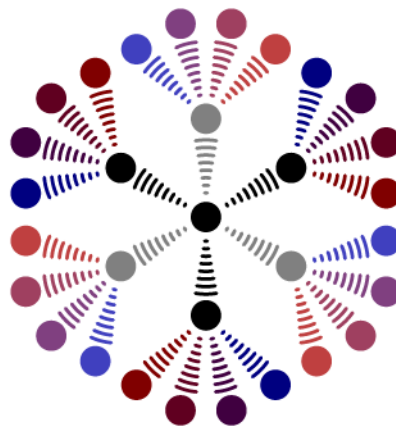


# THE KNOWLEDGE CLOUD

— Evolving the data management paradigm —




— PROJECT KLOWID —

WANNO DRIJFHOUT

Department  
Formal Methods and Tools

Faculty  
Electrical Engineering, Mathematics and Computer Science (EEMCS)

UNIVERSITY OF TWENTE.



The Knowledge Cloud — Evolving the data management paradigm

by

WANNO DRIJFHOUT

*A thesis submitted in fulfilment of the requirements  
for the degree of Master of Science*

in the  
department Formal Methods and Tools,  
faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS),  
at the University of Twente, The Netherlands

#### SUPERVISORS

dr.ir. MAURICE VAN KEULEN  
prof.dr.ir. AREND RENSINK  
dr.ing. CHRISTOPH BOCKISCH

(Databases)  
(Formal Methods and Tools)

#### PROJECT

Time frame: 3 July 2013 to 30 April 2015  
Study: Computer Science – Software Engineering

Resources for this project are available on-line at <http://klowid.eu/>.

*Document version 2015-04-29*



## SAMENVATTING

Vandaag de dag bepaalt software hoe we onze gegevens (mogen) opslaan. De programmatuur bepaalt welke bestandsindelingen, welke *database management systemen* of welke *cloud*-diensten worden ondersteund. Gegevens zijn lastig te delen tussen programma's of apparaten: bestanden zijn ongeschikt (te primitief); cloud-diensten zijn programmaspecifiek of kunnen gegevens niet (of slecht) uitwisselen met concurrerende diensten. Dientengevolge worden de gegevens van de gebruiker versplinterd over diverse opslagmechanismen.

Programma's zijn onnodig ingewikkeld; gegevens zijn slecht uitwisselbaar; de gebruikers hebben geen controle over hun gegevens; ontwikkelaars schrijven onnodige code. Ons hedendaagse 'gegevensbeheerparadigma' veroorzaakt deze problemen door een sterke koppeling van programma's met opslag. Wij stellen een nieuw gegevensbeheerparadigma voor om deze problemen op te lossen.

Ons nieuwe gegevensbeheerparadigma scheidt programma's en opslag met een centraal systeem voor informatievoorziening en -uitwisseling: de *kenniswolk* [knowledge cloud]. Programma's zijn niet langer bezorgd met opslag maar wisselen informatie uit met de kenniswolk met hetzelfde informatiemodel als het programma intern gebruikt—deze kwaliteit wordt *opslagonwetendheid* [storage-agnosticism] genoemd. De gebruiker kan de kenniswolk instellen, zonder medeweten van programma's, om de opslag en verwerking van zijn gegevens te sturen—deze kwaliteit wordt *omstandigheidsaanpasselijkheid* [context-adaptivity] genoemd.

De onderzoeksbijdrage bestaat uit een achtergrondstudie, een onderbouwd voorstel voor de kenniswolk, een ontwerp van diens architectuur, een implementatie van een kenniswolk-prototype met demonstratieprogramma's en een theoretische *case-study*. Deze resultaten geven antwoord op de hoofdvraag van het onderzoek:

„Wat zijn de gevolgen voor gebruikers en ontwikkelaars als programma's met een enkele kenniswolk werken om informatie op te halen en op te slaan, in plaats van met diverse afzonderlijke opslagmechanismen?”

Dankzij het gebruik van een kenniswolk, *kunnen* programma's minder ingewikkeld worden; *worden* gegevens beter uitwisselbaar; *krijgen* gebruikers betere beheersing van hun gegevens; *zouden* ontwikkelaars minder onnodige programmatuur moeten schrijven. Aanvullend onderzoek is noodzakelijk om onze resultaten te valideren in praktischere omstandigheden met meer complexiteit maar de eerste bevindingen zijn bemoedigend.



## ABSTRACT

In the state of the art, software programs decide how data is (or may be) stored. The program's code determines in which file formats, which database management systems or which cloud services are supported. Sharing data between programs or devices is difficult: files are inadequate (too low-level); cloud services tend to be program-specific or interoperate badly with competing services. Consequentially, the user's data is splintered across various storage mechanisms.

