical set theory (crisp sets). It allows one to say that a given value belongs to

a set with some degree of truth. In other words, we can say that the membership function of set A that indicates whether an element belongs to this set now returns a range of values between $[0,1]$ rather than either $\{0,1\}$. In a formal way we say that:

$$\mu_A : X \rightarrow [0,1]$$

where X represents the universal set (Straccia, 2013). The membership function in fuzzy logic is context dependent and can take many different shapes (Straccia, 2015). The most commonly used ones are the: trapezoidal; triangular; L-function; and, R-function (Straccia, 2015). This is particularly useful when modelling imprecise and vague concepts as is the case of NFRs (Di Noia et al., 2018). The encoding of the knowledge in (Di Noia et al., 2018) is done using the *fuzzy OWL 2* ontology which is based on fuzzy description logics (FDLs). Modelling in such a way makes it possible to represent and describe trade-offs and quality attributes of patterns in a fuzzy way (Di Noia et al., 2018):

- "For instance, in our context, FDLs allow one to model that “portability and adaptability are directly proportionate”, “stability and adaptability are inversely proportionate” (ontological knowledge) or that “the Adapter pattern has high portability” (factual knowledge)." (Di Noia et al., 2018)

- "Another type of expression allowed in our framework is “high adaptability implies a medium maintainability”. Let us note that in the previous statements, we can use fuzzy sets [71] to characterise concepts like high, medium and low." (Di Noia et al., 2018)

For instance, by substituting patterns (adapter, broker) for components (postgresql, mongodb, amqt) and pattern families (Adaptation and Extention, Distribution Infrastructure) for component domains (Relational databases, NoSQL, Message Queues) most of the formalisation logic remains the same. Relations between components and families, and NFRs and components are not always clear cut so a formalisation that enables both relational freedom and degree of truth makes sense.

Having knowledge structured in this way not only makes it possible to discover new one through combination of existing facts and relations but also enables automatic task selection from a set of requirements (Di Noia et al., 2018). Above else, it makes cooperation, sharing and data linking a possibility by relying on technology from the semantic web (fuzzy OWL 2 ontology and its many other layers, see Figure 2.17). The adaption of this technique to our problem means that components would be classified only by the families they belong to and the quality attributes they have, the combination is more than likely insufficient to select appropriate components. Moreover, it is difficult to determine values for quality-attributes in relation to components without some kind of context. To conclude, we think that statements such as "high adaptability implies a medium maintainability" might not always hold.

2.2.3 Multiple-criteria decision-making

Component Selection/Sourcing is often associated with the Operations Research sub-discipline MCDM. It deals with the modelling of conflicting multiple criteria which is common in software selection to deal with the evaluation of component origins (Commercial off-the-shelf (COTS), OSS, In-house, and Outsourcing) (Petersen et al., 2018); technology selection according to a set of criteria (Kaur and Singh, 2014; Kusters et al., 2016; Trienekens et al., 2017; Garg et al., 2017; Farshidi et al., 2018a,b,c); and, any problem where rigorous decision and planning is essential. For instance, in (Farshidi et al., 2018c) the decision model is a set of rules, facts and preferences that guide users towards a technology choice. To do so they have to hard-code a set of matrices that indicate the alternatives that a domain has, the features that domain supports, and the quality attributes that each feature obeys, see Figure 2.16. From these matrices a custom user preference in the form of a rule is consumed to rank the feasible alternatives whose scores are calculated using the Weighted Sum Model (Farshidi et al., 2018c).

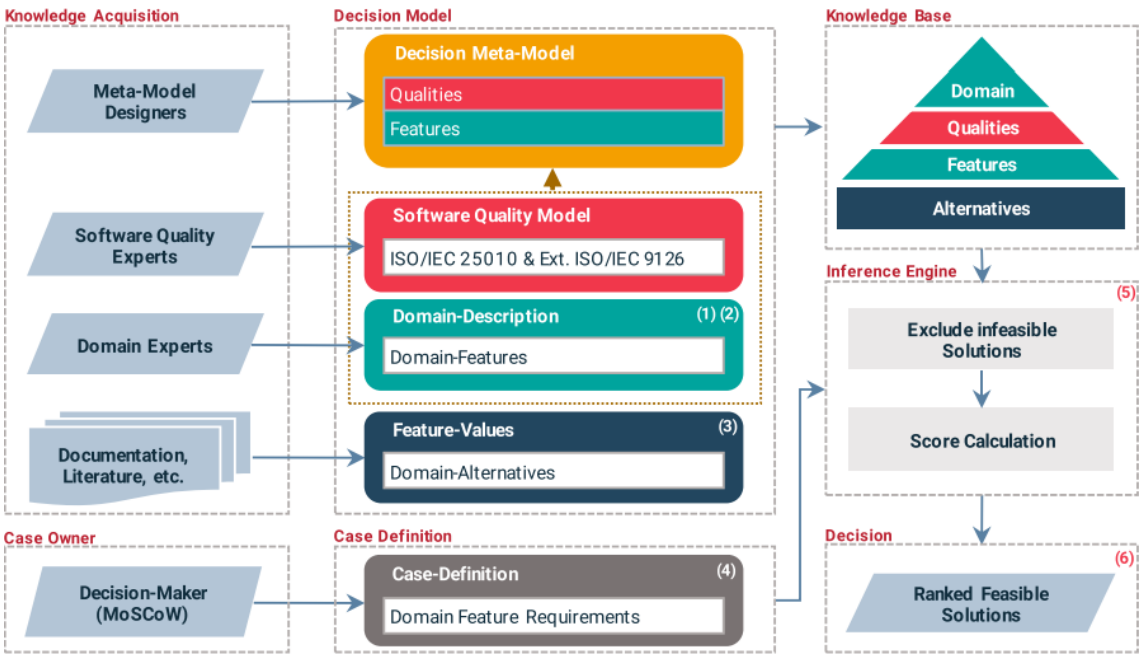

Figure 2.16: A Decision Support System for Technology Selection
Source: (Farshidi et al., 2018c)

The problem with using MCDM to tackle Software Selection using Software Architecture Knowledge in this way is that the relation between quality-attributes and features is highly dependent on problem context which is variable between domains and may involve lots of independent variables and rules to infer meaningful knowledge. For instance, read and write performance in datastores is highly reliant on the characteristics of data and access patterns, so a feature to do with partitioning methods (scatter and grab?, local secondary indexes?, hash-based?, value-based?, range scans?, etc) would have to have this information in mind to be linked accurately to

a quality attribute. A formalism to make such inferences would be extremely useful but unwieldy and difficult to contribute to. Not to mention the arduous and repetitive work of having to scan documentation to fill feature matrices and other information. The construction of an authoritative source of this type of knowledge which any interested party could consume is probably time better spent at least for now.

2.3 Data Interchange Models & Formats

As one of the main problems of Component Selection is data reuse it makes sense to analyse the perks of different Data Interchange Models & Formats. Ideally a knowledge base for Component Selection should be easy to share, consume, search and combine by interested parties.

2.3.1 RDF, RDFS, SPARQL, OWL, Linked Data

The traditional web, Web 1.0[16] and Web 2.0[17], is a set of content pages lacking meta-data information about its subjects, descriptions, etc with connections to other pages only through hyperlinks. To tackle this issue a set of proposals were made dating back as far as 1989[18] to extend the capabilities of the web to support structured data (Bizer et al., 2009). The tenet of these ideas is to shift from only human-readable documents to more machine-readable semantic information (Berners-Lee et al., 1994) thereby creating a Web of Data or Data Web that can be processed by machines (Berners-Lee and Fischetti, 1999), that is a Semantic Web[19] or as some call it Web 3.0[20]. To achieve this vision many standards came about as can be seen in Figure 2.17.

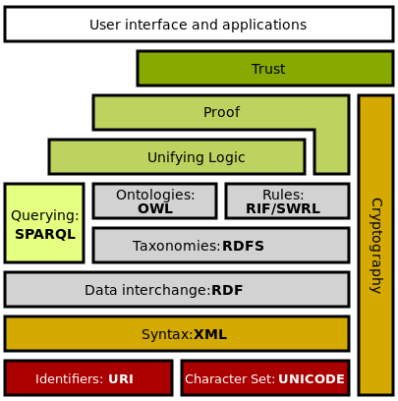

Figure 2.17: Semantic Web Stack
Source: https://commons.wikimedia.org/wiki/File:Semantic_web_stack.svg

[16] https://computer.howstuffworks.com/web-101.htm
[17] https://computer.howstuffworks.com/web-20.htm
[18] http://www.w3.org/History/1989/proposal.html
[19] http://web.archive.org/web/20070713230811/http://www.sciam.com/print_version.cfm?articleID=00048144-10D2-1C70-84A9809EC588EF21
[20] https://www.w3.org/2007/Talks/0123-sb-W3CEmergingTech

RDF[21] is a framework for modelling information using a set of statements in the form of (subject, predicate, object) called triples designed for combining information that originates from multiple sources (Heath and Bizer, 2011). To make it possible for data to be combined and grouped to avoid collisions subjects, predicates and objects can be identified through an Internationalized Resource Identifier (IRI) which is an Uniform Resource Identifier (URI)[22] with support for Unicode Characters. In other words a string that uniquely identifies a resource. RDF Links (predicates) connect subjects and objects together creating a global data graph (Bizer et al., 2009). Internal RDF Links connect resources within a single Linked Data Source while external RDF Links connect resources that are served by different Linked Data sources (Heath and Bizer, 2011). URIs in most cases take the form of an Hyper Text Transfer Protocol (HTTP) Uniform Resource Locator (URL) (which is an URI) for the simple fact that it provides a simple way to look-up more information about the respective term (Bizer et al., 2009) as well as enabling the domain owner to create new globally unique URIs in a decentralised fashion (Heath and Bizer, 2011). As a result external users can reference RDF subjects, predicates and objects defined elsewhere building a navigable semantic structured web of data. For that reason, a collection of IRIs meant for reuse is called an RDF vocabulary. And a set of triples is called an RDF graph, see Figure 2.18. RDF itself does not specify any serialisation formats however it is normally associated with eXtensible Markup Language (XML). Many more serialisation formats exist for RDF graph encoding[23], see Listing 1.

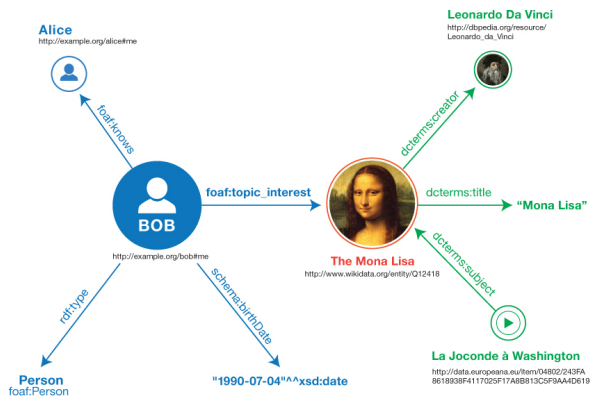

Figure 2.18: Example of an RDF graph
Source: https://www.w3.org/TR/rdf11-primer/

[21] http://www.w3.org/TR/rdf11-concepts/
[22] https://tools.ietf.org/html/rfc3986
[23] https://www.w3.org/TR/rdf11-primer/#section-graph-syntax

```
BASE   <http://example.org/>
PREFIX foaf:     <http://xmlns.com/foaf/0.1/>
PREFIX xsd:      <http://www.w3.org/2001/XMLSchema#>
PREFIX schema:   <http://schema.org/>
PREFIX dcterms:  <http://purl.org/dc/terms/>
PREFIX wd:       <http://www.wikidata.org/entity/>

    <bob#me>
        a foaf:Person ;
        foaf:knows <alice#me> ;
        schema:birthDate "1990-07-04"^^xsd:date ;
        foaf:topic_interest wd:Q12418 .

    wd:Q12418
        dcterms:title "Mona Lisa" ;
        dcterms:creator <http://dbpedia.org/resource/Leonardo_da_Vinci> .

    <http://data.europeana.eu/item/04802/243FA8618938F4117025F17A8B813C5F9AA4D619>
        dcterms:subject wd:Q12418 .
```

Listing 1: Example of the Turtle serialisation format
Source: https://www.w3.org/TR/rdf11-primer/

RDFS[24], is a vocabulary description language which provides a means to give more semantic meaning to RDF triples, a lightweight ontology also known as a vocabulary (Heath and Bizer, 2011). This is achieved by using its own RDF vocabulary to describe a set of constraints for subjects and properties in a similar way to an Object-oriented programming (OOP) type system (W3C, 2014). With the exception that properties in RDF are separate from classes giving it the ability to determine the type of the subject and object by inferring from its explicit domain and range. For instance it is possible to say that: subject S1 is a instance of class C1, class C2 is a subclass of class C1, property P1 has a given domain and range, property P1 is a sub-property of property P2, and so on. Because RDFS is just another RDF vocabulary we can use any RDF serialisation format to make these kind of semantic descriptions, see Figure 2.19 and Listing 2.

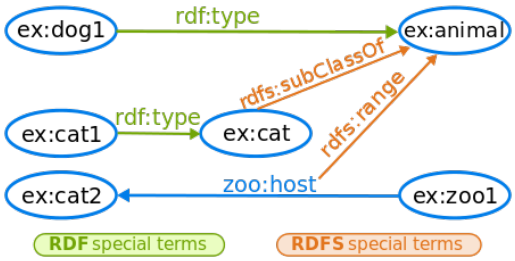

Figure 2.19: Example of an RDF graph with the RDFS vocabulary
Source: https://en.wikipedia.org/wiki/RDF_Schema

[24] https://www.w3.org/TR/rdf-schema/

```
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix ex:    <http://example.org/> .
@prefix zoo:   <http://example.org/zoo/> .

ex:dog1     rdf:type            ex:animal .
ex:cat1     rdf:type            ex:cat .
ex:cat      rdfs:subClassOf     ex:animal .
zoo:host    rdfs:range          ex:animal .
ex:zoo1     zoo:host            ex:cat2 .
```

Listing 2: Example of a RDFS description in Turtle
Source: https://en.wikipedia.org/wiki/RDF_Schema

Even though RDF is meant for internet-wide data exchange its triple-store data model is much more than just the Semantic Web (Kleppmann, 2017). It is similar to the Property Graph Model with the exception that everything is expressed through predicates including node properties and relations to other nodes, à la ternary facts in Prolog[25]. As a matter of fact there are technologies that have nothing to do with the Semantic Web that follow this concept, such as Datomic and its supporting query language Datalog (Kleppmann, 2017). Furthermore, the systems that store triples are called triple-stores[26] and most of them ingest some kind of RDF format[27]. To conclude, an RDF graph can be retrieved and manipulated in triple-stores or tools that implement the SPARQL Protocol and RDF Query Language (SPARQL)[28] declarative querying language. Each variable in a query ?foo can be associated with an amalgam of triples to bind them together as seen with ?person in Listing 3.

```
# Gets the name and email of all the subjects that are an
# instance of Person and have at least one name and email

PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT  ?name
        ?email
WHERE
    {
        ?person  a          foaf:Person .
        ?person  foaf:name  ?name .
        ?person  foaf:mbox  ?email .
    }
```

Listing 3: Example of a SPARQL query
Source: https://en.wikipedia.org/wiki/SPARQL

SPARQL[29] is also capable of querying named graphs which are RDF graphs identified by an

[25] https://www.metalevel.at/prolog
[26] https://en.wikipedia.org/wiki/Comparison_of_triplestores
[27] https://en.wikipedia.org/wiki/Resource_Description_Framework#Serialization_formats
[28] https://www.w3.org/TR/sparql11-query/
[29] https://www.w3.org/wiki/SparqlImplementations

IRI which are meant to organise statements according to different contexts, concerns, etc. Interestingly an RDF dataset is a collection of graphs built out of many named graphs but only with one default/unnamed one. See Listing 4.

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?homepage
# Present in a SPARQL-capable store in a named graph with IRI http://example.org/joe.
FROM NAMED <http://example.org/joe>

WHERE {
    GRAPH ?g {
        ?person foaf:homepage ?homepage .
        ?person foaf:mbox <mailto:joe@example.org> .
    }
}
```

Listing 4: Example of named graph SPARQL query
Source: https://en.wikipedia.org/wiki/Named_graph

Web Ontology Language (OWL)[30] is a declarative logic based language whose purpose is to augment the RDFS conceptual model. It allows for richer metadata descriptions to better frame a domain of knowledge, an ontology. And is no more than an RDF vocabulary meant to supply additional semantics for a reasoner[31] to infer additional knowledge. For instance it is possible to state that: two classes are equivalent; a class is a union or intersection of classes; specify restrictions in terms of values, cardinalities, etc; declare that object properties are inverse, symmetric, asymmetric, disjoint, reflexive, and so on; etc. Tools such as Protégé[32] and others[33] assist in the creation and editing of ontologies. See Listing 5.

[30] https://www.w3.org/TR/2012/REC-owl2-primer-20121211/
[31] https://www.w3.org/2001/sw/wiki/OWL/Implementations#Reasoners
[32] https://protege.stanford.edu/
[33] https://www.w3.org/wiki/Ontology_editors

```
@prefix : <http://example.com/owl/families/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .

:hasSpouse  rdf:type        owl:SymmetricProperty .

:Mother     owl:equivalentClass [
                rdf:type            owl:Class ;
                owl:intersectionOf  ( :Woman :Parent )
] .

:Parent     owl:equivalentClass [
                rdf:type            owl:Class ;
                owl:unionOf         ( :Mother :Father )
] .

:Jack       rdf:type [
                rdf:type            owl:Class ;
                owl:intersectionOf (
                    :Person
                    [ rdf:type owl:Class ; owl:complementOf :Parent ]
                )
] .
```

Listing 5: Example of a OWL description in Turtle
Source: https://www.w3.org/TR/2012/REC-owl2-primer-20121211/

Linked Data defines a set of principles to create machine-readable linked knowledge from different data sources and heterogeneous systems using standard Semantic Web tools (Bizer et al., 2009). That is a collection of linked datasets sharing relationships that act as the backbone of the Semantic Web (Web of Data, Data Web, Web 3.0). In other words, an attempt to bridge information islands by relating and composing with other ones (Heath and Bizer, 2011). Much like what the Web does for interconnecting HTML content pages through Hyperlinks creating a global *information space* with the twist that Linked Data Hyperlinks are used to connected disparate data into a global *data space* (Heath and Bizer, 2011). According to Tim Bernes-Lee Linked Data should:

- Use URIs to identify world objects and concepts and not just Web Documents and Digital content through Hyperlinks (Heath and Bizer, 2011; Tim Berners-Lee, 2006)
- Use HTTP URIs identifiers so that more information can be looked up about the object or concept over the HTTP protocol (Heath and Bizer, 2011; Tim Berners-Lee, 2006).
  - Get the HTML,

```
curl --header "Accept: text/html" --request GET
 ↪  "http://xmlns.com/foaf/0.1"
```

  - Get the RDF document,

```
curl --header "Accept: application/rdf+xml" --request GET -L
 ↪  "http://xmlns.com/foaf/0.1/"
```

- Use the standard RDF graph model and the SPARQL query language when structured data is looked up using URIs. (Heath and Bizer, 2011; Tim Berners-Lee, 2006)
- Use RDF Hyperlinks/Links that is typed Hyperlinks/Links (predicates identified with an URI) to connect to any type of thing. Instead of just Web Documents as is the case of normal HTTP hyperlinks. (Heath and Bizer, 2011; Tim Berners-Lee, 2006)

By following these principles all information is related through common unique predicates, subjects and objects building what is called a **giant global graph** (Heath and Bizer, 2011). Linked Data applications can then look up parts of the Linked Data global graph by dereferencing URIs (Heath and Bizer, 2011). Because there can be many different RDF datasets[34] spread across the web equal concepts can be represented with different URIs. For that reason it is advised to use well-known vocabularies[35]. However it is possible to link URIs using the owl *owl:sameAs* predicate. URIs can be looked up using the HTTP protocol to retrieve HTML representations or Linked Data RDF documents in two ways: 303 (See other) URIs, and Hash URIs (Heath and Bizer, 2011). The former involves issuing a *Get request* to a common *HTTP path* with a *crafted Accept Header* and then receiving a *303 See Other* indicating the path to the document with the respective format, see Listing 6, and 7.

```
GET /people/dave-smith HTTP/1.1
Host: biglynx.co.uk
Accept: text/html;q=0.5, application/rdf+xml
```

Listing 6: Dereferencing 303 URIs Request
Source: (Heath and Bizer, 2011)

```
HTTP/1.1 303 See Other
Location: http://biglynx.co.uk/people/dave-smith.rdf
Vary: Accept
```

Listing 7: Dereferencing 303 URIs Response
Source: (Heath and Bizer, 2011)

On response arrival, Listing 7, an *HTTP Get request* is then made using the *Header Location* to retrieve the appropriate document. The latter approach involves making just one *HTTP Get request* with an appropriate *Accept Header* and fragment identifier to get the RDF document without redirects, see Listing 8 and 9.