Programs are unnecessarily complex; data is badly interoperable; users are not in control of their data; developers must write inessential code. Our current 'data management paradigm' causes these problems by promoting a strong coupling between programs and storage. We propose a new data management paradigm to solve these problems.

Our new data management paradigm separates programs and storage by a central system for information provision and exchange: the *knowledge cloud*. Programs are no longer concerned with storage; they exchange information with the knowledge cloud in the same information model the program uses internally—we call this quality *storage-agnosticism*. The user can configure the knowledge cloud, oblivious to the programs, in order to control storage and processing of the user's data—we call this quality *context-adaptivity*.

Our research contribution includes a background study, a substantiated proposal for the knowledge cloud, a design of its architecture, an implementation of a knowledge cloud prototype with demonstration programs and a theoretical case study. These efforts permit us to answer the main question:

"What are the effects for users and developers if programs interacted with a single knowledge cloud for information access and storage, rather than with various storage mechanisms separately?"

By using a knowledge cloud: programs *may* be less complex; data *is* more interoperable; users *have* more control of their data; developers *should* write less inessential code. More research is necessary to validate our results in more practical conditions with more complexity, but our initial findings inspire confidence.



## PREFACE

Enschede, April 2015

Dear reader,

For years I struggled with a problem, a challenge, an annoyance—one that seemed *so obvious* that it just begged for a solution: programs didn't like my file organisation. My attempts to switch operating systems, switch browsers, switch mail clients all failed—all because the alternatives stored data in a completely different way, often hiding it in complex file formats and folder hierarchies or databases. I could not share, access or organise my data the way I wanted. *My data!* I was dismayed by the state of affairs but I had to accept it. Giants that were (and still are) far more experienced and knowledgeable than I didn't work on The Ultimate Solution, so there probably couldn't be any.

While studying Computer Science (specialisation Software Engineering) or exploring the Web, I occasionally stumbled upon some neat principle, some well-reasoned design or some weird powerful technology that made me wonder if these could help solve my problem. As small insights and ideas came to me, I chose to write them down and to bide my time for an eventual confrontation with my problem. The master thesis would be that confrontation.

The final project gave me the *opporturdy* (opportunity and burden) to seek out new knowledge and solutions; my problem definitely required those. In June 2012, six months before the final project would actually be relevant, I pitched my problem to Christoph Bockisch and he concurred that my problem seemed worthy of further investigation as a research topic.

When I started working on this problem as a research project, I had six pages of notes and scribbles. Since then, much time has passed and scribbles turned into research results, documents, source code and many new insights. I do not know what the past few years would have looked like, if I had not pursued my research idea or if I had not jolted down small ideas and insights over the years. No matter, though; I am actually quite content with reality as it is.

This conclusion to my research project would not have been possible without the help of many people.

I would like to thank my research supervisors: Christoph Bockisch, Maurice van Keulen and Arend Rensink. Their own enthusiasm for my research topic, their insights and their complementary expertises were of tremendous value. (Their patience is also greatly appreciated.) I also want to thank my many friends, especially those I got to know at the I.A.P.C. foundation and S.H.B.V. Sagittarius. Of course, my gratefulness extends to my family.

Thank you for your interest in my work!

Wanno Drijfhout



# CONTENTS

1	INTRODUCTION	1
1.1	Motivation & Goal	2
1.2	The new data management paradigm	2
1.2.1	The knowledge cloud	3
1.2.2	Fundamental Qualities	3
1.3	Research questions	4
1.4	Contributions	4
1.5	Approach	4
1.6	Guiding example: managing browser hyperlinks	5
2	BACKGROUND	7
2.1	Storage-related concerns	7
2.1.1	Data formats	8
2.1.2	Storage interfaces	8
2.1.3	Consistency guarantees	8
2.2	Program-related concerns	9
2.2.1	Program architecture	9
2.2.2	Impedance mismatch	9
2.2.3	Interoperability	10
2.3	Cloud-related concerns	10
2.3.1	Knowledge models	10
2.3.2	Model interchange	11
2.3.3	Model transformations	12
2.3.4	Federated databases and canonical models	12
2.4	Model structures	13
2.4.1	Hierarchical	13
2.4.2	Network	14
2.4.3	Object-Oriented	14
2.4.4	Relational	14
2.4.5	Object-Relational	15
2.4.6	Entity-Relationship	15
2.4.7	Functional	16
2.5	Related work	16
2.5.1	Semantic web	16
2.5.2	Microsoft Windows Future Storage (WinFS)	17
2.5.3	Horde	17
2.5.4	OpenLink Virtuoso Universal Server	18
2.5.5	Amos II	18
3	SOLUTION	19