```
GET /vocab/sme HTTP/1.1
Host: biglynx.co.uk
Accept: application/rdf+xml
```

Listing 8: Dereferencing Hash URIs Request
Source: (Heath and Bizer, 2011)

[34] https://lod-cloud.net/
[35] https://lov.linkeddata.es/dataset/lov

```
HTTP/1.1 200 OK
Content-Type: application/rdf+xml;charset=utf-8

<rdf:RDF
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#" >
        ...
```

Listing 9: Dereferencing Hash URIs Response
Source: (Heath and Bizer, 2011)

Because HTTP fragments[36] — e.g *http://biglynx.co.uk/vocab/sme**#Team*** — are not sent to HTTP servers it is up to the client application to handle the whole RDF document and only then process the fragment — e.g disregarding everything but the fragment, pinpointing the fragment in the document, etc. Even tough this approach reduces round-trips everything under the HTTP path is downloaded which is obviously not ideal when only a part is needed (Heath and Bizer, 2011). One the other hand, 303 URIs are more flexible since each resource can be identified by a path which may point to different redirection targets — e.g. an RDF document per person in separate URIs and possibly different datasets (Heath and Bizer, 2011) — see the *Location Header* in Listing 7. For this reason 303 URIs are normally used for very large RDF datasets such as DBPedia, see Listing 10, while RDF vocabularies such as http://www.foaf-project.org/ often use Hash URIs (Heath and Bizer, 2011). Fortunately it is possible to combine both methods — e.g. *http://biglynx.co.uk/vocab/sme/**Team#this*** (Heath and Bizer, 2011).

```
curl --header "Accept: application/rdf+xml" --request GET
 ↪  http://dbpedia.org/resource/Britney_Spears -v
```

```
HTTP/1.1 303 See Other
Content-Type: application/rdf+xml
Server: Virtuoso/07.20.3230 (Linux) x86_64-generic-linux-glibc25  VDB
TCN: choice
Vary: negotiate,accept
Alternates: ...
Link: <http://creativecommons.org/licenses/by-sa/3.0/>;rel="license",
<http://dbpedia.mementodepot.org/timegate/http://dbpedia.org/resource/Britney_Spears>;
rel="timegate"
Location: http://dbpedia.org/data/Britney_Spears.xml
```

Listing 10: Britney Spears DBPedia Partial Response

The dereferenced RDF Links may be of tree kinds:

- Relationship Links, relating subjects in one dataset to objects in another one, which in turn might point to entities in other datasets (Heath and Bizer, 2011), see Line 14 in Listing 1.
- Identify Links, different URIs can represent the same concept because anyone can create a Web Server under a domain name that they control to expose RDF triples (Heath and Bizer,

[36] https://en.wikipedia.org/wiki/Fragment_identifier#Examples

2011). For this reason, URIs that refer to equal concepts are called URI aliases and can be identified through the owl `http://www.w3.org/2002/07/owl#sameAs` predicate. The plurality of statements about equal concepts and the **owl:sameAs** predicate make it possible for (Heath and Bizer, 2011):

  - Web Publishers to express **different opinions** (Heath and Bizer, 2011)
  - **Trace opinions** about the same concept since the Web Publisher opinions are identified by an URI that they control and expose (Heath and Bizer, 2011)
  - **Avoid central points of failure** since there is no centralised URI naming authority to map a concept to a single URI (Heath and Bizer, 2011) — e.g. the unavailability of an URI does not obliterate the concept described since other URI aliases exist.
  - As there is no central naming authority the Web of Data relies on **evolutionary and distributed identity resolution** using the *owl:sameAs* predicate to make it easier for users to publish their statements under a URI without having to worry at first with other URIs that represent the same concept (Heath and Bizer, 2011). Even tough OWL semantics treat RDF statements as facts the predicate *owl:sameAs* is used in the Linked Web more as a way to identify different claims (Heath and Bizer, 2011).

- Vocabulary Links, RDF links may point to new RDF datasets but for data to be meaningful a set of common semantic descriptions is needed. The integration of data relies on common terminology identifiable through URIs defined in popular vocabularies to reduce heterogeneity. However, when there are no appropriate terms in vocabularies or only a subset is found new ones should be defined and used to describe RDF statements (Heath and Bizer, 2011). In the advent of similar terms in other vocabularies the publisher ought to identify similar URIs using appropriate RDF statements (Heath and Bizer, 2011), see Listing 11:

  - From OWL: *owl:equivalentClass*, *owl:equivalentProperty* (Heath and Bizer, 2011)
  - From RDFS: *rdfs:subClassOf*, *rdfs:subPropertyOf* (Heath and Bizer, 2011)

```
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-n#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .
@prefix co:   <http://biglynx.co.uk/vocab/sme#> .

<http://biglynx.co.uk/vocab/sme#SmallMediumEnterprise>
  rdf:type rdfs:Class ;
  rdfs:label "SmallorMedium-sizedEnterprise" ;
  rdfs:subClassOf <http://dbpedia.org/ontology/Company> ;
  rdfs:subClassOf <http://umbel.org/umbel/sc/Business> ;
  rdfs:subClassOf <http://sw.opencyc.org/concept/Mx4rvVjQNpwpEbGdrcN5Y29ycA> ;
  rdfs:subClassOf <http://rdf.freebase.com/ns/m/0qb7t> .
```

Listing 11: Relating SmallMediumEnterprise to other vocabulary terms
Source: (Heath and Bizer, 2011)

As we saw in section 2.2.2 building an ontology to describe a nuanced topic, as is the case of Component Selection, is a challenging endeavour with no single solution. All in all, as Component Selection is yet to mature adding too much formalism or strictness will likely hamper adoption and contributions. Building a lightweight ontology to capture architecture details first and then progressing towards an ontology is probably a wiser choice.

Several AKMSs have tried to leverage these technologies to build taxonomies and ontologies to support structured knowledge capture (Ding et al., 2014a; De Graaf et al., 2016; Sabou et al., 2018) but as mentioned previously adoption is still an issue. Other more lightweight approaches as the encoding of architecture knowledge through semantic wikis — e.g. Semantic MediaWiki — face the same issues regardless of improved content browsing and organisation. Furthermore, Design Rationale approaches such as ISAA (Zhang et al., 2013), and Exploratory Search for Architecture Knowledge in Enterprises (Sabou et al., 2018) have also adopted semantic structures and approaches but are still in their infancy. To conclude, one feature that seems yet to be explored is the use of Resource Description Framework in Attributes (RDFa)[37] to embed Architecture Knowledge in HTML pages supplying a means for web crawlers to extract structured information about architecture concepts, component features and others. Also, the addition of RDFa meta-data in the HTML documentation of software projects would probably be a good means to sum up important features as well as improving search and software comparisons. See Listing 12.

```
<html prefix="dc: http://purl.org/dc/elements/1.1/" lang="en">
  <head>
    <title>John's Home Page</title>
    <link rel="profile" href="http://www.w3.org/1999/xhtml/vocab" />
    <base href="http://example.org/john-d/" />
    <meta property="dc:creator" content="Jonathan Doe" />
    <link rel="foaf:primaryTopic" href="http://example.org/john-d/#me" />
  </head>
  <body about="http://example.org/john-d/#me">
    <h1>John's Home Page</h1>
    <p>My name is <span property="foaf:nick">John D</span> and I like
      <a href="http://www.neubauten.org/" rel="foaf:interest"
        lang="de">Einstürzende Neubauten</a>.
    </p>
    <p>
      My <span rel="foaf:interest" resource="urn:ISBN:0752820907">favorite
      book is the inspiring <span about="urn:ISBN:0752820907"><cite
      property="dc:title">Weaving the Web</cite> by
      <span property="dc:creator">Tim Berners-Lee</span></span></span>.
    </p>
  </body>
</html>
```

Listing 12: Example of an RDFa description
Source: https://en.wikipedia.org/wiki/RDFa#HTML5_+_RDFa_1.1

[37] https://www.w3.org/TR/rdfa-primer/

2.3.2 JSON, XML, JSON Schema, XML Schema, YAML

JavaScript Object Notation (JSON) and XML are very well known and supported human readable textual encondings. For some time now XML has been loosing ground to JSON primarily on Web facing applications because the latter is way less verbose in most cases, and much closer to the representation of Javascript objects. Also there is growing support for JSON document validation using JSON schema implementations[38] which was one of the strong factors in picking XML[39]. Nevertheless XML still has a much larger and more mature ecosystem with many standards for document namespacing, transformation, querying, encryption, stream-oriented parsing, etc[40,41] which is paramount for some enterprise applications and legacy systems. Moreover, its verbosity and file size can be reduced by respectively using element attributes instead of sub-elements when appropriate[42], and compression methods[43,44] that take advantage of its repetitive syntax. Lastly, XML supports comments, the separation of metadata and data through respectively attributes and elements, and is also a markup language.

YAML Ain't Markup Language (YAML) uses python style indentation and is a superset of JSON since version 1.2[45]. Compared to JSON it is more human readable making it a better target for configuration files. In addition it supports: multiple documents within a single file, comments, embedded block literals[46], relational anchors, extensible data-types, and mapping types preserving key order; see `https://learnxinyminutes.com/docs/yaml/`. However, JSON is a better serialisation format since it has a much simpler specification. Unfortunately YAML still lacks well established support for schema validation. However it can be binded to XML using *YAXML*[47] to leverage the XML stack. Also, even tough JSON is a subset of YAML most features can be mapped to the former. Hence JSON schema can be used in most cases[48].

To avoid any confusions around RDF/XML, XML is a serialisation format and markup language while RDF is a data model. In other words, XML is the delivery mechanism while RDF represents the actual information and its meaning. In fact, RDF can be expressed in many formats.

2.3.3 Avro, Protocol Buffers, Thrift

Binary encoding standards like Avro[49], Protocol Buffers[50], and Thrift[51] reduce encoding file size, and fix problems related to JSON, and XML textual encodings, such as:

[38] https://json-schema.org/implementations.html
[39] https://en.wikipedia.org/wiki/XML_schema
[40] https://en.wikipedia.org/wiki/XML#Related_specifications
[41] https://en.wikipedia.org/wiki/XML#Programming_interfaces
[42] https://stackoverflow.com/questions/1096797/should-i-use-elements-or-attributes-in-xml
[43] https://www.w3.org/XML/EXI/
[44] https://www.usenix.org/legacy/events/expcs07/papers/7-augeri.pdf
[45] https://en.wikipedia.org/wiki/JSON#YAML
[46] https://yaml-multiline.info/
[47] https://yaml.org/xml
[48] https://json-schema-everywhere.github.io/yaml
[49] https://avro.apache.org/
[50] https://developers.google.com/protocol-buffers/
[51] https://thrift.apache.org/

- JSON ambiguity when dealing with numbers; XML does not distinguish between numbers and strings except when an external schema is used (Kleppmann, 2017).

- JSON and XML use binary-to-text encoders like base64 to embed binary information (Kleppmann, 2017).

- Schema support for XML and JSON is quite complex to learn and in the case of the latter infrequently used harming correct interpretation for numbers and binary strings (Kleppmann, 2017).

For all these problems binary encodings are a good candidate for internal use in organisations (Kleppmann, 2017). JSON and XML do have binary versions for encoding respectively MessagePack, BSON, BJSON, YBJSON, BISON, Smile, etc, and WBXML, Fast Infoset, etc but they embed object field names inside the payload instead of using a schema (Kleppmann, 2017). Thus increasing file size.

Protocol Buffers and Thrift are very identical in the way they do the encoding unlike Avro. In contrast, it ditches the data type information and fields identification in the payload by requiring the writer's schema to do the decoding (Kleppmann, 2017). Because both the reader and the writer have their own schemas they can be compared and resolved to achieve a successful data translation to the reader's schema (Kleppmann, 2017), see Figure 2.20.

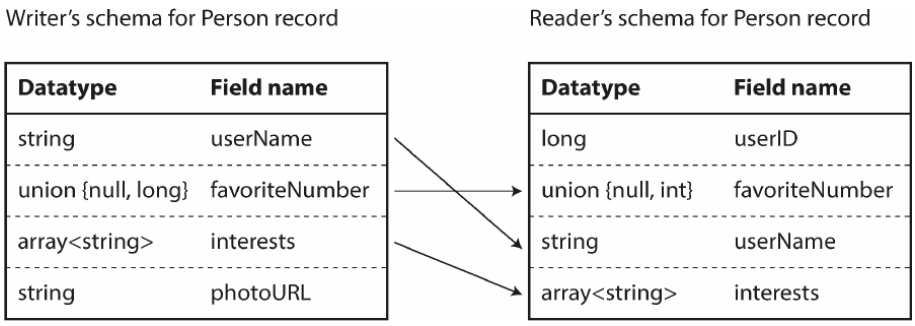

Figure 2.20: Resolving differences between the writer's and reader's schema
Source: (Kleppmann, 2017)

Avro's schema resolution technique could be an interesting way to compare the features of different components by contrasting their schemas using JSON and JSON Schema. As an example, JSON schemas could define targets that components of certain types could instantiate to indicate their functionalities. Software could then be compared even if their schemas are different but share some similarities. Each schema would represent a software domain with a set of appropriate feature values — e.g. Message Queues Features. As an example, an instance of *Datastores* could probably be comparable to an instance of *Message Queues* since their schemas more likely than not share concepts.

2.4 Comparison Websites

In this section we describe websites that assist users in contrasting and selecting components. Subsection 2.4.1, analyses Multi-Faceted Comparison Websites. And lastly, Subsection 2.4.2 discusses Feature-Comparison Websites.

2.4.1 Multi-Faceted Comparison Websites

Herein we describe websites that implement several mechanisms that assist component selection including in some cases component feature comparisons.

2.4.1.1 Slant

In Slant questions can be asked and tagged with fields to better find them. Questions have a set of options that can be up-voted. Each option has a set of experiences, pros, cons, and specs. Pros, and cons are user provided unstructured text and can be up or down voted. Experiences are user provided reviews in unstructured text with embedded support for pros and cons which users can mark as helpful. Specs is a key-value list of properties. It is not easy to compare options under a question because everything is unstructured, unrelated, and entirely subject to the whims of users. So the question has to be very specific so that users can up-vote the clear winner. Same options in different questions don't share a single thing so content like pros, cons, specs and if appropriate experiences have to be repeated occasionally. Edits are last write wins but activity is tracked and certain edits require karma. See Figures 2.21, 2.22, and 2.23.

BEST BACKEND WEB FRAMEWORKS	PRICE	WRITTEN IN	PLATFORMS
94 Phoenix (Elixir)	-	Elixir	Linux, Windows
85 Django	-	Python	-
82 Express.js	-	JavaScript	Cross-platform
82 Ruby on Rails	-	-	-
81 Flask	-	Python	-

SEE FULL LIST

Figure 2.21: Options for the question, What are the best backend web frameworks?
Source: https://www.slant.co/topics/362/~best-backend-web-frameworks

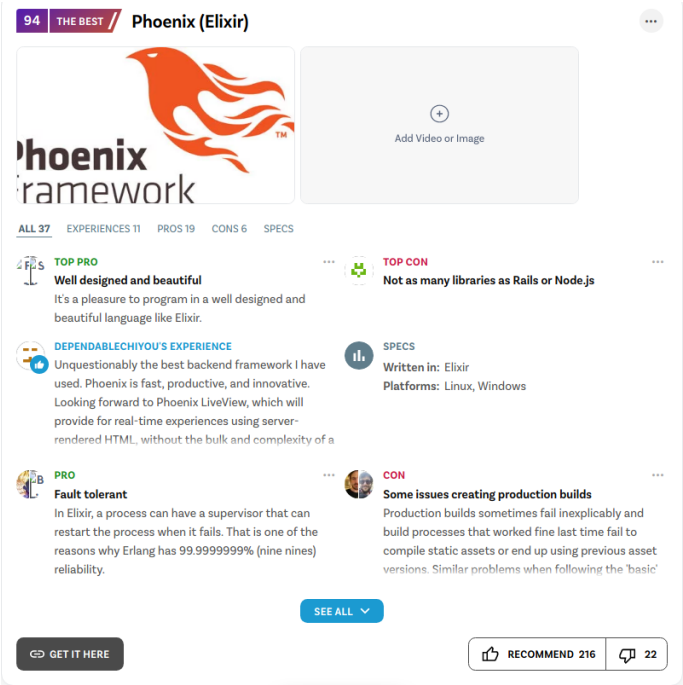

Figure 2.22: Phoenix details for question, What are the best backend web frameworks?
Source: https://www.slant.co/topics/362/~best-backend-web-frameworks

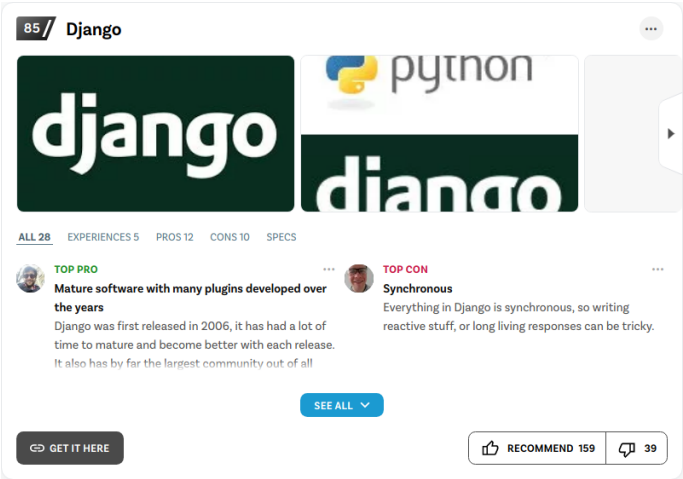

Figure 2.23: Django details for question, What are the best backend web frameworks?
Source: https://www.slant.co/topics/362/~best-backend-web-frameworks

2.4.1.2 Versus

Versus follows a structured approach to describe options. Each option belongs to a single category organised by sections, features and values. Only options from the same category are comparable and analysable through tables, graphs and charts. Users are only allowed to comment, and up or down vote the pertinence of features. Also the categories they focus on (Smartphones, Cameras,

Cities, etc) don't overlap which is sometimes not the case in software — e.g. Graph database, Relational database and NoSQL replication features. See Figures 2.24, 2.25, and 2.26.

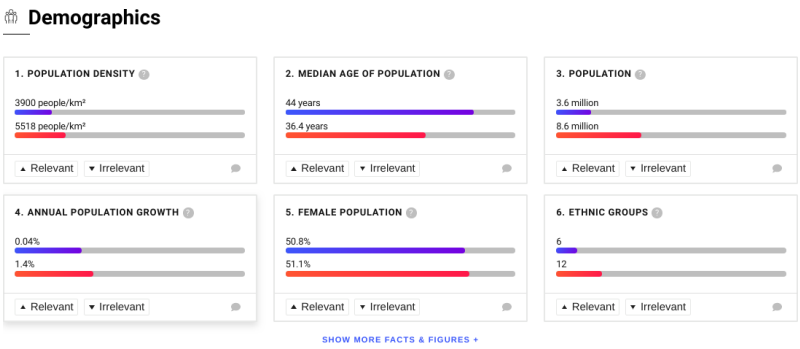

Figure 2.24: Berlin vs London Demographics Category
Source: https://versus.com/en/berlin-vs-london

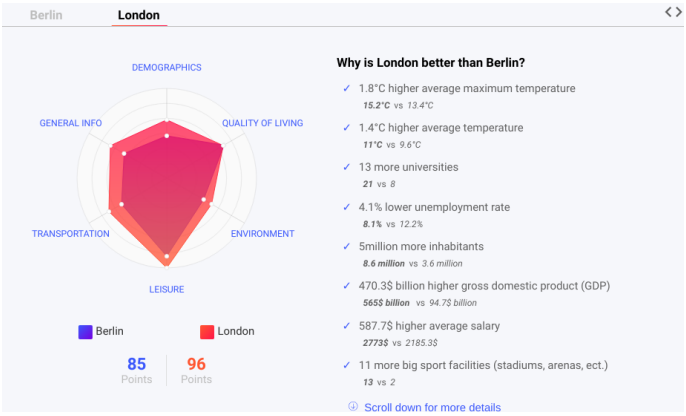

Figure 2.25: Berlin vs London Radar Chart
Source: https://versus.com/en/berlin-vs-london

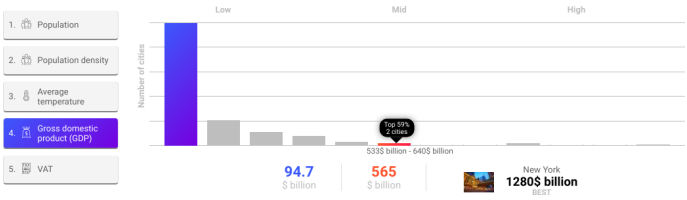

Figure 2.26: Berlin vs London Key Facts Chart
Source: https://versus.com/en/berlin-vs-london

2.4.1.3 StackShare

StackShare is a software discovery platform whose primary goal is to provide more meaningful software choice information by combining software stacks, tools, stack decisions, and job search.

It has three main functionalities: browsing stacks, exploring tools, comparing tools, and job search. Each tool page has a set of sections whose information in some cases is derived from software stack decisions: **What is Foo?**, is a brief textual description of the tool; **Foo stack decisions & reviews**, shows all the stack decisions that contain the Foo tool (see Figure 2.27); **Who uses Foo?**, lists all the companies and apps that use a stack that contains the Foo tool; **Foo integrates with**, seems to be added manually by the creator of the tool's page; **Why people like Foo**, is a set of one-line reasons for using the tool that can be up-voted, which is similar to slant; **Foo's alternatives**, is probably derived from the most common tool comparisons in the same group that involve Foo; **Explore other languages & frameworks tools that are know for**, links to other tools from the same category that share the most relevant reasons seen in *Why people like Foo*; **Similar tools & services**, shows other tools in the same group. It is also possible to look-up and apply for jobs that require the Foo tool. New tool submissions are vetted[52] and its page can be claimed by supplying a company email[53]. Tools are organised hierarchically through an uneditable tree of depth 3, *Home -> MainCategory -> Category -> Group*, for instance *Home -> Application and Data -> Data Stores -> Databases*.

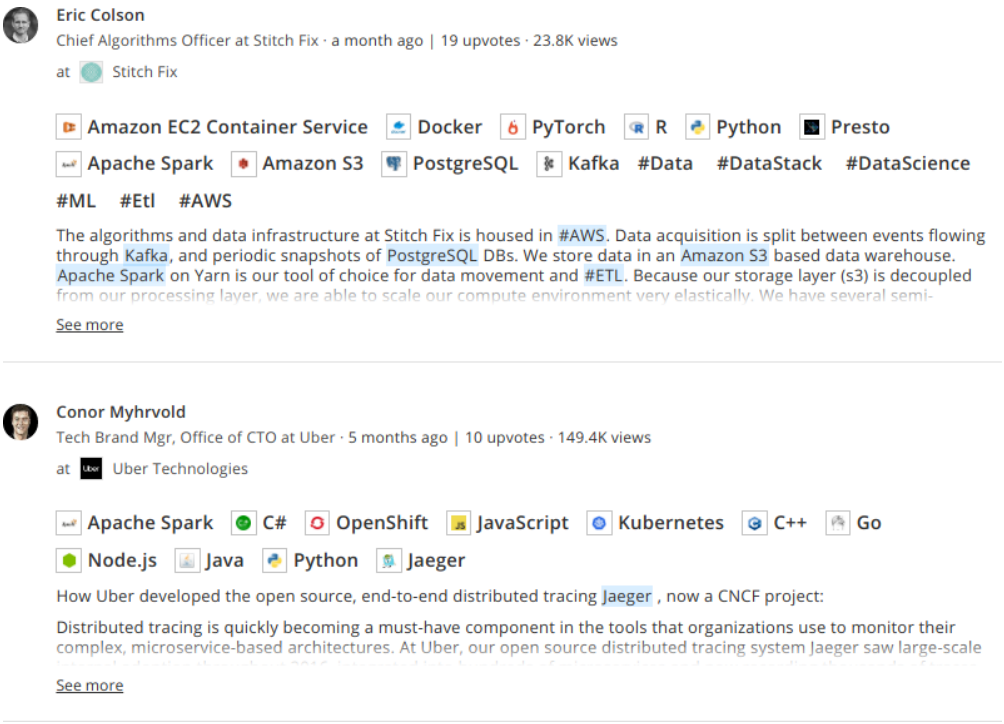

Figure 2.27: Apache Spark Reviews & Stack decisions
Source: https://stackshare.io/spark

In addition to the information about each tool the comparisons (stackups) page adds support for tool cons, and questions to ask the community for advice for that specific comparison. Comparisons between tools can be made irrespective of their nature, for instance, *Python vs Boostrap*,

[52]https://stackshare.io/submit
[53]https://stackshare.io/claim-service/amazon-ec2

so it is up to the user to choose similar tools. Unfortunately as what happens in slant, it is not easy to contrast tool features visually because they are encompassed in **Why do developers choose Foo?** free-form one liners which make no distinction between opinions and actual implemented features.

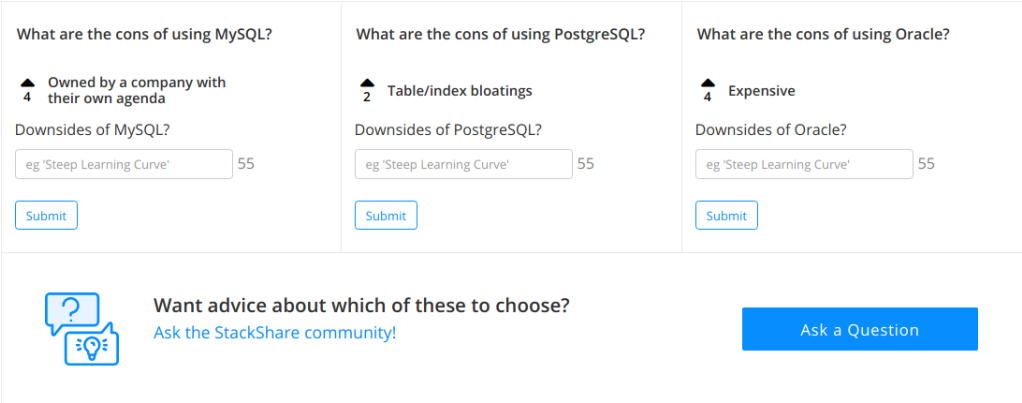

Figure 2.28: MySQL vs PostgreSQL vs Oracle
Source: https://stackshare.io/stackups/mysql-vs-oracle-vs-postgresql

One of the most interesting functionalities is the ability for users to share their personal stacks, and work stacks by adding a company email. Each tool of the stack is segmented according to its main category and possibly justified by a decision from a team member.

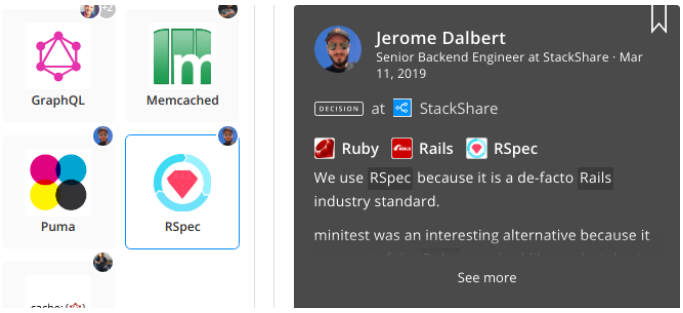

Figure 2.29: StackShare Stack
Source: https://stackshare.io/stackshare/stackshare

2.4.2 Feature-Comparison Websites

Websites such as the Open Source Time Series DB Comparison[54] (see Figure 2.30), Knowledge Base of Relational and NoSQL Database Management Systems[55], Ultimate Time Series DB Com-

[54] https://docs.google.com/spreadsheets/d/1sMQe9oOKhMhIVw9WmuCEWdPtAoccJ4a-IuZv4fXDHxM/edit
[55] https://db-engines.com/en/systems

parison[56], Relational Database Management Systems Comparison[57], among others provide an exhaustive list of features but with duplication of information between them often with no information about which component version has the indicated functionality. Which is probably a consequence of software not being described feature wise in software repositories following a common domain terminology. Also because features are separate from software justifications component consumers have to know the drawbacks and advantages that certain features pose on their situations.

	A	B	C	D	E
1	read this blog before commenting	**DalmatinerDB**	**InfluxDB**	**Prometheus**	**Atlas**
21	**Ingress**	tcp (binary protocol), OpenTSDB (text), Graphite (text), Prometheus (text), Metrics 2.0 (text), InfluxDB (http)	InfluxDB (http), InfluxDB (udp), OpenTSDB (text), OpenTSDB (http), Graphite (text) and a few others	scraping (text, protobuff)	http
22	**Egress**	http, tcp raw binary (no dql)	http	http	http
23	**Query Language Functionality**	3/5	4/5	5/5	
24	**Query Language Usability**	4/5	5/5	4/5	1/5
25	**Dynamic Cluster Management**	Yes	-	-	
26	**Continuous Query / Rollups / Downsampli**	No	Yes	Yes	No
27	**Security and ACL's**	No	Yes	No	No
28	**Data TTL (retention policy)**	per bucket	per database (retention policy)	global	
29	**Commercial Support**	Yes	Yes	Yes	No
30	**Commercial Support Link**	https://project-fifo.net/#support	https://portal.influxdata.com/	http://www.robustperception.io/	-
31	**Community Size**	small	large	large	small
32	**License**	MIT	MIT	Apache 2	Apache 2
33	**Latest Version**	v0.2.1	v1.3.5	v2.0.0-beta.2	v1.5
34	**Maturity**	Early adopter	Stable	Stable	Stable
35	**Pro's**	Reasonable to operate and scale (built on well known mature technologies). Clustering and fault tolerance is a first class citizen. High performance reads and writes and expressive query language. A steadily growing number of functions. The best option if you want TSDB features and need to scale to high reads and writes in future.	Easy to operate, highly customisable, lots of cool features and good performance on a single node. Documentation is well polished. The best option if you only want TSDB features and don't need to horizontally scale.	Easy to operate, good data model, high performance, lots of query functionality. The best option if you want an all in one monitoring system with a few weeks of history. Fits in really well with the container ecosystem.	Very fast and highly scalable if you have lots of money for ram. Probably good if you are Netflix or Facebook (who created Gorilla which looks similar but isn't open sourced yet).
36	**Con's**	Works best with locally attached storage (for ZFS). Erlang may make it harder for people to dig into the code and troubleshoot or submit changes. Not much community activity and the docs are all over the place. Client library support is limited, however, a metrics proxy supporting common protocols can be used.	History of bugs and breaking changes although seems better recently. Clustering no longer developed in open source edition which would make it terribly difficult to scale.	More than just a TSDB and not designed to be used as a backend. Designed to use alternative backend for long term storage which is a pro for a resilient monitoring system but a con for time series database comparison.	In memory queries mean atlas is only good for near real time (a few weeks of data). Query language is a bit weird. More of the Netflix software around the edges of Atlas needs to be released to make it work well.
37					

Figure 2.30: Open Source Time Series DB Comparison Google Sheets
Source: https://docs.google.com/spreadsheets/d/1sMQe9oOKhMhIVw9WmuCEWdPtAoccJ4a-IuZv4fXDHxM/edit

[56] https://tsdbbench.github.io/Ultimate-TSDB-Comparison/
[57] https://en.wikipedia.org/wiki/Comparison_of_relational_database_management_systems

Chapter 3

Problem Statement

Early in this chapter we present the major pain points in component selection, in Section 3.1. Afterwards, we propose a framework, in Section 3.2, described by: an approach relating it to current issues and possible alternatives, in Section 3.2.1; following a set of high level characteristics, in Section 3.2.2; with some assumptions, in Section 3.2.3; and an evaluation strategy, in Section 3.2.4.

3.1 Current Issues

As mentioned in Chapter 1 there are numerous ways to search for components and their characteristics but the information follows different syntaxes, semantics, and in most cases does not describe the architectural choices and quality attributes for different scenarios. The major issues are summed up bellow:

1. **Architectural Decision Knowledge of software projects is not re-purposed for Component Selection**: the process of component selection involves grasping the qualities of components therefore design rationale could possibly be reused for consumer reasoning.

2. **Architecture Decision Rationale is in most cases implicit, unstructured or not recorded through tools**: architecture knowledge management systems require too much information capture resulting in big chunks of developer time being allocated to the maintenance of documentation.

3. **Component Selection features are scattered and multiform**: there are too many syntax and semantic differences in online information about components which makes its search and comparison a hard task.

4. **Project documentation lives apart from code repositories**: in most cases documentation is not in sync with projects since it is maintained in external systems.

5. **The documentation of Components is too free-form**: there is no easy way to gather all the important features that characterise a piece of software without digging into its documentation.

3.2 Proposal

As evidenced by the issues above, software components projects rarely provide structured information about their features. As a consequence, component selection is an exploratory endeavour relying in most cases on manual search and comparison of related software. One possible way to try to solve this issue would be to reduce the amount of information required by most AKM data models and focus it on the description of the set of features that make up a software component. Preferably, the descriptions would be produced by the developers of the components themselves in order to diminish syncing with external systems such as wikis (QuABaseBD), other web places (Comparison Websites) and AKMSs (see Table E.6) while at the same time promoting its quality and reducing the need for repeated work on feature description.

By structured information we mean information that fits into a data model that facilitates the description of a software component following a set of features that a developer sees as an important characteristic of their solution. The characteristics that developers choose to describe their software will vary but they must be structured and comparable in order for that information to usable by component selection processes.

In sum the goal of this dissertation is to build a conceptual framework that helps with architectural decision making in particular the selection of components by making use of structured knowledge.

3.2.1 Approach

Because architectural knowledge from AKMSs is scarce and hard to re-purpose for component selection we focus on collecting features from Software Components present in repositories and exposing them through a service. In a sense we are building a knowledge base of software features capable of differentiating concerns — e.g. Big Data Technologies, quality-attributes, and so on — that can assist component selection. Thereby addressing to an extent the issues above.

Issues 1 and 2 in Section 3.1, have to do with AKMS and as seen in Chapter 2 almost no projects make use of it. So trying to re-purpose Decision Knowledge to aid Component Selection when the former is almost non-existent is for now a delusion. Solving these these two issues probably involves investigating a new information flexible framework for the capture of Architecture Knowledge, testing it and then having it adopted in projects. Only then would it be possible to attempt to re-purpose some of that knowledge for Component Selection. These two issues are related to our proposal only because features are structured information and part of Architectural Knowledge.

Issue 3 in Section 3.1, is only considered in part because the features of Components are assigned only by repository contributors. Unfortunately, by using our approach other Web Places

can not indicate additional features for Components. The construction of a website to centralise the description of Components in feature terms is in our opinion not a good solution since many other similar Websites already exist, see Section 2.4 in Chapter 2. To solve this problem in its entirety technology from the Semantic Web could be looked at. Software Repositories would need to be identified through an URI, and the RDF statements from the contributors would have to be exposed in an RDF dataset URI. Other websites to do with Component Selection could then use RDF and an ontology built by us or not to append to an interconnected sea of information where different claims about Components would coexist. Alas, Semantic Web and Linked Data technologies are still not really that well known, and occasionally looked at with disbelief so we opted for a less grandiose approach.

Issue 4 in Section 3.1, deals with the fact that software documentation tends to live far from code repositories which is obviously not ideal since developers spend most of their time working on code related activities. External documentation has the obvious drawbacks of being difficult to track and keep updated as the project grows which is more likely than not one of the main reasons why so few documentation exists. The addition of software features as a form of documentation to the repositories of projects while being far from a full solution does offer benefits to Component Consumers. As a matter of fact, a collection of the main features of a Component can act as a way to sum up major parts of documentation, Issue 5 in Section 3.1. However, a more sound approach to the summation of documentation could be attempted by using technology from the Semantic Web namely RDFa to embed RDF data directly inside HTML documents or even inside code comments. Unfortunately as stated above Semantic Web and Linked Data approaches are yet to ingrain in the mindset of developers.

3.2.2 Desiderata

Following from the approach and the set of issues described above, we conceived a list of characteristics that implementations should honour. There may be multiple solutions that follow the desiderata, herein described, that can present good solutions to the approach. Ours is just one interpretation of such requirements that we will have to analyse.

- **Capture and Grouping**, there should be a mechanism that enables the capture of structured information deemed important by developers for consumers about their software components in repositories. Because software may be described according to different concerns it should be possible to group related features.

- **Validation**, in order to contrast software and promote consistent definitions according to the concerns of a group of features there should be some kind of validation that makes sure that an instance of a concern is well defined. It guarantees that different repositories can be compared among each other as long as they respect the concern's vocabulary.

- **Reuse**, information about features and groups should be reusable and there should be a way to reduce information duplication — e.g. when the same feature appears in different groups, when mapping a set of features to a concern via a group, etc.
- **Versioning** is particularly important since for the same piece of software there may be releases with different features. The same is also true for concerns for the simple reason that they might evolve as a domain is better understood.
- **Search and Comparison**, the captured features should be usable by external entities. For instance, comparison websites could then leverage this information to augment their software descriptions possibly building new visualisations — e.g. feature clustering, iterative comparison through tables, software search, etc.

3.2.3 Assumptions

We will focus our efforts on the construction of a simple system following the principles stated in the desiderata, reusing existing technology whenever possible. Additionally, no dataset for features will be devised since it requires expert knowledge and industry practice using the concerned component types. Consequently, we will populate the system with data from QuABaseBD and other reputable sources (Kleppmann, 2017) which will be placed in custom repositories to help validate the approach.

3.2.4 Evaluation

We judge whether the desiderata, in Section 3.2.2, is achieved through our implementation, in Section 4.4, in order to ascertain the merits of our approach. This involves discussing the related implementation for each key principle, contrasting it with approaches in the literate review, and exemplifying its use through a prototype front-end application.

Chapter 4

Implementation

In this chapter we describe the many parts that constitute the developed system and state how the application can be used. The implementation is grounded on the principles established by the desiderata and aims to ease the capture and sharing of feature information. Section 4.1 summarises the implementation. Section 4.2 describes the architecture and the tools that where used to build the system. Section 4.3 presents the data model used to encode component features and schemas. Section 4.4 lists the features provided by our implementation. Section 4.5 describes the flow of use for the two roles considered: component producer and component consumer. Section 4.6 reports how the solution can be built.

4.1 Overview

Our implementation leverages the GitHub version control hosting service to capture information about software projects. Even though feature information is only captured from GitHub repositories the idea can be extended to other code hubs — e.g. by using their Application Programming Interface (API), git hooks, scripts, etc. In fact, adding support for the extraction of features from repositories may boost feature knowledge capture and curation since one of the reasons developers do not document them in a structured way is due to the little benefit it brings in the short term and the maintenance cost of keeping information synchronised with external systems. Moreover, we opted for GitHub because they are the most popular code hub and have a well documented API[1]. The GitHub API supplies a means to automate and improve the work-flow of repositories using GitHub Apps[2]. So we created a GitHub App called featurewise with a set of permissions that can be installed in repositories whose goal is to deliver events to our platform. When we receive those events we process them and if deemed appropriate we fetch a file in the root of the respective

[1] https://developer.github.com/
[2] https://developer.github.com/apps/

repository called *featurewise.{json,yaml}* whose contents describe the software's features. Lastly, our platform's API supplies component information documents to interested parties.

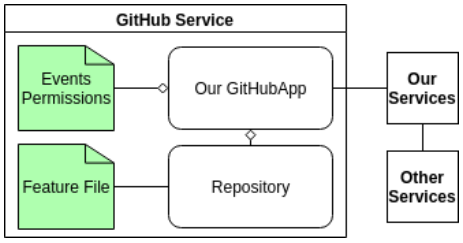

Figure 4.1: Implementation Overview

4.2 Architecture and Tools

Our implementation captures events from GitHub projects when they install our GitHub App whose name is Featurewise. GitHub Apps are the recommended way to integrate with GitHub and work by sending certain subscribed events configurable through GitHub's website and their payloads to a webhook URL. In our case we are only interested in the check_suite[3], check_run[4], release[5], and repository[6] events and their actions. As for permissions we only require read access to repository metadata and code, and write and read access to checks.

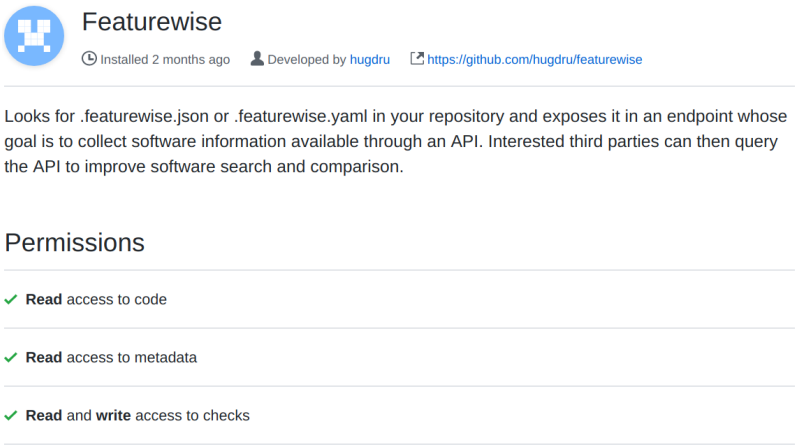

Figure 4.2: Featurewise GitHub App

A check_suite is a collection of check_runs for a specific commit. And its requested action event triggers when a new commit is pushed to a repository. From that point on check_runs can be created to analyse a commit. Moreover, the release event and its actions are used to capture the releases of components and their respective features and to synchronise our database on release

[3] https://developer.github.com/v3/activity/events/types/#checksuiteevent
[4] https://developer.github.com/v3/activity/events/types/#checkrunevent
[5] https://developer.github.com/v3/activity/events/types/#releaseevent
[6] https://developer.github.com/v3/activity/events/types/#repositoryevent

delete, edition, and so on. Likewise, repository events serve to synchronise on repository meta-data changes.

When the web-hooks arrive we load balance them between n Github Server nodes whose purpose is to consume and act on the events discussed above. If the event contains information about features or repository meta-data that is suitable for archival we store it in PostgresSQL[7]. To avoid creating additional check_runs for branches whose features should not be published but only checked we query Redis[8] for feature file hashes that have already been processed. Our backend then utilizes the PostgresSQL datastore to provide a REST API for external users to retrieve feature files, schemas and related meta-data so that they can augment software comparisons. Check the deployment diagram bellow, Figure 4.3, for a visual representation of the described above.

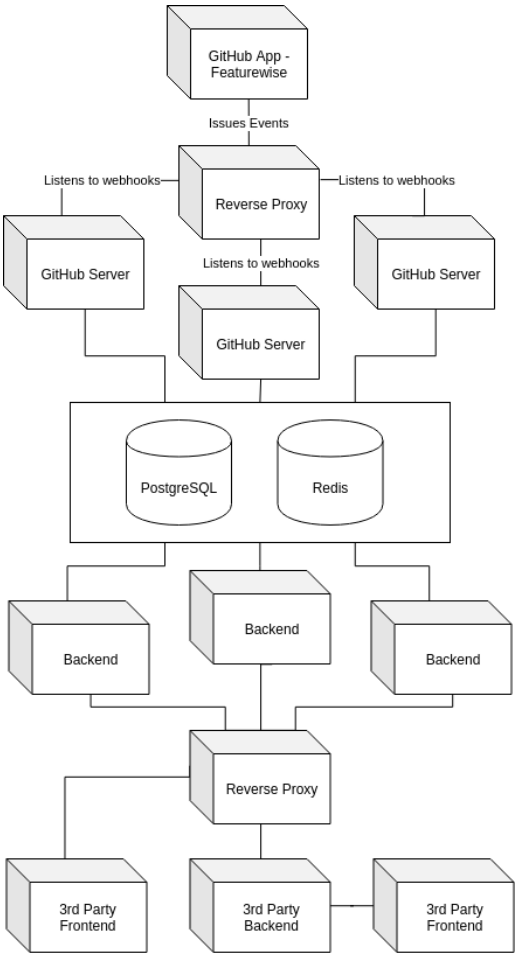

Figure 4.3: Deployment Diagram

The whole solution is developed using the Javascript programming language so that we can share code between the Github Server and the Backend. The former is built on top of a framework

[7] https://www.postgresql.org/
[8] https://redis.io/

for building Github Apps, Probot[9]. And the later uses a low overhead web framework inspired by Express.js[10] and Hapi.js[11], fastify[12]. For group validation and SQL manipulation we leverage respectively Ajv[13] JSON Schema validator and Objection.js[14]. The nodes and the datastores shown in the deployment diagram above are built using docker[15] images and run using the docker-compose[16] multi-container tool. Finally, Traefik[17] acts as the reverse proxy shielding and load balancing requests originating from outside our platform. Even though in the above diagram it shows two reverse proxies in actuality it is only one with two distinct locations. One for the Github App and another for the Backend.

4.3 Data Model

Feature files, and groups are represented in our data model as Featurewise, and Domain. A Featurewise row represents a repository feature file which originates from either a Release or a Branch Commit. Groups of related features are stored in Domain rows and may follow a validation schema so that related software can be located and compared. A schema can originate from a repository or from an external URL. In the case of the former repository meta-data is used to frame it, much like what happens with feature files because they are always located in repositories, see Figure 4.4 and its implementation in Listing 23 in appendix G.

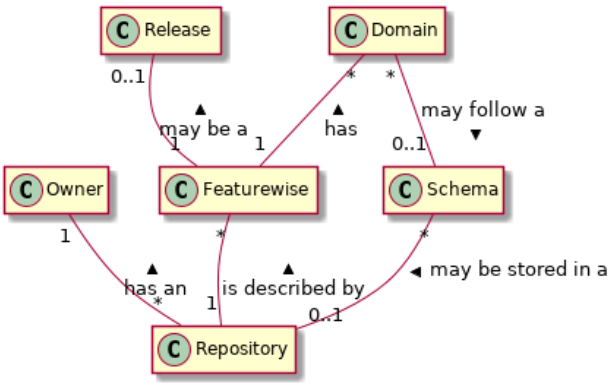

Figure 4.4: Class Diagram

[9] https://github.com/probot/probot
[10] https://github.com/expressjs/express
[11] https://github.com/hapijs/hapi
[12] https://github.com/fastify/fastify
[13] https://github.com/epoberezkin/ajv
[14] https://vincit.github.io/objection.js/
[15] https://www.docker.com/
[16] https://docs.docker.com/compose/
[17] https://traefik.io/

4.4 Features

Herein lies the features that we implemented following the desiderata, present in Section 3.2.2. Each feature has a brief description of the reasoning behind it in order to best convey its meaning and objective.

4.4.1 Feature 1 — encode features

Encoding of the features of a component in a file stored in the root of a repository — e.g. *.featurewise.json*. The file is readable by either humans or machines in a well adopted format, JSON. Support for processing of this file is added by installing a GitHub App called Featurewise which should work in a similar fashion to travis[18], see Figure 4.5 — i.e. a server under our control captures repository events issued by GitHub and acts on them by collecting and processing feature files.

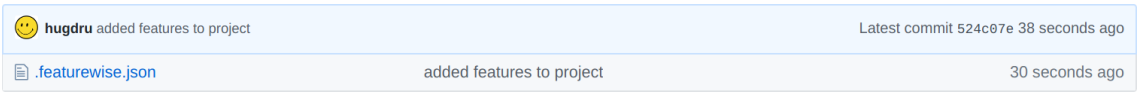

Figure 4.5: Feature File

4.4.2 Feature 2 — encode features per domain

Components can belong to multiple categories of Software as is the case of redis[19]. It is an in-memory datastore that can be used as a database, cache, or message broker. As a result the *domain* concept exists to separate different sets of features, see Listing 13 — e.g. cache, message broker, etc. However, it can be the case that a subset of features match in different domains — e.g. replication. To fix this issue either domains have to be more specific so as to avoid repetitions — e.g. a domain just for replication which would reduce the benefit of whole comparisons — or for there to be a mechanism to reduce duplication and link equal features together, see features 4.4.3, 4.4.4, and 4.4.9.