3.1	Reason for the knowledge cloud	19
3.1.1	Model abstractions: data, information, knowledge	20
3.1.2	Model perspective: internal, external, conceptual	20
3.1.3	Model applications: storage, program, cloud	21
3.2	Qualities of the knowledge cloud	22
3.2.1	Storage-agnosticism	22
3.2.2	Context-adaptivity	22
3.2.3	Benefits to data management	23
3.3	Functional requirements	24
3.4	Consequences of the new paradigm	26
3.4.1	Programs	26
3.4.2	Data	26
3.4.3	Users	27
3.4.4	Developers	27
4	DESIGN	29
4.1	Abstractions	29
4.2	Architecture	30
4.2.1	A client asks for knowledge in a domain	30
4.2.2	The cloud provides knowledge in a domain	32
4.3	Concepts	33
4.3.1	Knowledges, questions and answers (green)	33
4.3.2	Domains, traits and specifications (blue)	35
4.3.3	Feeds, offers, networks and orchestrator (orange)	38
4.3.4	Agents and bureaus (purple)	40
4.3.5	Clouds (black)	41
5	VALIDATION	45
5.1	Demonstration: a prototype	45
5.1.1	Purpose	45
5.1.2	Scope	46
5.1.3	Method & Implementation	46
5.1.4	Discussion	49
5.2	Theoretical case-study: Horde	51
5.2.1	Method	51
5.2.2	Results	52
5.2.3	Discussion	55
5.3	Results	55
5.3.1	Programs may be less complex	55
5.3.2	Data is more interoperable	56
5.3.3	Users have more control of their data	57
5.3.4	Developers should write less inessential code	57
5.3.5	Conclusion	58
6	CONCLUSION	59
6.1	Main research findings	60
6.2	Future work & Recommendations	60

NON-SCIENTIFIC REFERENCES	63
BIBLIOGRAPHY	65
A VALIDATION	71
A.1 Jena transformation file	71
A.2 RDF models	72
A.2.1 Source RDF-model	72
A.2.2 Inferred RDF-model	73





# 1

## INTRODUCTION

In the past decades, technology has evolved at an unprecedented pace. To manage a particular piece of information (like calendars and documents), we may have once used one device and one application. Today, we want to manage the same information at different devices (such as smart phones, tablets, laptops and desktop PCs). So, applications need *some* mechanism to store and access this structured (distributed) information. We notice that software products tend to select these mechanisms and thereby dictate how we (the users) may use, model, store and supply data. We find this problematic and unnecessary.

Software programs control data storage.

**TO ILLUSTRATE:** Mozilla Firefox uses bookmarks while Microsoft Internet Explorer uses favorites; synchronizing between these applications or across devices is all but impossible (as will be further discussed in section §1.6). Software that *does* offer synchronization features may only support a limited selection of protocols; for example: to exchange calendar appointments, Microsoft Outlook prefers Microsoft Exchange while Mozilla Thunderbird (with extensions) uses CalDAV.

To simply enable client applications to access and share some data (hyperlink collections, calendars), we need to manage various centralized storage servers or use some proprietary cloud service. And we may *still* find client applications we want to use that do not support that particular storage mechanism.

Sharing data is difficult.

**TRADITIONAL STORAGE** The most common storage mechanism is the hierarchical file system—originally invented for the Multics operating system in 1965 [14]. Its prime construct (the file segment) is essentially a raw byte array without predefined structure. Most applications will expect their files to be *formatted* in some fashion. Formats may be standardized (XML, JSON) and readable by different programs; alternatively, formats may be proprietary or human-unreadable.

Files lack structure.

To meet the modern need of sharing data between devices, various synchronization utilities (Dropbox, Microsoft OneDrive) and remote file access protocols (SMB, WebDAV) exist. However, many programs expect to be solely responsible for file management and ignore concurrent file modifications or locks by external parties. Applications may also mutate files and folders so quickly that synchronization utilities go haywire and possibly corrupt files. Moreover, one cannot easily share *fragments* of a (changing) file; if only because the lack of inherent structure makes the selection and recombination of relevant file fragments difficult.

Sharing data as files is inadequate.

**MODERN STORAGE** We notice a trend for modern applications to support data sharing and synchronization by storing user data ‘in the cloud’. In practice, this cloud is some proprietary (possibly application-specific) internet service to manage data over which the user has no control. Some cloud-like service software

Cloud services control data storage.

products may be run on a home server, but these are rare and often functionally-limited.

User data is splintered, hindering interoperability.

Paradoxically, to maintain and access a single state of *their* data with *their* devices, users need to yield control over their data and devices to *foreign* applications and services. With their data now splintered across various application-specific services on different machines with proprietary APIs, data interoperability may be hindered as well. For programs to access the user's data, developers now need to implement support and maintain *more* storage-related code (server APIs, synchronization logic, account management) in their applications. That functionality bloats programs and increases their complexity.