```
1 {
2     "domains": [{}, {}]
3 }
```

Listing 13: Multiple Domains

4.4.3 Feature 3 — support multiple encoding formats

Apart from JSON there is support for YAML since it is more appropriate for configuration files. Fortunately most of YAML can be translated to JSON. As a matter of fact YAML is crucial for

[18] https://travis-ci.com/
[19] https://redis.io/

diminishing information duplication as it has support for anchors(&) and references(*), see Listing 14 and 15.

```yaml
---
- Scalability: &id1
    Scalable Distribution Architecture: replicate complete database only
    Request Load Balancing: fixed connections to a request coordinator

- Scalability2: *id1

- Scalability3: *id1
    <<: *id1
    Request Load Balancing: uses HTTP-based load balancers
    Granularity of Write Locks: no locks - conflicts allowed
```

Listing 14: Example of anchors(&) and references(*) in YAML

```json
[
  {
    "Scalability": {
      "Scalable Distribution Architecture": "replicate complete database only",
      "Request Load Balancing": "fixed connections to a request coordinator"
    }
  },
  {
    "Scalability2": {
      "Scalable Distribution Architecture": "replicate complete database only",
      "Request Load Balancing": "fixed connections to a request coordinator"
    }
  },
  {
    "Scalability3": {
      "Scalable Distribution Architecture": "replicate complete database only",
      "Request Load Balancing": "uses HTTP-based load balancers",
      "Granularity of Write Locks": "no locks - conflicts allowed"
    }
  }
]
```

Listing 15: Listing 14 converted to JSON

4.4.4 Feature 4 — support for optional feature schemas

Support for schemas is a very important feature since it makes data more structured and homogeneous helping greatly with comparisons. Component domains can be identified via a schema URL wherein possible values for features are described, see Listing 16 and 17.

```
{
  "domains": [
    {
      "schema": {
        "url": "https://github.com/hugdru/some-repo/blob/master/.some-schema.json"
      },
      "data": {
        "Scalability": {
          "Scalable Distribution Architecture": "horizontal partitioning and replication",
          "Scaling Out - Adding Data Storage Capacity": "automatic data rebalancing"
        }
      }
    }
  ]
}
```

Listing 16: Features file with just one domain

```
{
  "title": "Big Data Architectures and Technologies",
  "description": "Classifies Big Data Technologies according to a set of features",
  "type": "object",
  "properties": {
    "Scalability": {
      "title": "Scalability",
      "description": "Describes how a system behaves when there is increased load or resource demand",
      "type": "object",
      "properties": {
        "Scalable Distribution Architecture": {
          "type": "string",
          "enum": [
            "replicate complete database only",
            "horizontal partitioning of database",
            "horizontal partitioning and replication"
          ]
        },
        "Scaling Data Storage Capacity": {
          "type": "string",
          "enum": [
            "automatic data rebalancing",
            "manual data rebalancing",
            "N/A - single server only"
          ]
        }
      },
      "additionalProperties": false
    }
  },
  "additionalProperties": false
}
```

Listing 17: Features Schema file using JSON Schema

Schemas are optional so as to not burden users with the declaration of possible values when domains are not yet fully understood. A best effort comparison involving simple string comparisons would have to be done in such cases when contrasting software. It may happen that semantically equal concepts will not be contrasted because of sleight differences in syntax. However as understanding grows around a domain, repository contributors can come together and define a schema that better contrasts their features with those of competitors extolling their virtues. A schemaless domain is defined by not specifying the *schema* object. In addition, the schema is also definable in YAML so that users can pick a common format for both features and schemas declaration.

4.4.5 Feature 5 — feature and schema versioning

The framework expects software features and schemas to evolve when respectively new releases or pushes are made, and feature vocabulary is updated. The features of software components are versioned from the get go since they are stored in GitHub repositories — e.g. commit id, timestamp, branch, release, tag, etc. The same is true for Schemas but only if they are stored in GitHub. When they are defined elsewhere the only information we have is the URL, and the file's contents. So we attach an id to it which gets incremented only if its contents change. To conclude, in both cases we look carefully into the feature and respective schema files per push per branch to check for actual changes so as not to populate the Featurewise table with duplicated information. Therefore we decode, normalise and compare features and schemas to their last related inserted value in all cases except for releases which always get stored if its associated commit contains a featurewise file. The comparison involves contrasting the hash of the normalised files using *SHA-256*[20].

4.4.6 Feature 6 — ignorable branches, releases, and branch publishes

Different versions of software are normally developed under separate *branches* with specific points of history *tagged* for the purpose of releases — e.g. see `https://github.com/apache/spark`. In spite of that some branches and tags might not hold any special meaning or be desirable for schema validation using the GitHub Checks API, see Feature 4.4.8, and publishing — e.g. branch *gh-pages* . For that reason it is possible to opt out or in for branches, and enable or disable support for releases and as a consequence their tags. In addition, publishes for branches can be disabled so that only checking is done, see Listing 18.

```
# Defaults to allow every branch except gh-pages
branches:
  only: master # Takes precedence over except
  except: master # Disregarded in this case since only is present
  publish: True # Defaults to False
# Defaults to True
releases: False
```

Listing 18: Ignoring branches, releases and publishes

[20] `https://en.wikipedia.org/wiki/SHA-2`

4.4.7 Feature 7 — capture push and release information

Each push or release event contains repository and commit metadata — e.g. commit id, timestamp, branch, release title and description, tag, etc — that is stored along side feature files for framing and informational purposes, see https://developer.github.com/v3/activity/events/types/#checksuiteevent for an example of the *check_suite* event payload and https://developer.github.com/v3/activity/events/types/#releaseevent for *release* events.

4.4.8 Feature 8 — schema validation using the github checks API

GitHub Apps[21] help improve the workflow of projects by automating different types of tasks. In our case we validate the correctness of domains when they are subject to a schema and provide a message if the feature file was successfully checked and/or published. For that reason we consume events and call APIs to do with check_suites[22,23] and checks[24,25]. See Figure 4.6 for an example of a validated but not published domain since its featurewise.json has that disabled, and Figure 4.7 for a job that failed.

Job Succeeded

The Job has **succeeded**.

DETAILS

domains[0] succeeded

The features were **validated** according to the schema.

Domain

schema: https://github.com/hugdru/github_integration/blob/master/.domain-schema.yaml

▼ Show contents

```
{
  "Scalability": {
    "Scale Out Architecture": "replicate complete database only",
    "Client Request Load Balancing": "fixed connections to a request coordinator",
    "Scaling Data Storage Capacity": "manual data rebalancing",
    "Data Object Based Locks on Writes": "locks on tables/collections",
    "Scalable Request Processing Architecture": "fully distributed - any node acts as a coordinator"
  }
}
```

Schema

title: Big Data Architectures and Technologies
description: Classifies Big Data Technologies according to a set of features

▶ Show contents

Figure 4.6: Example of a domain in a feature file that was validated

[21] https://developer.github.com/apps/
[22] https://developer.github.com/v3/activity/events/types/#checksuiteevent
[23] https://developer.github.com/v3/checks/suites/
[24] https://developer.github.com/v3/activity/events/types/#checkrunevent
[25] https://developer.github.com/v3/checks/runs/

Job Failed

The Job has **failed**.

DETAILS

Failed getting schema or validating features for at least one domain

domains[0] failed

The features failed to **validate** according to the schema.

Features do not obey the schema

Error 0

enum .Scalability['Scalable Request Processing Architecture']

Should be equal to one of the allowed values

```
{
"allowedValues": [
  "none",
  "not scalable (bottleneck)",
  "fully distributed - any node acts as a coordinator",
  "centralized coordinator, but can be replicated",
  "based on an external load balancer"
]
}
```

Check schemaPath **#/properties/Scalability/properties/Scalable%20Request%20Processing%20Architecture/enum** for more information

Domain

schema: https://github.com/hugdru/github_integration/blob/master/.domain-schema.yaml

▶ Show contents

Figure 4.7: Example of a domain in a feature file that failed validation

4.4.9 Feature 9 — re-use schemas to reduce heterogeneity

JSON schema can reduce duplication inside a document by using keywords to locate subschemas. As a convention they are defined in a top level object called *definitions*[26] and referenced through the *$ref* keyword which basically outputs the referenced value, see Listing 19. YAML descriptions can use their own convention for pointers or be directly translated from JSON.

```
1  {
2    "definitions": {
3      "details": {
4        "type": "object",
5        "properties": {
6          "description": { "type": "string" },
7          "pitfalls":{ "type": "array", "items": { "type": "string" } }
8        },
9        "required": ["description", "pitfalls"]
10     }
11   },
12
13   "title": "Big Data Architectures and Technologies",
14   "description": "Classifies Big Data Technologies according to a set of features",
15   "type": "object",
16   "properties": {
17     "Scalability": {
```

[26] https://json-schema.org/understanding-json-schema/structuring.html

```
18        "details": { "$ref": "#/definitions/details" },
19        "title": "Scalability",
20        "description": "Describes how a system behaves when there is increased load or resource demand",
21        "type": "object",
22        "properties": {
23          "Scaling Data Storage Capacity": {
24            "type": "string",
25            "enum": [
26              "automatic data rebalancing",
27              "manual data rebalancing",
28              "N/A - single server only"
29            ]
30          }
31        },
32        "additionalProperties": false
33      }
34    },
35    "additionalProperties": false
36  }
```

Listing 19: Definition reuse in JSON Schema

The *$ref* keyword expects a URI path to a subschema. So it may happen that the unit of reuse is a resource located elsewhere — e.g. a URL such as *"$ref": "http://mycontrolled.domain/otherSchema.json#definitions/details"* or *"$ref": "https://github.com/owner/repo/blob/master/otherSchema.json#definitions/details"* or simply *"$ref": "https://github.com/owner/repo/blob/master/otherSchema.json"*. JSON schema validators must validate documents with external subschemas but their downloads are not expected to be handled by them. As a consequence we implemented recursive subschemas fetching, and linkage using the addSchema function from Ajv[27]. In addition to the $ref & definitions approach to reuse it is also possible to utilise $id which is a way to identify one schema without navigating a JSON tree, resembling YAML anchors and references, see Listing 20.

```
1   {
2     "definitions": {
3       "details": {
4         "$id": "#details",
5         "type": "object",
6         "properties": {
7           "description": { "type": "string" },
8           "pitfalls":{ "type": "array", "items": { "type": "string" } }
9         },
10        "required": ["description", "pitfalls"]
11      }
12    },
13
14    "title": "Big Data Architectures and Technologies",
15    "description": "Classifies Big Data Technologies according to a set of features",
16    "type": "object",
17    "properties": {
18      "Scalability": {
```

[27] https://github.com/epoberezkin/ajv

```
19          "details": { "$ref": "#details" },
20        }
21      }
22    }
```

Listing 20: Definition reuse in JSON Schema using *ids*

4.4.10 Feature 10 — provide a public API

Our REST API provides a set of GET routes for the retrieval of software features with support for optional query parameters. One query parameter that exists for all routes is *eager* which allows for the retrieval of additional related table information. For instance */featurewise?eager=[release, domains.schema.repository.owner, repository.owner]* returns a JSON whose content is the result of joining all those tables.

- */domain & /domain/:id* — routes for the retrieval of domains.
- */featurewise & /featurewise/:id & /featurewise/latest* — routes for the retrieval of feature files.
- */owner & /owner/:id* — routes for the retrieval of owners.
- */release & /release/:id* — routes for the retrieval of releases.
- */repository & /repository/:id* — routes for the retrieval of repositories.
- */schema & /schema/:id & /schema/latest* — routes for the retrieval of schemas.

Software comparison websites and others can then use the information supplied by our API to enrich their comparisons, descriptions, component justifications, and others.

4.5 Using the solution

In this section we describe how users can utilise our implementation following the two different use cases. As a developer of a piece of software, a producer, or as a user looking for feature information about components, a consumer.

4.5.1 Producer

Firstly the Featurewise GitHub App must be installed so that our service can receive GitHub Events. To do so the user must navigate to the Apps URL, `https://github.com/apps/featurewise`, see Figure 4.8, and install it on the desired repositories, see Figure 4.9.

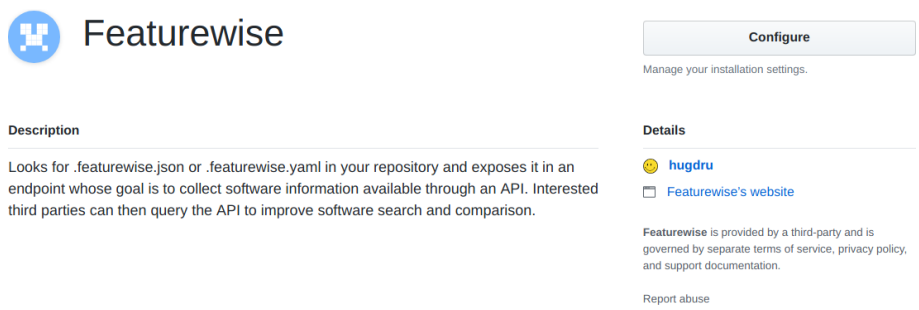

Figure 4.8: Featurewise installation page

After the installation branch commits and releases issue events that are captured by our service triggering the download of a featurewise.json or a featurewise.yaml in the root of the repository. Their settings (i.e. branches: & releases:) along with the version control meta-data allow us to frame and process the features accordingly — i.e. ignore branches and/or releases, skip processing, validate or validate and publish, and so on. Unfortunately the construction of the featurewise file is a manual text-based task. However because we use schemas it should be easy to create a form per domain with autocompletion. This functionally could even be added to a front-end app which uses GitHub OAuth to import and export featurewise files from user repositories.

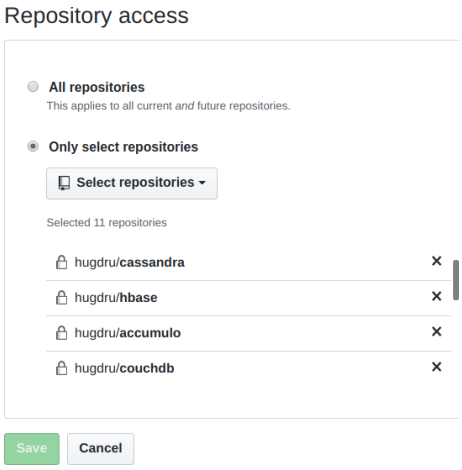

Figure 4.9: Featurewise configuration page

4.5.2 Consumer

The information provided by our service can be queried directly by end-users through the REST API, see Feature 4.4.10, as a way to build component feature catalogues for use in Component Selection Processes and MCDM systems. Additionally, other services, such as Software Comparison Websites, may build on top of our API to supply current and structured visualisations.

4.6 Building the Solution

To assemble our service as described in Figure 4.3 we use docker and docker-compose. There exists three different docker-compose files, one for development in Listing 26, another for production in Listing 24, and lastly one for traefik in Listing 25. Each follows different environment variable settings which are set automatically by running a helper script located in the root of the project called, *./run.sh* see Listing 21. Traefik does not belong to the production docker-compose.yml even though it is only meant for production because it is possible for it to route requests to different composes or even to cluster technologies such as docker swarm as long as they belong to the same network which the ./run.sh creates when ran with the *traefik d* option.

```
Usage: ./run.sh env (p package)|d) call
    env is a file in env/ without the extension where
        secrets and environment variables are stored
    p cds to a package in packages/ and runs a command
    d cds to an env folder in docker/ and runs a command
    call represents a command and arguments to run

Examples:
    ./run.sh dev d docker-compose up
    ./run.sh dev p backend yarn dev
    ./run.sh dev p github-app yarn dev
    ./run.sh dev p frontend yarn start
    ./run.sh traefik d docker-compose up
    ./run.sh prod d docker-compose up
    ./run.sh prod p frontend yarn start
```

Listing 21: Run options for run.sh

To avoid having to create a server online in order to receive the GitHub App Events we use smee[28] for development purposes. It is a small service that proxies payloads from a webhook source to a local machine, see Figure 4.10.

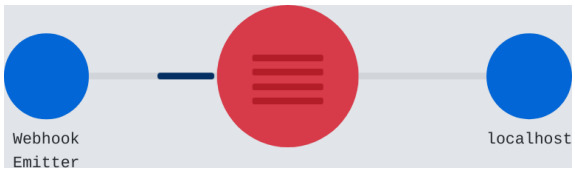

Figure 4.10: Smee in action

[28] https://smee.io/

To run in production mode locally traefik must receive host information so that it can redirect the requests to the appropriate containers. It is possible to emulate this by adding entries to the Static table lookup for hostnames, in file */etc/hosts*, see Listing 22.