## 1.1 MOTIVATION & GOAL

The current data management paradigm couples programs and storage.

A 'data management paradigm' is a set of *concepts* and *thought patterns* on data management. Today, programs need to explicitly support particular storage systems. Assigning the responsibility for storage to applications is the fundamental characteristic of our data management paradigm that makes it *primitive*. This paradigm was adequate in times when people used (or even shared) one device to manage all their data. Nowadays, people use different devices (with possibly different applications) to manage their data. The traditional paradigm cannot serve modern demands and causes various problems:

- *Programs are unnecessarily complex* because they must support and abstract different storage mechanisms.
- *Data is badly interoperable* because it is stored to conform exactly to some program's (possibly unique) format.
- *Users are not in control of their data* because programs decide how data is used and stored.
- *Developers must write inessential code* to implement, support or abstract from storage services.

Clearly, both users and developers stand to benefit from a data management paradigm that ameliorates these problems. The goal of this research project is to present and substantiate our vision for such a new data management paradigm.

## 1.2 THE NEW DATA MANAGEMENT PARADIGM

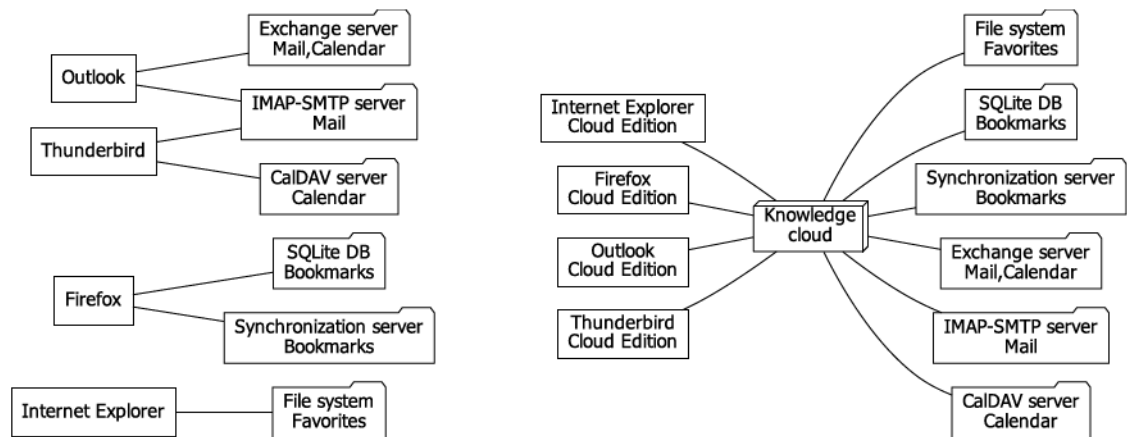
The proposed data management paradigm separates programs and storage.

The proposed new data management paradigm contributes to solving the four identified problems by moving the responsibility for storage from programs to a central information provision and exchange service: the *knowledge cloud*.

### 1.2.1 The knowledge cloud

We imagine that each device would run a knowledge cloud instance, controlled and configured by the user. Where applications would traditionally access the file system or network sockets to access storage, they would now contact the knowledge cloud instead, as shown in figure 1. Programs exchange information *in to their own model* with this intermediary system and delegate other responsibilities (such as data storage) to this intermediary system. Applying this paradigm requires programs to be structured fundamentally differently; interaction with the knowledge cloud is not imagined as an additional feature to existing programs.

Client programs exchange information with a knowledge cloud in their own model.



**Figure 1:** The effect of the current (left) and proposed (right) data management paradigms on program-storage-interaction.

### 1.2.2 Fundamental Qualities

The knowledge cloud is a generic abstraction layer between programs and storage mechanisms and must be flexible enough to use various storage mechanisms in various circumstances. The knowledge cloud enables two fundamental qualities of the new paradigm: *storage-agnosticism* and *context-adaptivity* (as will be described in more detail by section §3.2 on page 22).

The knowledge cloud enables storage-agnosticism and context-adaptivity.

**storage-agnosticism** permits users and programs to remain ignorant of *storage concerns*. Rather than interpreting data (in some format from some data source), a storage-agnostic program specifies an information model and expects the information from the knowledge cloud to conform.

**context-adaptivity** permits other parties to invisibly manipulate the interaction between programs and storage. By interception and adaptation, other parties (contexts) could support *cross-cutting concerns* (authorization, data transformation) without the program noticing.

### 1.3 RESEARCH QUESTIONS

Our research project intends to answer the following main question:

**Q0** “What are the effects for users and developers if programs interacted with a single knowledge cloud for information access and storage, rather than with various storage mechanisms separately?”