```
127.0.0.1 featurewise.com
127.0.0.1 api.featurewise.com
127.0.0.1 www.featurewise.com
127.0.0.1 github.featurewise.com
```

Listing 22: Table lookup for hostnames

Chapter 5

Evaluation

In this chapter we describe the methodology, in Section 5.1, used to analyse the implementation of each desiderata topic: *Capture and Grouping*, in Section 5.2; *Validation*, in Section 5.3; *Reuse*, in Section 5.4; and, *Versioning*, in Section 5.5.

5.1 Methodology

We judge whether the desiderata, introduced in Section 3.2.2, is achieved through our implementation, described in Section 4.4, in order to ascertain the merits of our approach. This involves evaluating the implementation for each desideratum following a sequence of steps:

- Summarising each desideratum
- Briefly describing the implementation
- Discussing how the implementation relates to the principle
- Contrasting the implementation with approaches in the literate review
- Exemplifying its use through a prototype front-end application

To evaluate our system we built a dataset consisting of 10 repositories each with a feature file using knowledge extracted from QuABaseBD. The following repositories where built: hugdru/accumulo, hugdru/cassandra, hugdru/couchdb, hugdru/hbase, hugdru/mongodb, hugdru/neo4j, hugdru/redis, hugdru/riak, hugdru/voltdb, and lastly hugdru/schemas. Our GitHub App was installed in all repositories and their featurewise files captured by our service on commit push, and release publish. A frontend written in React[1], a library for building Single-Page Application user interfaces declaratively, and Typescript[2], a statically typed super-set of Javascript, queries our service and presents views for component and schema search, and comparison.

[1] https://reactjs.org/
[2] https://www.typescriptlang.org/

5.2 Capture and Grouping

Part of the desiderata is for there to be a mechanism that enables the capture of structured knowledge and scoping of different feature concerns into groups. In our implementation, software features are captured by downloading feature files after handling release and commit events issued by GitHub. Depending on a set of rules they might or might not be stored in our service. For instance, the metadata and featurewise file of releases is always stored in our service as long as the associated tag/commit contains a features file. But commit featurewise files only get saved if the current featurewise file is different from the last one inserted considering the current branch. In essence, our implementation establishes a means for the creation of a queryable curated knowledge base built straight from code repositories and it relates to the literature review in the following way:

- In contrast to QuABaseBD we relax the model disregarding concepts such as scenarios and tactics but promote contributions whilst handling structured information. Additionally we extend the model so as to allow for different domains to co-exist when describing the same piece of software — e.g. one domain to describe Big Data Features, another to describe its key quality attributes, and so on.

- The solution resembles Feature-Comparison Websites as discussed in the literate review with the exception that features are scoped to different concerns through domains following strict validation rules — i.e. feature groups are implemented using the concept of domains and schemas. For instance it is possible to characterise Redis[3] both as a database, cache, and message broker in the same knowledge unit instead of relying on different websites that focus on each of those specific domains.

- Furthermore, our framework much better describes component functionality when compared to Multi-Faceted Comparison Websites that leverage lists in the form of free-text to contrast software. Alas, those lists commonly mix features and opinions and do not group them by concern.

In Figure 5.1 we can see a list of repositories that have at least one associated featurewise file stored in our service.

[3] https://redis.io/

featurewise Home Repositories Schemas Compare

repository	description	homepage
hugdru/mongodb	MongoDB is a general purpose, document-based, distributed database built for modern application developers and for the cloud era	https://www.mongodb.com/
hugdru/voltdb	VoltDB is a in-memory database for modern applications requring an unprecedented combination of data scale, volume, and accuracy	https://www.voltdb.com/
hugdru/riak	Riak provides NoSQL database solutions, enabling distributed systems to scale large amounts of unstructured data	https://riak.com/
hugdru/redis	Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache and message broker	https://redis.io/
hugdru/neo4j	Is a graph ACID-compliant transactional database with native graph storage and processing	https://neo4j.com/
hugdru/hbase	Apache HBase is the Hadoop database, a distributed, scalable, big data store	https://hbase.apache.org/
hugdru/couchdb	Seamless multi-master sync, that scales from Big Data to Mobile, with an Intuitive HTTP/JSON API and designed for Reliability	http://couchdb.apache.org/
hugdru/cassandra	Manage massive amounts of data, fast, without losing sleep	http://cassandra.apache.org/

Figure 5.1: All the repositories stored in our service @frontend/repositories

Each repository may contain different featurewise files each having groups of features called domains which may follow a schema. As can be seen in Figure 5.2, MongoDB has one domain description with *id: 60* in its featurewise file with *id: 60*, which follows a schema to do with Big Data Features. In its domain table a list of features follow.

featurewise Home Repositories Schemas Compare

hugdru/mongodb featurewise (id: 60)

domain	schema	description
60	Big Data Architectures and Technologies (id: 3)	Classifies Big Data Technologies according to a set of features

Big Data Architectures and Technologies (id_domain: 60)

Admin

Subcategory	Value
Cluster monitoring	snapshot
Configuration Files	single
Database object count	supported
Node addition/removal	centralized tool
Physical storage usage	supported

Figure 5.2: Example of a domain that follows a schema @frontend/featurewise/60

The *Capture and Grouping* desideratum was attained successfully in respect to the capture of features and groups of features. Although the idea of placing a featurewise file alongside project code is applicable to all cases, its capture in our implementation is done only from GitHub repositories that have our GitHub App installed. Nevertheless, the approach could be easily extended to other code hubs that provide APIs. Moreover, the featurewise file could also serve to inform other

services and its transmission for consumption could be achieved using different techniques such as git hooks, scripts, and so on.

5.3 Validation

The desiderata expects feature group validation so that there is a guarantee that groups of features that follow a concern are comparable between different components. In our implementation a concern is represented through a domain which encompasses a schema URL and a set of features. The validation process involves checking whether the JSON Schema matches the data and alerting the user using the GitHub Checks API. Contrasting with the literature review:

- QuABaseBD uses semantic annotations and other techniques from Semantic MediaWiki to link concepts together creating structured information. However, group validation for well behaved comparisons does not apply in this case because knowledge exists in a single big knowledge unit focusing only in one domain.
- Feature-Comparison Websites too make no use of mechanisms to scope and validate features because they target a single domain per website which makes grouping related software according to sets of features a non issue.
- Multi-Faceted Comparison Websites have no group of features validation whatsoever to organise what is from what is not comparable. In most cases they rely on tags to identify similar software and free-form text comparisons.

Scoping features according to a schema allows us to find and compare any software that instantiated it. For instance Redis could be compared to different message brokers, and databases if it instantiated a schema for each one of those concerns in its featurewise file. See Figure 5.3.

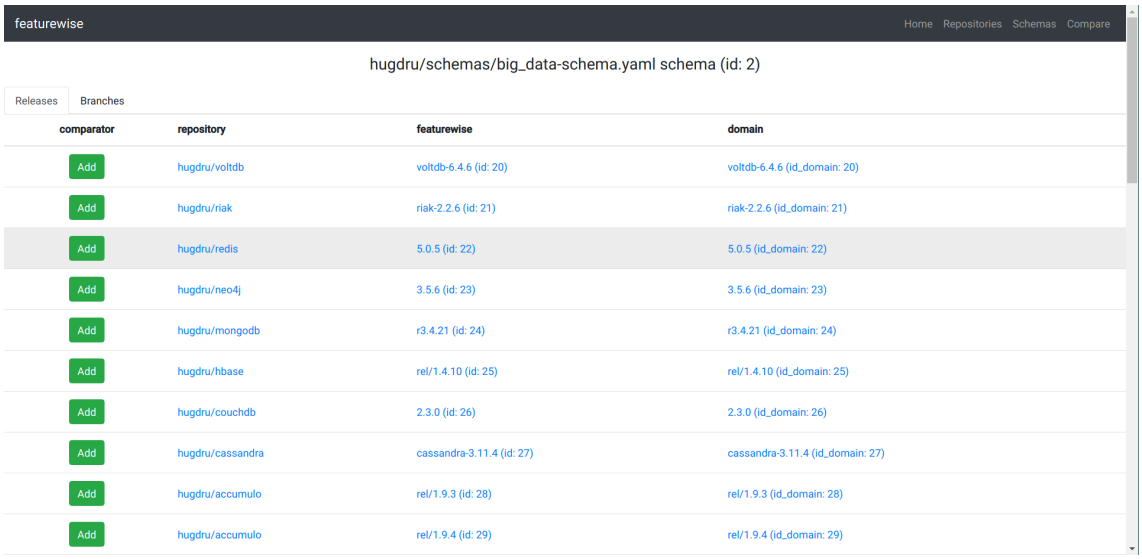

Figure 5.3: Table of software that follows a schema

The use of schemas to frame concerns does in fact make software components comparable among each other. Additionally, because groups with equal concerns follow the same schema all related software can be easily found and organised naturally through its actual feature sets and not through rigid hierarchies or tags.

5.4 Reuse

According to the desiderata, features and concerns should be reusable and duplicate information should be kept at a minimum. In an attempt to solve the above we:

- support YAML in addition to JSON to reference already defined concepts using anchors and references in featurewise and schema files;
- leverage JSON schema concepts in the construction of schema files to reduce duplicate definitions inside a schema and mapping to external sub-schemas;
- link groups of features to a concern using a schema URL.

Contrasting with the literature review:

- QuABaseBD has reuse functionality since it uses Semantic MediaWiki but because it only focuses on one domain in a single knowledge repository concern reuse does not apply.
- Feature-Comparison Websites too only touch one concern so reuse among different groups does not apply.
- Multi-Faceted Comparison websites lack any sort of reuse as defined above even though its support could provide grounds for a more common structure aiding comparisons between software with similar concerns.

As software feature descriptions are in the majority of cases scattered and multi-form the implemented reuse primitives can in fact reduce duplication and heterogeneity.

5.5 Versioning

The desiderata also calls for the versioning of releases and concerns because different software may contain different features possibly following updated definitions. This inevitably happens as domains are better understood. In our implementation different versions of similar software and the same software can be compared among each other so as to decide for instance between stable and mainline versions. Our solution improves upon the Feature and Comparison Websites investigated and the QuABaseBD knowledge base since they do not support any sort of versioning.

featurewise Home Repositories Schemas Compare

hugdru/riak featurewises

Releases Branches

comparator	featurewise	target_commitish	name	html_url
Add	riak-2.2.6 (id: 21)	master	riak-2.2.6	https://github.com/hugdru/riak/releases/tag/riak-2.2.6
Add	riak-2.2.7 (id: 51)	master	riak-2.2.7	https://github.com/hugdru/riak/releases/tag/riak-2.2.7
Add	riak-2.2.8 (id: 52)	master	riak-2.2.8	https://github.com/hugdru/riak/releases/tag/riak-2.2.8
Add	riak-2.2.9 (id: 53)	master	riak-2.2.9	https://github.com/hugdru/riak/releases/tag/riak-2.2.9
Add	riak-2.2.10 (id: 54)	master	riak-2.2.10	https://github.com/hugdru/riak/releases/tag/riak-2.2.10
Add	riak-2.2.11 (id: 55)	master	riak-2.2.11	https://github.com/hugdru/riak/releases/tag/riak-2.2.11

Figure 5.4: Example of multiple riak versions @frontend/repository/192900930

For two featurewise files to be comparable they must contain at least one common schema in a domain. However, schemas can evolve as well. Therefore, in our implementation they are also versioned. See Figure 5.5 for an example of two related schemas.

featurewise Home Repositories Schemas Compare

schema	description	repository	file
Big Data Architectures and Technologies (id: 2)	Classifies Big Data Technologies according to a set of features	hugdru/schemas	big_data-schema.yaml
Big Data Architectures and Technologies (id: 3)	Classifies Big Data Technologies according to a set of features	hugdru/schemas	big_data-schema.yaml

Figure 5.5: Example of two schemas @frontend/schemas

As elucidated by the above figures, our implementation versions feature files and concerns. As a result our approach and implementation not only provides current software feature information but also a history of its changes. Consequently, key software feature changes can be catalogued in a structured way acting as a structured software release notes changelog. Component selection processes can then leverage this information to judge the need for version upgrades and the selection of similar software following different versions — e.g. mongodb mainline vs mongodb stable, mongodb stable vs cassandra mainline, and so on.

5.6 Search and Comparison

Another requirement of the desiderata is the exposure of features and concerns so that external services can use this information to augment component search, comparison, and selection. In our implementation features are captured via the processing of GitHub App events and stored in a knowledge base which can be queried through a REST API. It contrasts to the literature review in the following way:

- Even though QuABaseBD provides a knowledge-base for Big Data Architecture and Technologies its contents are not meant to be queried by external systems. It does contain search functionality though but it is very rudimentary due to wiki limitations.

- Feature-comparison Websites do have meaningful information for component selection purposes but their extraction is in most cases via web scraping and is only concerned with certain component types.

- Multi-Faceted Comparison Websites such as StackShare, in Section 2.4.1.3, and Slant, in Section 2.4.1.1, leave a lot to be desired when comparing features since they are free form; so even if they provided an API their contents would not be very useful. They do excel at gathering user opinions about software which might help component selection processes but that is not our intent.

As can be seen in Figures 5.6 and 5.7 structured visualisations can be rendered from the data made available by our service.

Scalability

Subcategory	hugdru/mongodb r3.6.0 (id: 60)	hugdru/mongodb r3.5.6 (id: 46)
Request Load Balancing	fixed connections to a request coordinator	fixed connections to a request coordinator
Granularity of Write Locks	locks on whole database locks on tables/collections	locks on whole database
Scalable Distribution Architecture	horizontal partitioning of database horizontal partitioning and replication	horizontal partitioning of database horizontal partitioning and replication
Scalable Request Processing Architecture	centralized coordinator, but can be replicated	centralized coordinator, but can be replicated
Scaling Out - Adding Data Storage Capacity	automatic data rebalancing	automatic data rebalancing

Figure 5.6: Comparison between two mongo versions

In the frontend prototype different versions of software can be compared through tables in order to assist in the determination of the most suitable piece of software.

Data Distribution

Subcategory	hugdru/mongodb r3.6.0 (id: 60)	hugdru/riak riak-2.2.11 (id: 55)	hugdru/cassandra cassandra-3.11.8 (id: 36)
Query Architecture	distributed coordinator for shard key lookup	external load balancer required	distributed coordinator for shard key lookup
Data Distribution Method	assigned key ranges to nodes hash key	consistent hashing	consistent hashing
Automatic Data Rebalancing	new storage triggered administrative rebalancing tools data growth triggered	new storage triggered administrative rebalancing tools	administrative rebalancing tools
Physical Data Distribution	single cluster multiple data centers	single cluster multiple data centers	single cluster rack-aware on single cluster multiple data centers
Data Distribution Architecture	master-single slave master-multiple slaves	peer-to-peer	peer-to-peer
Queries using Non-Shard Key Value	non-indexed (scan)	secondary indexes	secondary indexes
Merging Query Results from Multiple Shards	sorted order paged from server	sorted order paged from server	random order paged from server

Figure 5.7: Comparison between different software

The sharing of repository features was made available by developing a public REST API and as exemplified above through the front-end prototype figures the service can in fact be used to assist component selection processes. Resulting in a knowledge-base that is queryable and evolves as projects change. Furthermore, the presence of a featurewise file in the root of a project makes it possible for other tools to consume its information without having to rely on our service.

5.7 Conclusion

The desiderata (Section 3.2.2) was attained successfully in all cases presenting noticeable improvements over the literature review approaches. We **capture features and groups of features** by encoding knowledge in a featurewise file, thus extending the gathering of structured features to multiple domains. Use schemas to **validate** whether a group of features follows a concern allowing for them to be compared, and organised naturally through their actual feature sets and not through rigid hierarchies or tags. Define **reuse** primitives to reduce duplication and heterogeneity when describing software features. Support **versioning** of feature files and concerns so that software with different versions can be catalogued and contrasted. And lastly, provide a public REST API that can, in fact, be used to assist component selection processes. Resulting in a knowledge-base that is **queryable** and evolves as projects change.

Chapter 6

Conclusions and Future Work

In this chapter we sum up the work done in this dissertation. Section 6.1, presents the main difficulties we encountered. Section 6.2, enunciates our contributions. Section 6.3, describes future work which could solidify and bolster the approach and implementation. And finally, Section 6.4, sums up the results of the dissertation.

6.1 Main Difficulties

The literature review, that is described in Chapter 2, was very exploratory and touched many areas of knowledge. This is the case because there are multiple papers about Component Selection with no clear connection to Software Architecture and vice versa. Also it was difficult to find approaches that guide users towards the choice of a component having as input structured architectural knowledge or problem context. Therefore, the literature review focused on finding different platforms, frameworks, data formats, and websites that were related to knowledge capture and component selection as a means to unravel issues and understand how the component selection process could be improved. As a consequence, the elaboration of the problem statement was more focused on identifying the major issues in this topic and the key characteristics that implementations should follow in order to effectively support an approach that helps with component selection.

6.2 Contributions

The main contribution of our work is a new way to capture structured knowledge in a way that fosters contributions and reuse of that information by other services. As a result the contributions of this dissertation are the following:

- A **framework** that assists producers and consumers of software in capturing the features of their software per domain, searching for appropriate components, and comparing them. As a side effect this would serve as a stepping stone towards Architecture Refactorings.

- A structured **approach** to capture project knowledge stored along side code that could possibly be extended to scenarios other than component selection — e.g. mapping of features to code through annotations which could be useful for building a dataset for machine learning purposes, whose aim would be to do the inverse, mapping code into features.

- A set of key characteristics described in a **desiderata**, an implementation of those principles in a service and a client side prototype for the comparison of software.

6.3 Future Work

As research is an incremental and evolutionary process, we would have liked to have delved into additional ideas and technical aspects of the framework. Next, we mention some of those aspirations.

6.3.1 Approach

Having built an implementation which followed our approach and desiderata, gathering feature information from repositories, the next logical step would be to come up with an approach to use that information to guide users towards a particular component considering problem context. For instance, combining software feature information with a reasoning process such as CoCoADvISE which is based on QOC. An implementation of such could involve building a web app where criteria for component selection would be described collaboratively in a versioned QOC decision tree. Ideally, comparison websites would then fetch those representations to guide their users.

6.3.2 Prototype

Even though the implementation follows the desiderata there is still work to be done to make the system more user friendly, and production ready:

- Manually creating feature files according to schemas is error-prone because they are only checked after commit pushes to GitHub through the *GitHub checks* functionality. Featurewise files creation could be improved by building a web application that generates a form with auto-completion per domain according to its schema. Additionally, the website could use GitHub OAuth to import and export featurewise files from/to repositories;

- Moreover, IDE plugins could be developed to support that same functionality directly from editors;

- Finally, there is still work to be done on testing and infrastructure code to make the system production ready — i.e. leverage docker swarm or kubernetes to manage and scale containers as events increase or decrease — and more searchable — i.e using a search engine to index featurewise files and schemas.

6.3.3 Ideas to explore

As we understood more of the problem at hand several alternatives were discussed that could be looked at further:

- Use the Semantic Web stack to combine component knowledge from multiple sources including structured information from repositories.

- Leverage RDFa to embed RDF statements inside the HTML documentation of software projects or even inside code comments as a means to sum up their major traits.

- Create and investigate how a lightweight AKMS with support for the encoding of design rationale located along-side code could aid both architectural decisions recording as well as component selection.

- Determine how the multiple parts of the architecture of a system (the code, documentation, and operational metrics) could be described in a rich model to promote not only the suggestion of alternate components but also, and more importantly, the refactoring of architectures using better-suited structures and components.

6.3.4 User Studies

To confidently show the merits of the approach, user studies would need to be conducted. They could help us identify flaws in the approach or the prototype. Some of these user studies could be:

- Exploratory interviews or questionnaires to assess if component developers would consider using our tool, and if not, why.

- Interviews or questionnaires with experienced architects to understand the merits of the approach by developers creating the components, and by those selecting the most appropriate components for the software that they are creating.

- Industrial case studies using the implemented prototype, to help us understand the benefits of using the tool (and the underlying approach) for component selection in real-world scenarios.

6.4 Conclusion

Our interpretation of the desiderata resulted in an implementation that followed its principles which were derived from the issues identified in the literature review and the approach. In conclusion, after the analysis of the desiderata in the evaluation chapter we remain confident that our approach and implementation better assist feature comparisons in component selection processes when compared to current approaches, but it still needs to be put to the test by researchers willing to conduct users studies with developers of components and its consumers.

References

M A Babar, X Wang, and I Gorton. PAKME: A tool for capturing and using architecture design knowledge. In *2005 Pakistan Section Multitopic Conference, INMIC*, 2005. doi: 10.1109/INMIC.2005.334419. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-50249121296{&}doi=10.1109{%}2FINMIC.2005.334419{&}partnerID=40{&}md5=fbae611ed497b3084fc61262a3deed7b.

M A Babar, I Gorton, and B Kitchenham. *A framework for supporting architecture knowledge and rationale management*. 2006. doi: 10.1007/978-3-540-30998-7_11. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-70350692949{&}doi=10.1007{%}2F978-3-540-30998-7{_}11{&}partnerID=40{&}md5=c6004aac9c251c3831ab60998031b280.

L Bass, M Klein, and F Bachmann. Quality attribute design primitives and the attribute driven design method. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2290:169–186, 2002. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-84944072009{&}partnerID=40{&}md5=89fb8f547b2ceb5c160e378411006526.

Tim Berners-Lee and Mark Fischetti. *Weaving the Web : the origins and future of the World Wide Web*. London : Orion Business, 1st ed edition, 1999. ISBN 0752820907.

Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The World-Wide Web. *Communications of the ACM*, 37(8):76–82, aug 1994. ISSN 00010782. doi: 10.1145/179606.179671. URL http://portal.acm.org/citation.cfm?doid=179606.179671.

Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked Data - The Story So Far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22, jul 2009. ISSN 1552-6283. doi: 10.4018/jswis.2009081901. URL http://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/jswis.2009081901.

Barry Boehm and Hasan Kitapci. The WinWin Approach: Using a Requirements Negotiation Tool for Rationale Capture and Use. In *Rationale Management in Software Engineering*, pages 173–190. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. doi: 10.1007/978-3-540-30998-7_8. URL http://link.springer.com/10.1007/978-3-540-30998-7{_}8.

Jan Bosch. Software Architecture: The Next Step. (May 2004):194–199, 2004. ISSN 0302-9743. doi: 10.1007/978-3-540-24769-2_14. URL http://link.springer.com/10.1007/978-3-540-24769-2{_}14.

Rafael Capilla, Francisco Nava, Sandra Pérez, and Juan C Dueñas. A Web-based Tool for Managing Architectural Design Decisions. *SIGSOFT Softw. Eng. Notes*, 31(5), 2006. ISSN 0163-5948. doi: 10.1145/1163514.1178644. URL http://doi.acm.org/10.1145/1163514.1178644.

Rafael Capilla, Anton Jansen, Antony Tang, Paris Avgeriou, and Muhammad Ali Babar. 10 years of software architecture knowledge management: Practice and future. *Journal of Systems and Software*, 116:191–205, 2016. ISSN 01641212. doi: 10.1016/j.jss.2015.08.054. URL http://dx.doi.org/10.1016/j.jss.2015.08.054.

Dilip Soni Christine Hofmeister, Robert Nord. *Applied Software Architecture*. Addison-Wesley Professional, 1999. ISBN 9780201325713,0201325713. URL https://books.google.pt/books?hl=en{&}lr={&}id=3klAPCIB3hQC{&}oi=fnd{&}pg=PR13{&}dq=0201325713{&}ots=NoImzT6yON{&}sig=grpeKWzIjh3BphbKU{_}HKDWl7KDU{&}redir{_}esc=y{#}v=onepage{&}q=globalanalysis{&}f=false.

Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond*. Pearson Education, 2002. ISBN 0201703726.

E J Conklin. A Process-Oriented Approach to Design Rationale. *Human-Computer Interaction*, 6(3-4):357–391, 1991. doi: 10.1080/07370024.1991.9667172. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-0026407773{&}doi=10.1080{%}2F07370024.1991.9667172{&}partnerID=40{&}md5=2b4b61b758fba700c452ad10be79a0e3.

R C De Boer, R Farenhorst, P Lago, H Van Vliet, V Clerc, and A Jansen. Architectural knowledge: Getting to the core. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4880 LNCS:197–214, 2007. doi: 10.1007/978-3-540-77619-2_12. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-49949095038{&}doi=10.1007{%}2F978-3-540-77619-2{_}12{&}partnerID=40{&}md5=0da08141fdcbed800f32c39c7972519e.

K. A. De Graaf, P. Liang, A. Tang, and H. Van Vliet. How organisation of architecture documentation affects architectural knowledge retrieval. *Science of Computer Programming*, 121:75–99, 2016. ISSN 01676423. doi: 10.1016/j.scico.2015.10.014. URL http://dx.doi.org/10.1016/j.scico.2015.10.014.

Tommaso Di Noia, Marina Mongiello, and Umberto Straccia. Fuzzy description logics for component selection in software design. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9509, pages 228–239. 2015. ISBN 9783662492239. doi: 10.1007/978-3-662-49224-6_19. URL http://link.springer.com/10.1007/978-3-662-49224-6{_}19.

Tommaso Di Noia, Marina Mongiello, Francesco Nocera, and Umberto Straccia. A fuzzy ontology-based approach for tool-supported decision making in architectural design. *Knowledge and Information Systems*, pages 1–30, mar 2018. ISSN 0219-1377. doi: 10.1007/s10115-018-1182-1. URL https://doi.org/10.1007/s10115-018-1182-1http://link.springer.com/10.1007/s10115-018-1182-1.

Wei Ding, Peng Liang, Antony Tang, and Hans Van Vliet. Knowledge-based approaches in software documentation: A systematic literature review. *Information and Software Technology*, 56(6):545–567, 2014a. ISSN 09505849. doi: 10.1016/j.infsof.2014.01.008. URL http://dx.doi.org/10.1016/j.infsof.2014.01.008.

Wei Ding, Peng Liang, Antony Tang, Hans Van Vliet, and Mojtaba Shahin. How do open source communities document software architecture: An exploratory survey. *Proceedings of the IEEE International Conference on Engineering of Complex Computer Systems, ICECCS*, pages 136–145, 2014b. doi: 10.1109/ICECCS.2014.26.

Siamak Farshidi, Slinger Jansen, Rolf De Jong, and Sjaak Brinkkemper. A decision support system for cloud service provider selection problem in software producing organizations. *Proceeding - 2018 20th IEEE International Conference on Business Informatics, CBI 2018*, 1:139–148, 2018a. ISSN 1521-6934. doi: 10.1109/CBI.2018.00024.

Siamak Farshidi, Slinger Jansen, Rolf De Jong, and Sjaak Brinkkemper. Multiple criteria decision support in requirements negotiation. *CEUR Workshop Proceedings*, 2075, 2018b. ISSN 16130073.

Siamak Farshidi, Slinger Jansen, Rolf de Jong, and Sjaak Brinkkemper. A decision support system for software technology selection. *Journal of Decision Systems*, 27(sup1):98–110, may 2018c. ISSN 1246-0125. doi: 10.1080/12460125.2018.1464821. URL https://doi.org/10.1080/12460125.2018.1464821https://www.tandfonline.com/doi/full/10.1080/12460125.2018.1464821.

X Franch, A Susi, M C Annosi, C Ayala, R Glott, D Gross, R Kenett, F Mancinelli, P Ramsamy, C Thomas, D Ameller, S Bannier, N Bergida, Y Blumenfeld, O Bouzereau, D Costal, M Domínguez, K Haaland, L López, M Morandini, and A Siena. Managing risk in open source software adoption. In *ICSOFT 2013 - Proceedings of the 8th International Joint Conference on Software Technologies*, pages 258–264, 2013. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-84887052413{&}partnerID=40{&}md5=3bc1e271f87bdcec926e75f5f458ca25.

Rakesh Garg, R. K. Sharma, Kapil Sharma, and R. K. Garg. MCDM based evaluation and ranking of commercial off-the-shelf using fuzzy based matrix method. *Decision Science Letters*, 6:117–136, 2017. ISSN 19295804. doi: 10.5267/j.dsl.2016.11.002. URL http://www.growingscience.com/dsl/Vol6/dsl{_}2016{_}30.pdf.

J González-Huerta, E Insfrán, and S Abrahão. Defining and validating a multimodel approach for product architecture derivation and improvement. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8107 LNCS:388–404, 2013. doi: 10.1007/978-3-642-41533-3_24. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-84886848508{&}doi=10.1007{%}2F978-3-642-41533-3{_}24{&}partnerID=40{&}md5=470a1e83060cea2a038d5adeb4f1f478.