In particular, we consider the following research questions:

- Q1** Can the use of a knowledge cloud reduce program complexity?
- Q2** Can the use of a knowledge cloud improve data interoperability?
- Q3** Can the use of a knowledge cloud improve control by users over their data?
- Q4** Can the use of a knowledge cloud reduce the need for inessential code?

### 1.4 CONTRIBUTIONS

We present a new vision on data management.

With this research project, we contribute to a fundamental discussion on data management paradigms, their weaknesses and qualities. We describe the state of the art paradigm and identify its fundamental weakness: assigning responsibility for storage to applications. We also present an alternative paradigm that separates storage from application and introduce the ‘knowledge cloud’ to realize this separation. We define two fundamental qualities this new paradigm would have due to having a knowledge cloud: storage-agnosticism and context-adaptivity. We substantiate our vision by elaborating on the reason and design of a knowledge cloud and implementing a prototype knowledge cloud (*Project Klowid*). A case-study further illustrates the supposed effects of a knowledge cloud on software.

### 1.5 APPROACH

This document describes our efforts to find and validate answers to the research questions, which include:

- section §1.6 • specifying a example use-case of the knowledge cloud;
- chapter 2 • investigating concerns of programs, storage and the knowledge cloud;
- chapter 3 • establishing what requirements and benefits a knowledge cloud should have;
- chapter 4 • designing a general architecture for a knowledge cloud;
- section §5.1 • implementing a knowledge cloud prototype with demonstration programs;
- section §5.2 • simulating the effects of using a knowledge cloud on an existing program;
- chapter 6 • concluding with our findings and recommendations for future research.

## 1.6 GUIDING EXAMPLE: MANAGING BROWSER HYPERLINKS

In this document we introduce and discuss various abstract concepts. To ease understanding and discussion, one imaginative problem case will guide us throughout this document: the ‘web browser’ problem we mentioned at the very beginning.

All modern web browsers have some feature for users to maintain a collection of hyperlinks. Different browsers support different organization models and storage back-ends. Microsoft Internet Explorer calls remembered hyperlinks ‘Favorites’ and organizes them hierarchically. Mozilla Firefox maintains an hierarchy of ‘Bookmarks’ and allows each bookmark to be ‘tagged’ by multiple categories.

This problem case exhibits the problems we introduced in section §1.1 as follows.

**PROGRAMS HAVE UNNECESSARY COMPLEXITY** Internet Explorer accesses and observes the file system to maintain an up-to-date hierarchy of favorites. Firefox needs to access a database and optionally maintain a synchronisation server connection. The browsers implement functionality of non-trivial complexity (e.g., parsing, maintaining connections, handling I/O and connection failures) to simply maintain correspondence between data storage and their in-memory information model.

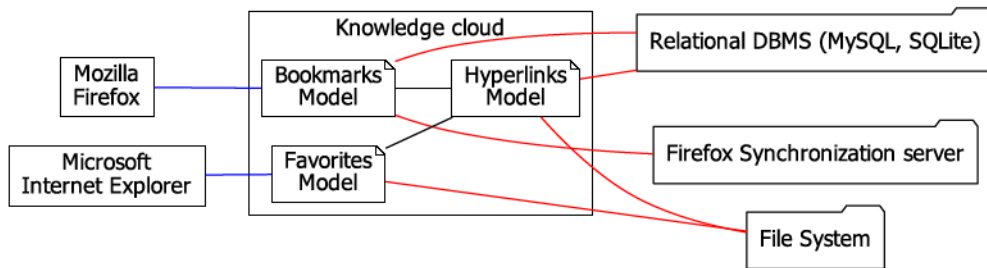
**DATA IS BADLY INTEROPERABLE** Neither browser in question is able to use hyperlink collections of the other: Firefox does not use favorites; Internet Explorer does not use bookmarks. From the user’s perspective: both browsers dictate how hyperlink collections are modelled, stored and used—i.e., users suffer vendor lock-in. For other applications and users to access bookmarks and favorites, they would still need to support each browser-specific data format and storage mechanism.

Internet Explorer and Firefox persist bookmarks somewhere in the user’s personal directory on the local file system. Internet Explorer uses a folder hierarchy of hyperlink files with an obscure data format. Firefox uses a SQLite file in an obscure location, which is difficult to access. While Internet Explorer *favorites* may be synchronized across devices as ordinary files, their presentation order would be lost [a]. To synchronize Firefox *bookmarks*, one would need to run a server application [b] or submit their data to a vendor-run service.

**USERS DO NOT CONTROL DATA** Internet Explorer saves favorites in a hierarchy of files and folders on the file system; Firefox uses a SQLite-database. The user has no control over these data formats and storage mechanisms.

**DEVELOPERS WRITE INESSENTIAL CODE** Browser developers write and maintain the code for the aforementioned complexity. *Other* programs may need to also access favorites or bookmarks; *their* developers must first determine *how* and may end up implementing similar code *again*.

**SOLUTION: THE KNOWLEDGE CLOUD** Currently, both browsers are concerned with data storage and thereby hinder data interoperability and user control. The browsers could support all sorts of sharing, synchronization and exchange features, but this would only increase the user's reliance on the browser, bloat the code base etc. In our proposed paradigm, browsers would connect their information model to the knowledge cloud and would no longer be concerned with data formats and storage. Instead, the (user-controlled) knowledge cloud provides the information browsers need, transforms between models if needed and manages the storage of information, as shown in figure 2.



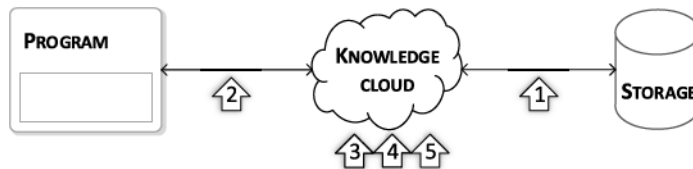
**Figure 2:** Simplified model of a possible configuration of the knowledge cloud for guiding example (section §1.6). Note-shaped nodes denote information; edges between them denote possible transformations. Folder-shaped nodes denote storage mechanisms that are accessed by the cloud to bring external data into the cloud for further processing (shown with red edges).



## 2 | BACKGROUND

We proposed (chapter 1) a new data management paradigm in which programs are *not* responsible for data storage—a *knowledge cloud* would be instead. A knowledge cloud that assumes responsibility of storage for arbitrary programs and arbitrary storage mechanisms needs to handle various concerns. This chapter elaborates on relevant concerns, grouped by five aspects of the new data management paradigm (as shown in figure 3):

- §1. **Storage-related concerns:** data formats, storage interfaces, consistency guarantees;
- §2. **Program-related concerns:** storage abstraction, impedance mismatch, interoperability;
- §3. **Cloud-related concerns:** model interchange, federations, canonical models, knowledge models, existing solutions;
- §4. **Model structures:** the features and (dis)advantages of relational, object-oriented and other models;
- §5. **Related work:** existing knowledge-cloud-like systems.



**Figure 3:** The aspects of the knowledge cloud: (1) between cloud and storage, (2) between program and cloud and (3, 4, 5) inside the cloud.

### 2.1 STORAGE-RELATED CONCERNS

In the current data management paradigm, programs are concerned with accessing storage and they may do so in different ways. In the proposed paradigm, the knowledge cloud is expected to access storage, instead. We consider some common storage-related concerns a knowledge cloud would need to consider.

### 2.1.1 Data formats

Data formats impose structure on raw data.

Data formats define the structure with which data is described and (de)serialized. A data format is defined by a (parsable) grammar and constraints (e.g., encoding, validation).

Nowadays, programs tend to use a general-purpose data format such as XML, YAML or JSON. Generally, such formats model information hierarchically (section §2.4.1). XML and YAML support cross-referencing, making them fit more easily to object-oriented information models (section §2.4.3) as well. Before the advent of general-purpose data formats, programs defined custom file formats with custom parsers. Some programs (e.g., Microsoft Word) even deprecated their custom data formats (DOC) in favour of XML-based successor formats (DOCX). By using a general-purpose data format, programs can also reuse common libraries (e.g., XML-parsers) to abstract from raw data (cf. data independence [12]).

Database management systems tend to use custom binary (i.e., non-textual) formats for their databases. Database formats are optimized for being used with a DBMS (e.g., PostgreSQL) or library (e.g., SQLite).

### 2.1.2 Storage interfaces

Programs already avoid accessing storage directly.

Programs access storage via some programmatic interface, often in conjunction with some query language, protocol or library.

SQL [10] is a well-known declarative language for querying and mutating data in a relational database (section §2.4.4). Most relational database management systems (RDBMS) can interpret SQL-queries and proprietary extensions thereto. Although SQL is standardized, programs must still explicitly support the interface of particular RDBMSs.

The Open Database Connectivity standard (ODBC) aims for database independence by offering a generic database interface to client programs. It was originally developed by Microsoft in the 1990's and is supported by many programs in its Office-suite. It seems not to be used often by modern applications.

Libraries (such as JDBC [c]) can abstract from particular database management systems (cf. database-agnosticism [d]), facilitate query construction through fluent APIs [e] and (de)serialize query results into program entities. Some programming languages facilitate storage access through first-class language constructs (such as .NET LINQ [f]).

### 2.1.3 Consistency guarantees

Relational database management systems traditionally support transactions (a unit of work on the database). Reliable transaction systems satisfy the ACID-properties [25, 27]:

**atomicity** Either the transaction fails (without a trace) or succeeds (with full results).

**consistency** Committed transactions only have legal (valid) results.



**isolation** Effects of transactions are hidden from concurrently executed transactions.

**durability** Once the transaction is committed, its results are guaranteed to survive calamity.

Relational DBMSs generally support transactions and allow clients to specify how strictly the ACID-properties should be enforced due to a performance trade-off.