Ian Gorton, John Klein, and Albert Nurgaliev. Architecture Knowledge for Evaluating Scalable Databases. In *2015 12th Working IEEE/IFIP Conference on Software Architecture*, pages 95–104. IEEE, may 2015. ISBN 978-1-4799-1922-2. doi: 10.1109/WICSA.2015.26. URL http://ieeexplore.ieee.org/document/7158508/.

Ian Gorton, Rouchen Xu, Yiming Yang, Hanxiao Liu, and Guoqing Zheng. Experiments in Curation: Towards Machine-Assisted Construction of Software Architecture Knowledge Bases. *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017*, pages 79–88, 2017. doi: 10.1109/ICSA.2017.27.

Tom Heath and Christian Bizer. Linked Data: Evolving the Web into a Global Data Space. *Synthesis Lectures on the Semantic Web: Theory and Technology*, 1(1):1–136, feb 2011. ISSN 2160-4711. doi: 10.2200/S00334ED1V01Y201102WBE001. URL http://www.morganclaypool.com/doi/abs/10.2200/S00334ED1V01Y201102WBE001.

C. Hofmeister, R.L. Nord, and D. Soni. Global Analysis: moving from software requirements specification to structural views of the software architecture. *IEE Proceedings - Software*, 152(4):187, 2005. ISSN 14625970. doi: 10.1049/ip-sen:20045052. URL https://digital-library.theiet.org/content/journals/10.1049/ip-sen{_}20045052.

IEEE Architecture Working Group. IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. *IEEE Std 1471-2000*, 2000. doi: 10.1109/IEEESTD.2000.91944.

ISO/IEC/IEEE. ISO/IEC/IEEE 42010: 2011 Systems and software engineering – Architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)*, 1:1–46, 2011. doi: 10.1109/IEEESTD.2011.6129467.

A Jansen, P Avgeriou, and J S van der Ven. Enriching software architecture documentation. *Journal of Systems and Software*, 82(8):1232–1248, 2009. doi: 10.1016/j.jss.2009.04.052. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-67949120215{&}doi=10.1016{%}2Fj.jss.2009.04.052{&}partnerID=40{&}md5=cafdac96d9f1509604e32915fee646e3.

Anton Jansen and Jan Bosch. Software architecture as a set of architectural design decisions. *Proceedings - 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005*, 2005: 109–120, 2005. ISSN 0163-769X. doi: 10.1109/WICSA.2005.61.

A.P.J. Jarczyk, Peter Löffler, and FM Shipman. Design rationale for software engineering: a survey. *Proceedings of the Hawaii International Conference on System Sciences*, 25:577–577, 1992. doi: 10.1109/HICSS.1992.183309. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.122.4936{&}rep=rep1{&}type=pdf.

Loveleen Kaur and Hardeep Singh. Software Component Selection techniques - A review. *pdfs.semanticscholar.org*, 5(3):3739–3742, 2014. URL https://pdfs.semanticscholar.org/4c60/dc4fdfc3c778ea22a52ce3166ba2a47d443b.pdf.

Rick Kazman, J. Asundi, and Mark Klein. Quantifying the Costs and Benefits of RSS in Perishables. *Agenda*, (August):297–306, 2001. ISSN 0270-5257. doi: 10.1109/ICSE.2001.919103. URL http://ieeexplore.ieee.org/document/919103/.

John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. Application-Specific Evaluation of No SQL Databases. *Proceedings - 2015 IEEE International Congress on Big Data, BigData Congress 2015*, pages 526–534, 2015a. ISSN 16113349. doi: 10.1109/BigDataCongress.2015.83.

John Klein, Ian Gorton, Neil Ernst, Patrick Donohoe, Kim Pham, and Chrisjan Matser. Performance Evaluation of NoSQL Databases. In *Proceedings of the 1st Workshop on Performance Analysis of Big Data Systems - PABS '15*, pages 5–10, New York, New York, USA, 2015b. ACM Press. ISBN 9781450333382. doi: 10.1145/2694730.2694731. URL http://dl.acm.org/citation.cfm?doid=2694730.2694731.

Martin Kleppmann. *Designing data-intensive applications : the big ideas behind reliable, scalable, and maintainable systems*. 2017. ISBN 9781491903094.

P B Kruchten. The 4+1 View Model of architecture. *IEEE Software*, 12(6):42–50, nov 1995. ISSN 0740-7459. doi: 10.1109/52.469759.

Philippe Kruchten. An Ontology of Architectural Design Decisions in Software-Intensive Systems. *Groningen Workshop on Software Variability management*, (January 2004):55–62, 2004.

Rob J Kusters, Lieven Pouwelse, Harry Martin, and Jos Trienekens. Decision Criteria for Software Component Sourcing - Steps towards a Framework. *Proceedings of the 18th International Conference on Enterprise Information Systems*, 1(Iceis):580–587, 2016. doi: 10.5220/0005915005800587. URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0005915005800587.

Jintae Lee. Extending the Potts and Bruns model for recording design rationale. In *[1991 Proceedings] 13th International Conference on Software Engineering*, pages 114–125. IEEE Comput. Soc. Press, 1991. ISBN 0-8186-2140-0. doi: 10.1109/ICSE.1991.130629. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=130629http://ieeexplore.ieee.org/document/130629/.

Rick Kazman Len Bass, Paul Clements. *Software architects in practice*. Professional, Addison-Wesley, 2003. ISBN 0321154959.

Ioanna Lytra, Huy Tran, and Uwe Zdun. Supporting consistency between architectural design decisions and component models through reusable architectural knowledge transformations. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7957 LNCS:224–239, 2013. ISSN 16113349. doi: 10.1007/978-3-642-39031-9_20.

Ioanna Lytra, Gerhard Engelbrecht, Daniel Schall, and Uwe Zdun. Reusable Architectural Decision Models for Quality-Driven Decision Support: A Case Study from a Smart Cities Software Ecosystem. *Proceedings - 3rd International Workshop on Software Engineering for Systems-of-Systems, SESoS 2015*, pages 37–43, 2015. doi: 10.1109/SESoS.2015.14.

Allan MacLean, Richard M Young, Victoria M E Bellotti, and Thomas P Moran. Questions, Options, and Criteria: Elements of Design Space Analysis. *Human–Computer Interaction*, 6(3-4):201–250, 1991. doi: 10.1080/07370024.1991.9667168. URL https://www.tandfonline.com/doi/abs/10.1080/07370024.1991.9667168.

Ivano Malavolta, Patricia Lago, Henry Muccini, Patrizio Pelliccione, and Antony Tang. What industry needs from architectural languages: A survey. *IEEE Transactions on Software Engineering*, 39(6):869–891, 2013. ISSN 00985589. doi: 10.1109/TSE.2012.74.

Raymond J. McCall. PHI: a conceptual foundation for design hypermedia. *Design Studies*, 12(1):30–41, jan 1991. ISSN 0142694X. doi: 10.1016/0142-694X(91)90006-I. URL http://linkinghub.elsevier.com/retrieve/pii/0142694X9190006I.

Mehdi Mirakhorli and Jane Cleland-Huang. Detecting, Tracing, and Monitoring Architectural Tactics in Code. *IEEE Transactions on Software Engineering*, 42(3):206–221, 2016. ISSN 00985589. doi: 10.1109/TSE.2015.2479217.

David L. Parnas and Paul C. Clements. A rational design process: How and why to fake it. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 186 LNCS, pages 80–100. 1985. ISBN 9783540151999. doi: 10.1007/3-540-15199-0_6. URL http://link.springer.com/10.1007/3-540-15199-0{_}6.

F. Pena-Mora, D. Sriram, and R. Logcher. SHARED-DRIMS: SHARED design recommendation-intent management system. In *[1993] Proceedings Second Workshop on Enabling Technologies@m_Infrastructure for Collaborative Enterprises*, pages 213–221. IEEE Comput. Soc. Press, 1993. ISBN 0-8186-4082-0. doi: 10.1109/ENABL.1993.263047. URL http://ieeexplore.ieee.org/document/263047/.

Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992. ISSN 01635948. doi: 10.1145/141874.141884. URL http://portal.acm.org/citation.cfm?doid=141874.141884.

Kai Petersen, Deepika Badampudi, Syed Muhammad Ali Shah, Krzysztof Wnuk, Tony Gorschek, Efi Papatheocharous, Jakob Axelsson, Séverine Sentilles, Ivica Crnkovic, and Antonio Cicchetti. Choosing Component Origins for Software Intensive Systems: In-House, COTS, OSS or Outsourcing? - A Case Survey. *IEEE Transactions on Software Engineering*, 44(3):237–261, 2018. ISSN 00985589. doi: 10.1109/TSE.2017.2677909.

C Potts and G Bruns. Recording the Reasons for Design Decisions. In *Proceedings of the 10th International Conference on Software Engineering*, ICSE '88, pages 418–427, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press. ISBN 0-89791-258-6. URL http://dl.acm.org/citation.cfm?id=55823.55863.

Marta Sabou, Fajar J. Ekaputra, Tudor Ionescu, Juergen Musil, Daniel Schall, Kevin Haller, Armin Friedl, and Stefan Biffl. Exploring Enterprise Knowledge Graphs: A Use Case in Software Engineering. In *Advances in Information Technologies for Electromagnetics*, volume 1, pages 560–575. Springer Netherlands, 2018. doi: 10.1007/978-3-319-93417-4_36. URL http://dx.doi.org/10.1007/978-3-319-93417-4{_}36http://link.springer.com/10.1007/978-1-4020-4749-5{_}3http://link.springer.com/10.1007/978-3-319-93417-4{_}36.

Arman Shahbazian, Youn Kyu Lee, Duc Le, Yuriy Brun, and Nenad Medvidovic. Recovering Architectural Design Decisions. *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*, pages 95–104, 2018. ISSN 12245984. doi: 10.1109/ICSA.2018.00019.

John Slankas and Laurie Williams. Automated extraction of non-functional requirements in available documentation. *2013 1st International Workshop on Natural Language Analysis in Software Engineering, NaturaLiSE 2013 - Proceedings*, pages 9–16, 2013. ISSN 09473602. doi: 10.1109/NAturaLiSE.2013.6611715.

Mohamed Soliman, Amr Rekaby Salama, Matthias Galster, Olaf Zimmermann, and Matthias Riebisch. Improving the Search for Architecture Knowledge in Online Developer Communities. *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*, pages 186–195, 2018. doi: 10.1109/ICSA.2018.00028.

Umberto Straccia. *Foundations of Fuzzy Logic and Semantic Web Languages*. 2013. ISBN 9781439853481.

Umberto Straccia. All about fuzzy description logics and applications. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9203, pages 1–31. 2015. ISBN 9783319217673. doi: 10.1007/978-3-319-21768-0_1. URL http://link.springer.com/10.1007/978-3-319-21768-0{_}1.

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. doi: 10.1016/j.jss.2006.08.040. URL https://www.scopus.com/inward/record.uri?eid=2-s2.0-33947402169{&}doi=10.1016{%}2Fj.jss.2006.08.040{&}partnerID=40{&}md5=e14624606973821f6e56eba995c32998.

Antony Tang, Muhammad Ali Babar, Ian Gorton, and Jun Han. A survey of architecture design rationale. *Journal of Systems and Software*, 79(12):1792–1804, dec 2006. ISSN 01641212. doi: 10.1016/j.jss.2006.04.029. URL https://linkinghub.elsevier.com/retrieve/pii/S0164121206001415.

Tim Berners-Lee. Linked Data - Design Issues, 2006. URL https://www.w3.org/DesignIssues/LinkedData.html.

Dan Tofan, Matthias Galster, Paris Avgeriou, and Wes Schuitema. Past and future of software architectural decisions - A systematic mapping study. *Information and Software Technology*, 56(8):850–872, 2014. ISSN 09505849. doi: 10.1016/j.infsof.2014.03.009. URL http://dx.doi.org/10.1016/j.infsof.2014.03.009.

Stephen E Toulmin. *The Uses of Argument*. Cambridge University Press, 1958. doi: 10.1080/00048405985200191.

Jos. J. M. Trienekens, Rob Kusters, Jan van Moll, and Casper Vos. Decision Criteria for Software Component Sourcing - An Initial Framework on the Basis of Case Study Results. *Proceedings of the 19th International Conference on Enterprise Information Systems*, 2(Iceis): 279–286, 2017. doi: 10.5220/0006294302790286. URL http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006294302790286.

Jeff Tyree and Art Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27, 2005. ISSN 07407459. doi: 10.1109/MS.2005.27.

HWJ Rittel W Kunz. *Issues as elements of information systems*. 1970.

W3C. RDF Schema 1.1. Technical report, 2014. URL https://www.w3.org/TR/rdf-schema/.

Rainer Weinreich and Iris Groher. Software architecture knowledge management approaches and their support for knowledge management activities: A systematic literature review. *Information and Software Technology*, 80:265–286, 2016. ISSN 09505849. doi: 10.1016/j.infsof.2016.09.007. URL http://dx.doi.org/10.1016/j.infsof.2016.09.007.

Yingzhong Zhang, Xiaofang Luo, Jian Li, and Jennifer J. Buis. A semantic representation model for design rationale of products. *Advanced Engineering Informatics*, 27(1):13–26, 2013. ISSN 14740346. doi: 10.1016/j.aei.2012.10.005. URL http://dx.doi.org/10.1016/j.aei.2012.10.005.

Appendix A

Feature Taxonomy of Gorton et al

Here lies the tables that represent the taxonomy of features of (Gorton et al., 2015) first described in 2.1. Additional features and values where added by looking at the current state of QuABaseBD.

A.1 Data Architecture

A.1.1 Data Model

Table A.1: Data Model QuABaseBD Subcategories
Source: (Gorton et al., 2015)

Subcategories	Features
Data Organization	Data Model, Fixed Schema, Opaque Data Objects, Hierarchical Data Objects
Keys and Indexes	Automatically allocated Primary Key, Composite Keys, Secondary Indexes
Query Approaches	Query by Key Ranges, Query by Partial Keys, Query by Non-key Values, Map Reduce API, Indexed Text Search

Table A.2: Data Model QuABaseBD
Source: (Gorton et al., 2015)

Features	Allowed Values
Data Model	Column, Key-Value, Graph, Document, Object, Relational, XML
Fixed Schema	Required, Not required, Optional
Opaque Data Objects	Required, Not required
Hierarchical Data Objects	Supported, Not supported
Automatically allocated Primary Key	Supported, Not supported
Composite Keys	Supported, Not supported
Secondary Indexes	Supported, Not supported
Query by Key Ranges	Supported, Not supported
Query by Partial Keys	Supported, Not supported
Query by Non-Key Values (Scan)	Supported, Not supported
Map Reduce API	Builtin, Integrated with an external framework, Not supported
Indexed Text Search	Support in a plugin (e.g Solr), Proprietary (database-specific), Not supported

A.1.2 Query languages

Table A.3: Query languages QuABaseBD Subcategories
Source: (Gorton et al., 2015)

Subcategories	Features
Query Language Options	API-based, Declarative Query Language, REST/HTTP-based Queries, Languages Supported
Query Language Features	Cursor-based Queries, JOIN-style queries, Complex Data Types, Restrict Query Result Set Size, Key Matching Options, Sort Options, Triggers, Data Object Expiry

Table A.4: Query languages QuABaseBD
Source: (Gorton et al., 2015)

Features	Allowed Values
API-based	Supported, Not supported
Declarative Query Language	Supported, Not supported
REST/HTTP-based Queries	Supported, Not supported
Languages Supported	Java, C#, Python, C/C++, Perl, Ruby, Scala, Erlang, Javascript, PHP
Cursor-based Queries	Supported, Not Supported
JOIN-style Queries	Supported, Not Supported
Complex Data Types	Lists, Maps, Sets, Nested Structures, Arrays, Geospatial, None
Restrict Query Result Set Size	Supported, Not Supported
Key Matching Options	Exact, Partial Match, Wildcards, Regular Expressions
Sort Options	Ascending, Descending, None
Triggers	Pre-commit, Post-commit, Not supported
Data Object Expiry	Supported, Not Supported

A.1.3 Consistency

Table A.5: Consistency QuABaseBD Subcategories
Source: (Gorton et al., 2015)

Subcategories	Features
Strong Consistency	Object Level Atomic Updates, ACID Transactions, Distributed Transactions, Durable Writes
Eventual Consistency Features (Read and Write Setting)	Quorum Reads and Writes, Number of Replicas to Read, Number of Replicas to Write, Writes with Unavailable Replicas, Read from Master Only
Eventual Consistency Features (Other Settings)	Resolving Write Conflicts

Table A.6: Consistency QuABaseBD
Source: (Gorton et al., 2015)

Features	Allowed Values
Object Level Atomic Updates	Supported, Multi Version Concurrency Control (MVCC), Not supported - conflicts allowed
ACID Transactions	Supported, Lightweight transactions (e.g. compare and set), Not Supported
Distributed Transactions	Supported, Not supported
Durable Writes	Supported, Not supported
Quorum Reads and Writes	In the client, In the database, In both the database and data center, Not relevant, Not supported
Number of Replicas to Read	In the client, Not applicable - master-slave, Not supported
Number of Replicas to Write	In the client, Not applicable - master-slave, Not supported
Writes with Unavailable Replicas	A rollback at all replicas, No rollback: write returns replication error, Hinted handoffs: writes are applied later when a replica recovers, Not applicable
Read from Master Only	Not applicable - peer to peer, Not supported, Specified in the client, Specified in the database configuration, Specified in the application configuration (e.g. Web load balancer)
Resolving Write Conflicts	Supported, Not supported, Not applicable: master-slave, Not applicable: single threaded

A.2 Software Architecture

A.2.1 Scalability

Table A.7: Scalability QuABaseBD
Source: (Gorton et al., 2015)

Features	Allowed Values
Scalable Distribution Architecture	Replicate complete database only, Horizontal partitioning of database, Horizontal partitioning and replication
Scaling Out - Adding Data Storage Capacity	Automatic data rebalancing, Manual data rebalancing, N/A - single server only
Request Load Balancing	Fixed connections to a request coordinator, Client requests load balanced across coordinators, Uses HTTP-based load balancers
Granularity of Write Locks	No locks - conflicts allowed, No locks - optimistic concurrency model, Locks on updated objects only, Locks on tables/collections, Locks on whole database, No locks - single threaded execution
Scalable Request Processing Architecture	Fully distributed - any node can act as a coordinator, Centralised coordinator but can be replicated, Not scalable (bottleneck), Based on an external load balancer

A.2.2 Data Distribution

Table A.8: Data Distribution QuABaseBD Subcategories
Source: (Gorton et al., 2015)

Subcategories	Features
Distribution Architecture	Data Distribution Architecture, Data Distribution Method, Automatic Data Rebalancing, Physical Data Distribution
Querying Distributed Database	Query Architecture, Queries using Non-Shard Key Value, Merging Query Results from Multiple Shards

Table A.9: Data Distribution QuABaseBD
Source: (Gorton et al., 2015)

Features	Allowed Values
Data Distribution Architecture	Single database only, Master-single slave, Master-multiple slaves, Peer-to-peer
Data Distribution Method	Assigned key ranges to nodes, Hash key, Consistent hashing, Not relevant (single server only)
Automatic Data Rebalancing	Failure triggered, New storage triggered, Data growth triggered, Schedulable rebalancing, Administrative rebalancing tools, No rebalancing (single server only)
Physical Data Distribution	Single cluster, Rack-aware on single cluster, Multiple data centers
Query Architecture	Centralized coordinator for shard key lookup, Distributed coordinator for shard key lookup, Direct shard connection only (resolved in client), External load balancer required
Queries using Non-Shard Key Value	Not supported, Secondary indexes, Non-indexed (scan)
Merging Query Results from Multiple Shards	Random order, Sorted order, Paged from server, Not supported

A.2.3 Data Replication

Table A.10: Data Replication QuABaseBD Subcategories
Source: (Gorton et al., 2015)

Subcategories	Features
Replication Features	Replication Architecture, Replication for Backup, Replication across Data Centers, Replica Writes, Replica Reads, Read Repair
Failover Features	Automatic Replica Failure Detection, Automatic Failover, Automatic New Master Election after Failure, Replica Recovery and Resynchronization

Table A.11: Data Replication QuABaseBD
Source: (Gorton et al., 2015)

Features	Allowed Values
Replication Architecture	Master-slave, Peer-to-peer
Replication for Backup	Supported, Not supported
Replication across Data Centers	Supported by data center aware features, Supported by enterprise version only (data center aware), Supported by standard data replication mechanisms
Replica Writes	To master replica only, To any replica, To multiple replicas, To specified replica (configurable)
Replica Reads	From master replica only, From any replica, From multiple replicas, From specified replica (configurable)
Read Repair	Per query, Background, Not relevant, Not applicable
Automatic Replica Failure Detection	Supported, Not supported
Automatic Failover	Supported, Not supported
Automatic New Master Election after Failure	Supported, Not supported, Not relevant
Replica Recovery and Resynchronization	Supported - automatic, Performed by administrator, Not supported

A.2.4 Security

Table A.12: Data Replication QuABaseBD Subcategories
Source: (Gorton et al., 2015)

Subcategories	Features
Authentication	Client Authentication, Server authentication, Credential Store
Role Based Security	Role Based Security, Security Role Options, Scope of Roles
Database Security and Logging	Database Encryption, Logging

Table A.13: Security QuABaseBD
Source: (Gorton et al., 2015)

Features	Allowed Values
Client Authentication	Custom user/password, X509, LDAP, Kerberos, SSL
Server Authentication	Shared keyfile, SSL, Not secured, Server account credentials
Credential Store	In database, External file, Certificates only
Role Based Security	Supported, Not supported, Supported - enterprise version only, Supported - requires programmatic extension
Security Role Options	Multiple roles per user, Role inheritance, Default roles, Custom roles, Not supported
Scope of Roles	Cluster, Database, Collection, Object, Field
Database Encryption	Supported, Not supported
Logging	No logging, Configurable event logging, Fixed event logging, Requires external components (e.g. Web Servers)

A.2.5 Administration and Monitoring

Table A.14: Administration and Monitoring QuABaseBD
Source: (Gorton et al., 2015)

Features	Allowed Values
Configuration files	Single, Multiple
Node command line access	Authenticated, Non-authenticated, Not supported
Node addition/removal	Centralized tool, Single file, Multiple files
Cluster monitoring	Real-time, Snapshot, Entreprise version only
Dump database configuration	Supported, Not supported
Database object count	Supported, Not supported
Physical storage usage	Supported, Not supported

Appendix B

ISO/IEC/IEEE 42010:2011 Conceptual Models and Definitions

Here lies the conceptual models for architecture descriptions according to standard ISO/IEC/IEEE 42010:2011 (ISO/IEC/IEEE, 2011).

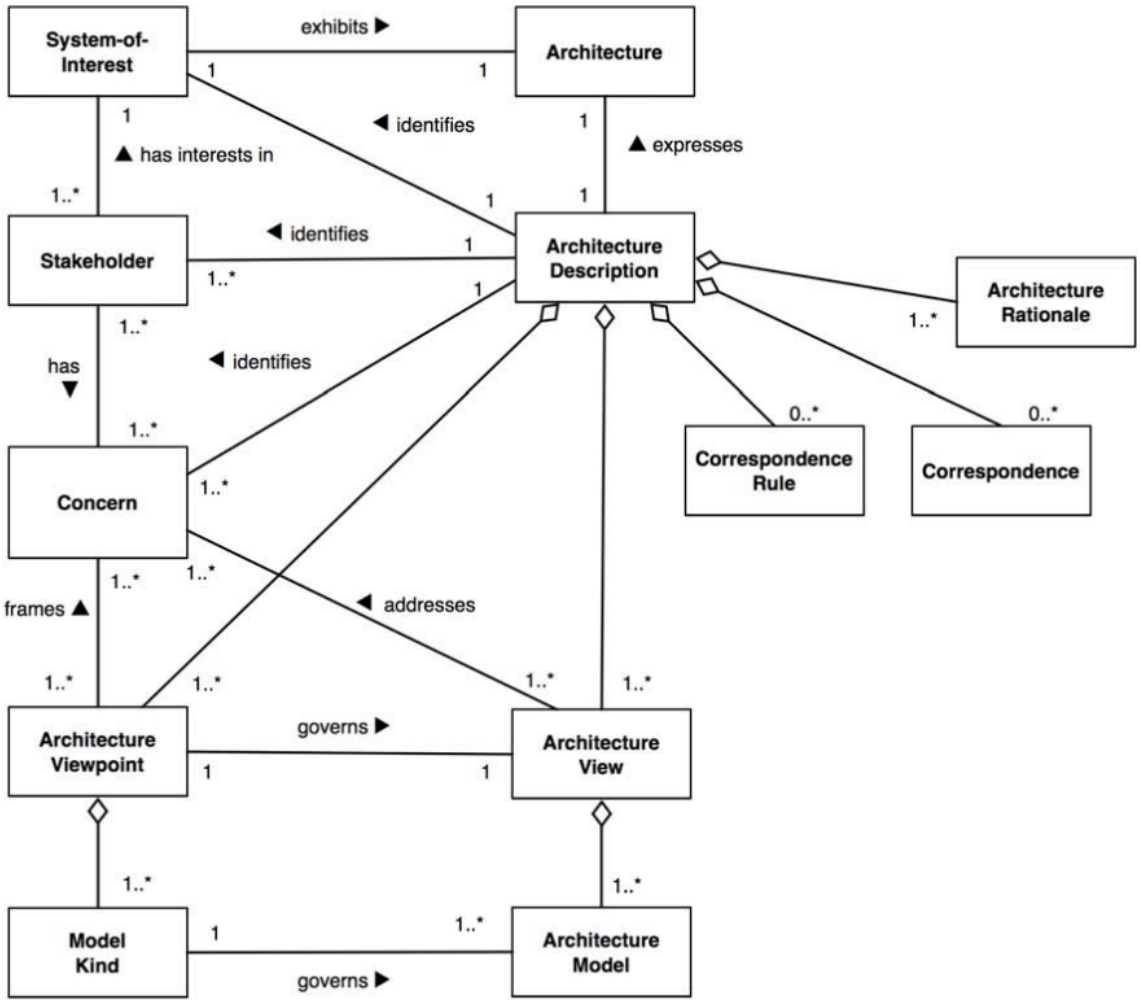

Figure B.1: Conceptual model of an architecture description
Source: (ISO/IEC/IEEE, 2011)

"Whereas an architecture description is a work product, an architecture is abstract, consisting of concepts and properties." ISO/IEC/IEEE (2011).

"This International Standard does not specify any format or media for recording architecture descriptions. It is intended to be usable for a range of approaches to architecture description including document-centric, model-based, and repository-based techniques." ISO/IEC/IEEE (2011)

"This International Standard does not prescribe the process or method used to produce architecture descriptions. This International Standard does not assume or prescribe specific architecting methods, models, notations or techniques used to produce architecture descriptions." ISO/IEC/IEEE (2011)

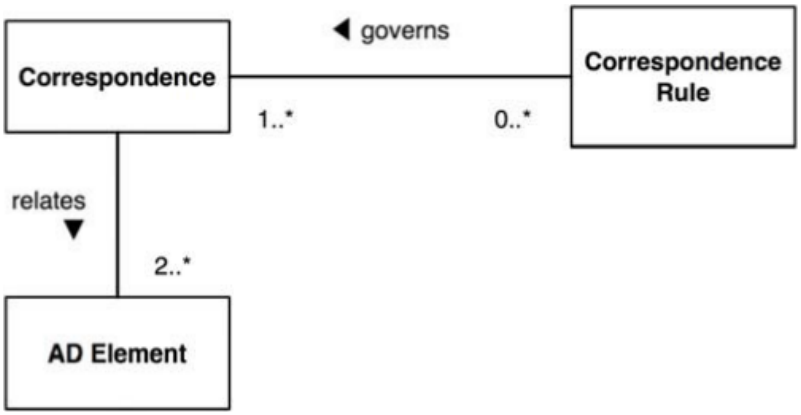

Figure B.2: Conceptual model of architectural description elements and correspondences
Source: (ISO/IEC/IEEE, 2011)

"An AD element is any construct in an architecture description. AD elements are the most primitive constructs discussed in this International Standard. Every stakeholder, concern, architecture viewpoint, architecture view, model kind, architecture model, architecture decision and rationale (see 4.2.7) is considered an AD element. When viewpoints and model kinds are defined and their models are populated, additional AD elements are introduced." ISO/IEC/IEEE (2011)

"A correspondence defines a relation between AD elements." ISO/IEC/IEEE (2011). "Correspondences and correspondence rules are used to express and enforce architecture relations such as composition, refinement, consistency, traceability, dependency, constraint and obligation." ISO/IEC/IEEE (2011)

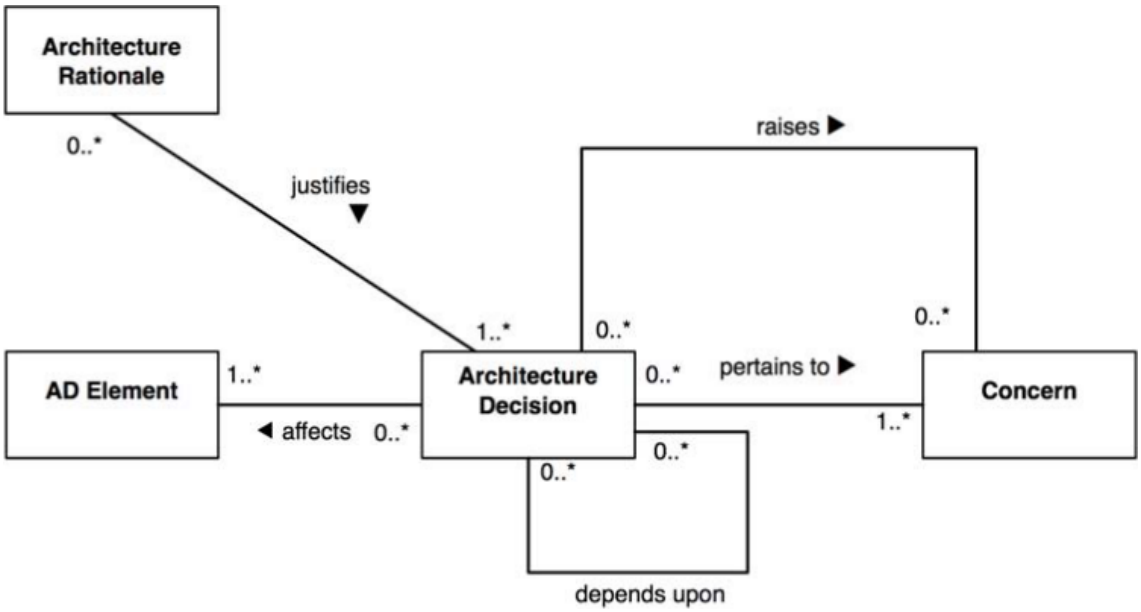

Figure B.3: Conceptual model of architectural decisions and rationale
Source: (ISO/IEC/IEEE, 2011)

"Architecture rationale records explanation, justification or reasoning about architecture decisions that have been made. The rationale for a decision can include the basis for a decision, alternatives and trade-offs considered, potential consequences of the decision and citations to sources of additional information." ISO/IEC/IEEE (2011)

"Decisions pertain to system concerns; however, there is often no simple mapping between the two. A decision can affect the architecture in several ways. These can be reflected in the architecture description as follows: requiring the existence of AD elements; changing the properties of AD elements; triggering trade-off analysis in which some AD elements, including other decisions and concerns, are revised; raising new concerns." ISO/IEC/IEEE (2011)

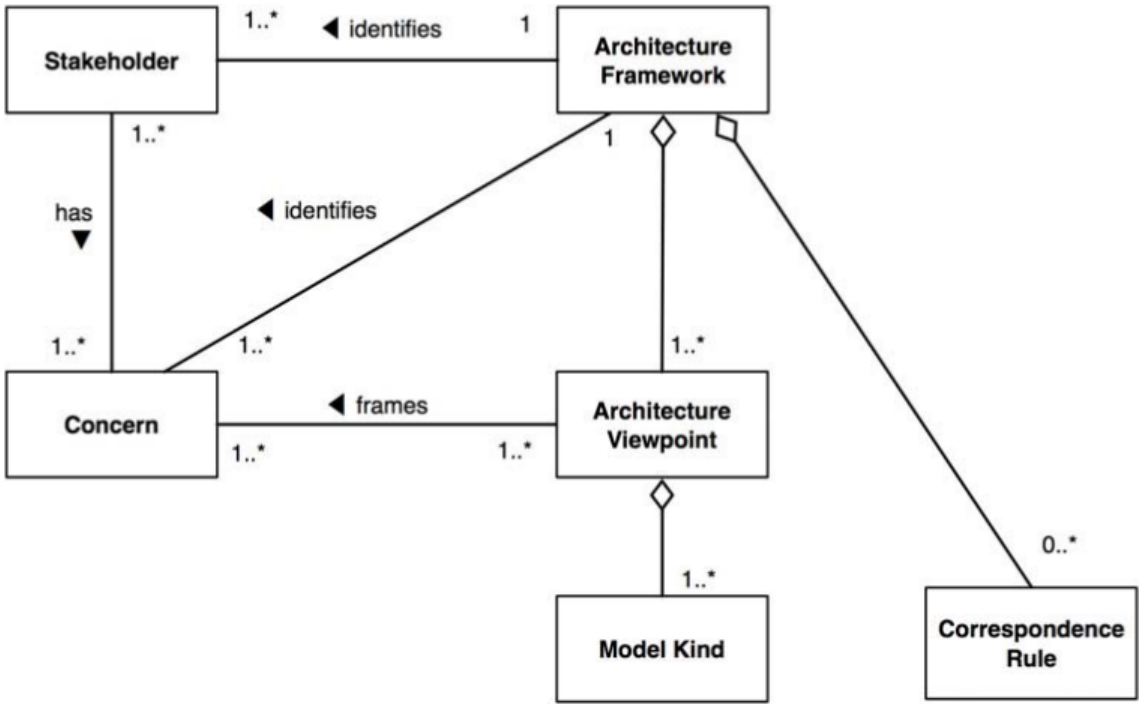

Figure B.4: Conceptual model of an architecture framework
Source: (ISO/IEC/IEEE, 2011)

"An architecture framework establishes a common practice for creating, interpreting, analyzing and using architecture descriptions within a particular domain of application or stakeholder community." ISO/IEC/IEEE (2011)

"A view is governed by its viewpoint: the viewpoint establishes the conventions for constructing, interpreting and analyzing the view to address concerns framed by that viewpoint. Viewpoint conventions can include languages, notations, model kinds, design rules, and/or modelling methods, analysis techniques and other operations on views." ISO/IEC/IEEE (2011)

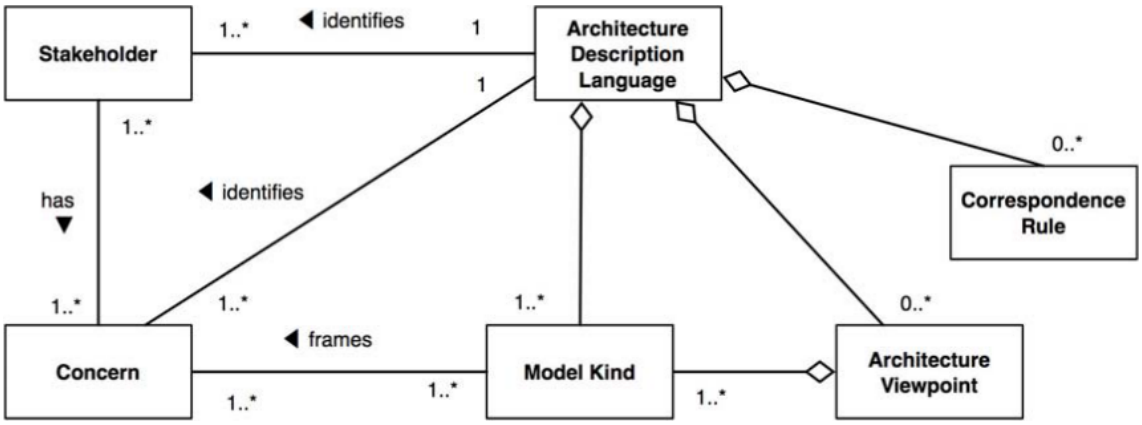

Figure B.5: Conceptual model of an architecture description language
Source: (ISO/IEC/IEEE, 2011)

"An architecture description language (ADL) is any form of expression for use in architecture descriptions." ISO/IEC/IEEE (2011). "An ADL provides one or more model kinds as a means to frame some concerns for its audience of stakeholders. An ADL can be narrowly focused, defining a single model kind, or widely focused to provide several model kinds, optionally organized into viewpoints. Often an ADL is supported by automated tools to aid the creation, use and analysis of its models." ISO/IEC/IEEE (2011)

Appendix C

ADvISE Meta-model

Here lies the meta-model for ADvISE.

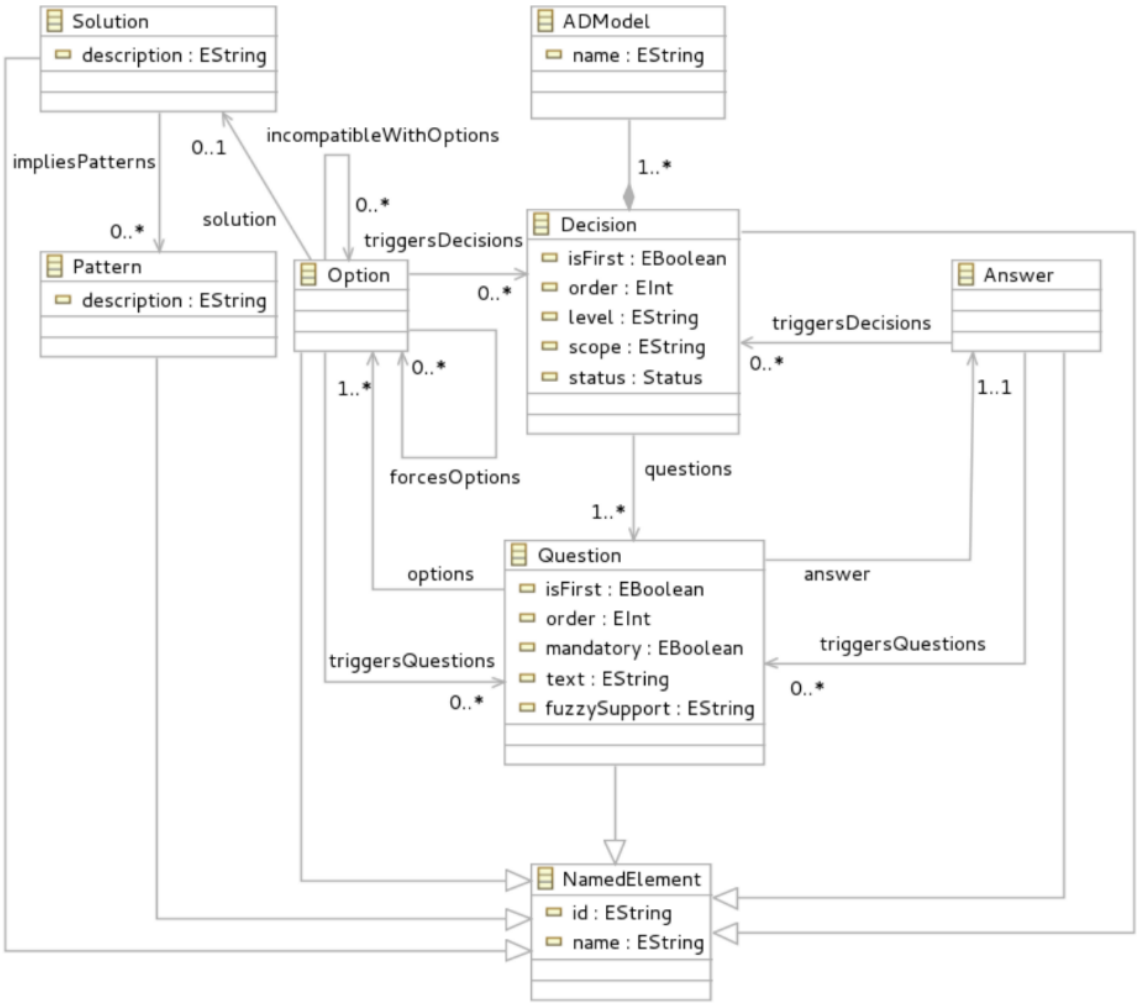

Figure C.1: ADvISE meta-model
Source: ADvISE official website[1]

Appendix D

CoCoADvISE Meta-model

Here lies the meta-model for CoCoADvISE.

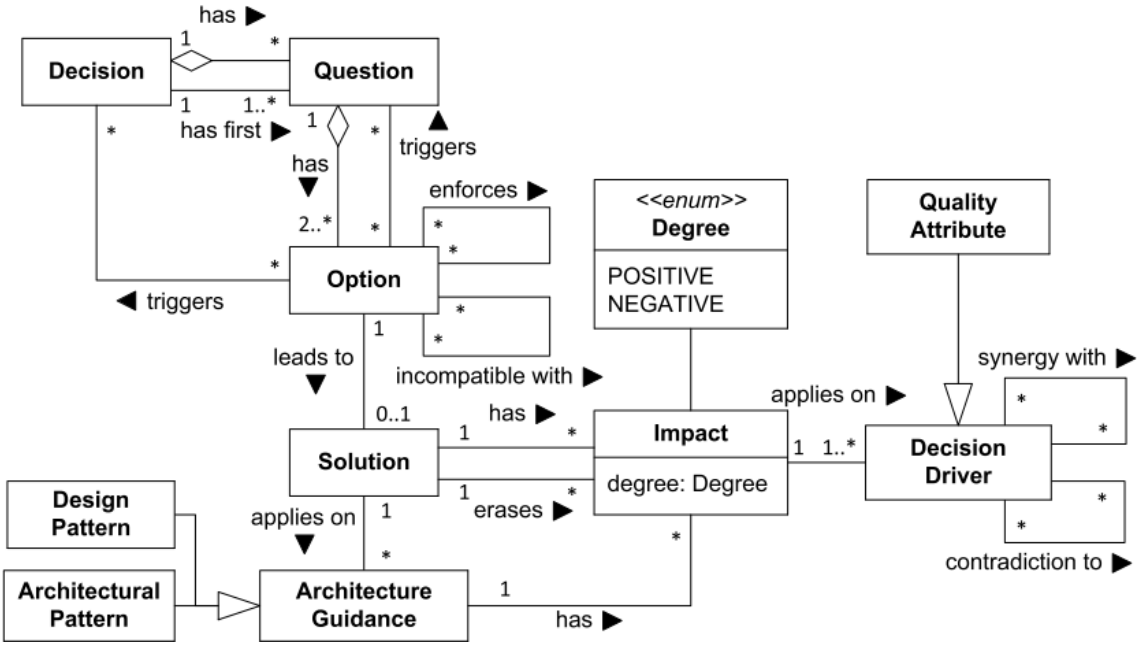

Figure D.1: CoCoADvISE meta-model
Source: (Lytra et al., 2015)

Appendix E

Architecture Knowledge Management Systems Features, and Strengths and Weaknesses

Here lies the table of features, and the current strengths and weaknesses of AKMSs.

E.1 Architecture Knowledge Management Systems Features

Table E.1: Knowledge Capture
Source: (Weinreich and Groher, 2016)

Knowledge Capture	**template-based**	PAKME, ADDSS, Decision Capture Tool, LISA, ADManager, A28, A44, A45
	schema-based	CORE, Tyree Template, Kruchten Ontology
	view-based	ADF, Decision Capture Tool, AR-diagram, TVM, A34
	annotations	Knowledge Architect, Decision Capture Tool, ArchiMind
	value-based	DDRD/DGA, CADDMS
	reuse-based	PAKME, ADDSS, RADM, ADMD3, ADUAK, SEURAT_Architecture, A26, ADvISE, A33, A34, A40, ArchPad, ADManager, ADMentor
	automation / generation	ABC/DD, STREAM-ADD, TopDocs, A36, Latent Semantic Analysis, A44, LISA
	recovery	NDR, TopDocs, A25, A26, A28, ADDRA, Latent Semantic Analysis, A45, A41, DVIA

Table E.2: Knowledge Application/Presentation
Source: (Weinreich and Groher, 2016)

Knowledge application / presentation	**visualization**	Knowledge Architect, LISA, Decision Capture Tool, NDR, Compendium, QuOnt, EA Anamnsesis, ShyWiki
	analysis	AREL, LISA, ArchiMind, Decision Capture Tool, NDR, Archium, AR-diagram, TopDocs, A46, AQUA, ADManager
	evaluation	TopDocs, QuOnt, PAKME, AREL, ADF, Knowledge Architect, Latent Semantic Analysis, AQUA, LISA, NDR, AR-diagram, A41
	decision-making	AREL, ADF, ABC/DD, DDRD/DGA, RADM, Shywiki, ADMD3, SEURAT_Architecture, A27, ADvISE, A33, ArchPad, EA anamnesis, Software Architecture Warehouse, A52, TDD/Decision Buddy, ISARCS

Table E.3: Knowledge Maintenance
Source: (Weinreich and Groher, 2016)

Knowledge maintenance	**history tracking**	PAKME, AREL, ADF, ADDSS, Decision Capture Tool, RADM, ShyWiki
	process	TVM, Decision Documentation Model
	transformation	ADvISE, AQUA

Table E.4: Knowledge Sharing
Source: (Weinreich and Groher, 2016)

Knowledge sharing	**central access**	PAKME, AK Sharing Portal, ADDSS, ADUAK, Software Architecture Warehouse, TDD/Decision Buddy, ISARCS, Knowledge Architect, ShyWiki, ArchiMind, RADM
	knowledge base/repository	TopDocs, A45, A52, NDR, A28, A25, Decision Capture Tool
	model-focused	MQPM, CORE, CADDMS

Table E.5: Knowledge Reuse
Source: (Weinreich and Groher, 2016)

Knowledge reuse	**generic/project-specific**	RADM, ADMentor, TDD/Decision Buddy
	pattern-based	ArchPad, A40, A33, ADvISE, ADUAK
	partial solutions	A36, A44, Archium

Table E.6: Technology
Source: (Weinreich and Groher, 2016)

Technology	**web-based**	AK Sharing Portal, PAKME, ADDSS, ADUAK, Software Architecture Warehouse, TDD/Decision Buddy, ISARCS, Knowledge Architect
	wiki-based	ShyWiki, ArchiMind, RADM
	Eclipse-based	LISA, SEURAT_Architecture, ADManager, ABC/DD, ADvISE, Decision Capture Tool
	UML-based	AREL, UML profile, EA Anamnesis, ADMentor
	DSL-based	Archium

E.2 Architecture Knowledge Management Systems Strengths and Weaknesses

Stakeholders	Internal Factors		External Factors		Solutions
	Positive	Negative	Positive	Negative	
End-users Business Managers	Share decisions	Don't trust the utility of AK	Reduce the budget for software maintenance	Lack of budget; No time for AK	Increase budget; Convince managers that maintenance will be more efficient and less costly overall
Software Architects	Share decisions; Understand evolution; Understand reasons; Identify critical decisions; Quality decisions; Understanding what to capture; Prevent knowledge vaporization	Lack of motivation; Afraid to share own expertise; Afraid to be challenged on decisions Effort to create extra trace links	Learn from other experts; Reuse AK from other projects	Lack of tools; Lack of time; Effort required	Convince software architects on the utility to document and use AK; Provide adequate tools; Adjust project schedule; Systematize the AK process; Provide adequate tools; Establish right design culture Convince software architects that additional trace links using design decisions will ease software maintenance and reduce maintenance effort
Software Maintainers	Understand the ripple effect; Identify toot causes of changes; Understand evolution and impact analysis	Effort to maintain extra trace links	Reduce the burden of maintenance	Lack of time; Effort required	Provide adequate tools Keep the size of the decisions network manageable

Figure E.1: Current Strengths and Weaknesses of AKMSs
Source: (Capilla et al., 2016)

Appendix F

Architecture Documentation stakeholders might find useful

Stakeholder	Module views				C&C views	Allocation views			Other					
	Decomposition	Uses	Generalization	Layers	Various	Deployment	Implementation	Work assignment	Interfaces	Contest diagrams	Mapping between views	Variability guides	Analysis results	Rationale and constraints
Project manager		x				x		x		x				x
Member of development team			x	x	x		x		x	x	x			x
Testers and integrators		x	x	x	x		x		x	x	x			
Designers of other systems									x	x				
Maintainers	x	x	x	x	x				x	x	x			x
Product line application builder		x		x	x				x	x	x	x		
Customer						x		x					x	
End user					x	x								
Analyst	x	x	x	x	x	x			x	x			x	x
New stakeholder	x	x	x	x	x	x	x	x	x	x	x	x	x	x
Current and future architect	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Figure F.1: Architecture documentation stakeholders might find useful
Source: (Clements et al., 2002)

Appendix G

Data Model Data Definition Language