Relatively recently, NoSQL systems [9] emerged as alternatives to the traditional RDBMS. The class of NoSQL-systems encompasses storage systems with non-relational data models (such as the key-value store, a graph or a document). Distributed, big data and real-time web applications tend to use NoSQL-systems. Compared to traditional and well-researched RDBMSs, NoSQL-systems suffer a number of weaknesses hindering their widespread adoption [25]. Distributed NoSQL-systems generally sacrifice *consistency* for the sake of *availability* and *partition tolerance* (as defined by Brewer's theorem [24]). Most NoSQL-systems do not guarantee ACID-properties but may guarantee the weaker BASE-properties (Basically Available, Soft state, Eventual consistency [45, 3]).

Consistency and performance must be traded off.

Different storage mechanisms offer different consistency guarantees.

## 2.2 PROGRAM-RELATED CONCERNS

Programs use an information model of some domain. Programs concerned with storage must also maintain the relation between this information model and the storage (data) model. In the proposed paradigm, the knowledge cloud would assume or affect such responsibilities, like the following.

### 2.2.1 Program architecture

Most programs abstract from storage by using libraries, language constructs and/or applying some architectural pattern.

The traditional three-tier architecture [h] distinguishes a presentation tier (with the UI), an application tier (with business logic and an information model) and a data tier (with the data model and storage interfacing). To abstract the relation between information model and data model, the *Active Record* pattern [22] could be applied.

An alternative to top-down architectures (like the three-tier architecture) is the onion architecture [i] where infrastructural services (like storage) are attached on the outside of the architecture (instead of 'built on top of'). The *Repository* pattern [22] could be applied to maintain the relation between information and data model in this architecture.

Storage concerns affect program architecture.

### 2.2.2 Impedance mismatch

Managing the relation between storage data models and program information models affects not only architectural elegance. Commonly, *object-oriented* programs

Models may be fundamentally incompatible.

use a *relational* DBMS for storage. The fundamental differences (i.e., incompatibilities) between these models give rise to the infamous object-relational *impedance mismatch* [33]. Object-relational mappers (ORMs) may be used to bridge the difference between these models, but their unsuitability for non-trivial mappings earns ORMs the moniker “the Vietnam of Computer Science” [j].

### 2.2.3 Interoperability

Developers have no real incentive to consider interoperability.

Interoperability is the quality of heterogeneous systems to exchange information [43]. Apart from customer demand, there is no real impetus for program developers to consider interoperability; most programs are designed to simply store data for their own purpose—not to exchange it with other programs.

Interoperability requires solutions to many differences.

Interoperability is hindered by heterogeneity on various levels [39], such as syntactics (e.g., data format differences), semantics (e.g., interpretational differences), pragmatics (e.g., communicational differences) and the social world (e.g., cultural and legal differences). For one program to exchange data with another, both programs need to support interoperability on all these levels. As programs are currently responsible for storage and thereby for the actual exchange of data between programs, they determine the opportunities for interoperability as well.

To support data interoperability between programs, various functionalities are required [43, 41], such as:

- a mechanism to communicate and exchange information;
- a mechanism to browse available information;
- a mechanism to adapt information (from different schemata or structures);
- a mechanism to integrate information (from different sources).

## 2.3 CLOUD-RELATED CONCERNS

The proposed knowledge cloud assumes responsibilities for storage, for *many* programs and *many* storage mechanisms. Consequently, the knowledge cloud is concerned with integrating heterogeneous data sources and models, providing program-specific information models, aggregating, transforming and exchanging data. *Knowledge* about the models, their structure and relevance is critical. The following sections elaborate on related concerns.

### 2.3.1 Knowledge models

Knowledge is information with context on relevance.

We define *knowledge* as information with *context* to determine its relevance (see later section §3.1.1). Literature defines a *knowledge level model* [60] as “a model constructed in a manner whereby no specific attention is paid to implementation issues and decisions”. Knowledge level models, such as ontologies and problem solving models, are useful for system engineering and human understanding. They

are also relevant to establishing canonical models (section §2.3.4) and transformations between models (section §2.3.3).

**ontologies:** an *ontology* is “an explicit specification of a conceptualization” [26]. Such models generally describe the *shared* understanding of some domain by multiple agents. Ontologies improve interoperability, re-usability and sharing of information. The models differ in the level of formality (highly informal up to rigorously formal), purpose or subject matter (specialized versus general world).