```sql
CREATE TABLE Owner (
  id INTEGER PRIMARY KEY,
  login TEXT NOT NULL,
  html_url TEXT NOT NULL,
  avatar_url TEXT,
  gravatar_id INTEGER,
  type TEXT NOT NULL
);

CREATE TABLE Repository (
  id INTEGER PRIMARY KEY,
  id_owner INTEGER NOT NULL REFERENCES Owner(id),
  name TEXT NOT NULL,
  full_name TEXT NOT NULL,
  html_url TEXT NOT NULL,
  homepage TEXT,
  description TEXT,
  fork BOOLEAN NOT NULL,
  private BOOLEAN NOT NULL,
  stargazers_count INTEGER NOT NULL,
  watchers_count INTEGER NOT NULL,
  language TEXT
);

CREATE TABLE Release (
  id INTEGER PRIMARY KEY,
  name TEXT NOT NULL,
  html_url TEXT NOT NULL,
  tag_name TEXT NOT NULL,
  body TEXT NOT NULL,
  target_commitish TEXT NOT NULL,
  commit_id TEXT NOT NULL,
  published_at TIMESTAMP WITH TIME ZONE NOT NULL
);

CREATE TABLE Featurewise (
  id SERIAL PRIMARY KEY,
  id_repository INTEGER NOT NULL REFERENCES Repository(id),
  id_release INTEGER UNIQUE REFERENCES Release(id),
  domains_hash TEXT NOT NULL,
  head_branch TEXT,
```

```
  head_sha TEXT
);

CREATE TABLE Schema (
  id SERIAL PRIMARY KEY,
  id_repository INTEGER REFERENCES Repository(id),
  commit_id TEXT,
  file_path TEXT,
  version INTEGER,
  external_url TEXT,
  data JSONB NOT NULL,
  data_hash TEXT NOT NULL,
  created_at TIMESTAMP WITH TIME ZONE
);

CREATE TABLE Domain (
  id SERIAL PRIMARY KEY,
  id_featurewise INTEGER NOT NULL REFERENCES Featurewise(id),
  id_schema INTEGER REFERENCES Schema(id),
  data JSONB NOT NULL,
  data_hash TEXT NOT NULL
);
```

Listing 23: Data Description Language

Appendix H

Docker files

```
version: '3.7'

services:
  backend:
    build:
      context: ../../
      dockerfile: docker/prod/backend.Dockerfile
    networks:
      - app-network
      - datastore-network
    environment:
      - DEBUG
      - NODE_ENV
      - SESSION_SECRET
      - SESSION_COOKIE_PATH
      - SESSION_COOKIE_DOMAIN
      - SESSION_REDIS_URL
      - BACKEND_PORT
      - DATASTORES_POSTGRES_URL
    depends_on:
      - postgres
      - redis
    labels:
      - "traefik.enable=true"
      - "traefik.docker.network=app-network"
      - "traefik.backend=backend"
      - "traefik.basic.frontend.rule=Host:${TRAEFIK_BACKEND_HOST}"
      - "traefik.basic.port=${BACKEND_PORT}"
      - "traefik.basic.protocol=http"
      - "traefik.port=80"
    restart: always
  github-app:
    build:
      context: ../../
      dockerfile: docker/prod/github-app.Dockerfile
    networks:
      - app-network
      - datastore-network
    environment:
      - PORT=${GITHUB_APP_PORT}
      - DEBUG
```

```
      - NODE_ENV
      - APP_ID
      - PRIVATE_KEY_PATH
      - WEBHOOK_PROXY_URL
      - WEBHOOK_SECRET
      - FEATUREWISE_REDIS_URL
      - LOG_LEVEL
      - DATASTORES_POSTGRES_URL
    depends_on:
      - postgres
      - redis
    labels:
      - "traefik.enable=true"
      - "traefik.docker.network=app-network"
      - "traefik.backend=github-app"
      - "traefik.basic.frontend.rule=Host:${TRAEFIK_GITHUB_APP_HOST}"
      - "traefik.basic.port=${GITHUB_APP_PORT}"
      - "traefik.basic.protocol=http"
      - "traefik.port=80"
    restart: always
  postgres:
    build:
      context: ../../datastores/
      dockerfile: ../docker/prod/postgres.Dockerfile
    networks:
      - datastore-network
    container_name: ${POSTGRES_CONTAINER_NAME}
    environment:
      - POSTGRES_USER
      - POSTGRES_PASSWORD
      - POSTGRES_DB
      - PGDATA=/var/lib/postgresql/data/pgdata
    volumes:
      - postgres-data:/var/lib/postgresql/data/pgdata
    restart: always
  redis:
    image: redis
    container_name: ${REDIS_CONTAINER_NAME}
    networks:
      - datastore-network
    restart: always

networks:
  app-network:
    external: true
  datastore-network:

volumes:
  postgres-data:
    driver: local
```

Listing 24: docker-compose.yml for prod

```
version: '3.7'

services:
  traefik:
```

```
    image: traefik
    command: --api
    ports:
      - 80:80
      - 8080:8080
    networks:
      - app-network
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock
      - ./traefik.toml:/traefik.toml
    container_name: traefik
    restart: always

networks:
  app-network:
    external: true
```

Listing 25: docker-compose.yml for traefik

```
version: '3.7'

services:
  postgres:
    build:
      context: ../../datastores/
      dockerfile: ../docker/prod/postgres.Dockerfile
    networks:
      - app-network
    container_name: ${POSTGRES_CONTAINER_NAME}
    environment:
      - POSTGRES_USER
      - POSTGRES_PASSWORD
      - POSTGRES_DB
      - PGDATA=/var/lib/postgresql/data/pgdata
    ports:
      - 5432:5432
    restart: always
  redis:
    image: redis
    container_name: ${REDIS_CONTAINER_NAME}
    networks:
      - app-network
    ports:
      - 6379:6379
    restart: always

networks:
  app-network:
    driver: bridge

volumes:
  postgres-data:
    driver: local
```

Listing 26: docker-compose.yml for dev