**problem solving models:** a *problem solving model* is “a description of problem solving at the knowledge level” and “specifies which bodies of knowledge participate in problem solving and how they relate to each other” [60]. Such models express the structure of problem solving while abstracting from implementation details. Problem solving models improve re-usability and efficiency. The models differ on scope, abstraction level, formalism, problem categorisation and problem solving methods.

### 2.3.2 Model interchange

Uschold [60] defines *knowledge interchange* as “the exchange of information content of two or more independently defined *knowledge bases*” to facilitate *knowledge sharing* and *reuse*. Sharing knowledge between *agents* requires a communication protocol, which may be a *translation* between the internal formats of individual agents or translations with a common *interchange format*.

In our terminology: *model interchange* (cf. knowledge interchange) is the “exchange of *content* of multiple independently defined models” to facilitate interoperability (cf. knowledge sharing) and reuse. Interoperability between programs and storage applications (cf. agents) requires transformations between their internal models (cf. formats) or transformations with a common interchange or canonical model (cf. interchange format).

Achieving full interoperability without interchange model requires transformations between each pair of internal models. Using an interchange model reduces the complexity of translating between models [60], as depicted in figure 4. One obstacle is the *interaction problem*, which states that the tasks of an application affect the nature and content of the model (“knowledge base”), harming its genericness.

Model interchange requires transformations.

A common interchange model reduces the number of needed transformations.



**Figure 4:** The required number of transformations (depicted as arrows) between  $n$  different models, without (left;  $\frac{1}{2}n(n-1)$ ) and with (right;  $2n$ ) an interchange model.

### 2.3.3 Model transformations

Bidirectional model transformations are hard to implement.

Developing sufficiently powerful transformations between models is a difficult problem [60]. Transformations should be deterministic and often need to be bi-directional [53]. A bidirectional transformation with consistency relation  $R$  would consist of two directional functions (parametrised by source and target models):

$$\begin{aligned}\vec{R} &: A \times B \rightarrow B \\ \overleftarrow{R} &: A \times B \rightarrow A\end{aligned}$$

In practice, bidirectional transformations are *not bijective*; this requirement would be too restrictive [53]. However, maintaining consistency (deterministically) with non-bijective bidirectional transformations is more difficult. Theoretically, transformations allow *sequential composition* (e.g.,  $(A \rightarrow B) \circ (B \rightarrow C) = A \rightarrow C$ ), but this is problematic in practice. Transformations are *coherent* if they are *correct*, *hippocratic* (i.e., they do not modify models that are already consistent; also known as ‘check-then-enforce’ semantics) and *undoable* (i.e., restoring one model to a previous state restores the other model to the previous state). In practice, undoability is too restrictive.

The trade-offs between qualities and restrictions are design considerations for transformation approaches [52]. Other design considerations are the following. Are transformations explicitly bidirectional? Should the user resolve inconsistencies or can inconsistencies be tolerated? Must source models be updated as the target model is modified?

There are various approaches for implementing transformations.

Model transformation is an active research area in computer science. Much research on bidirectional research is based on (triple) graph grammars. Triple graph grammars [49] define graphs for the source model, target model and a graph with correspondences between source and target nodes. Initiatives like OMG’s *Model Driven Development* aim to reify model transformations into first-class software engineering constructs and drive the development of model transformation approaches and tools, such as QVT [37] or ATL [35].

Another foundation of transformations is the *lens* [8, 32]: a bidirectional transformation between pairs of connected structures. Originally, lenses were *asymmetric*: one model was the primary model, the other was a ‘view’. Since then, the mathematical foundation of lenses has been strengthened [32] to permit sequentially-composable *symmetric* lenses. Lenses have been used to synchronize hierarchically structured data [21], to implement a generic tree synchronization framework (“Harmony”) [20] and to enable updatable views for relational databases [7].

### 2.3.4 Federated databases and canonical models

Federated and composite databases facilitate data interoperability.

A federated database “interconnect[s] databases [with minimal] central authority yet support[ing] partial sharing and coordination among database systems” [30]. A *composite* database system has a central schema, contrary to a *federated* database system, although both are named ‘federated’ in practice.

The databases in a federation may be heterogeneous; each database defines an export and import schema. Conflicts between their models (e.g., with regard to naming, value domains, relation cardinality) must be resolved, for example by defining schema mappings (i.e., database *views*). The autonomy of each component database and the heterogeneity between them makes consistency harder to guarantee, as transactions in a federation may span multiple databases, possibly with different consistency guarantees (section §2.1.3).

The federation as a whole may have a *canonical model* (a unified data model) that serves as interchange model (section §2.3.2) between the different component database schemata. The canonical model may be derived from component schemata as a virtual composite view (the ‘Global as View’ approach). Alternatively, the ‘Local as View’ approach calls for component database schemata to implement a fragment of the canonical model.

Unified data models require a structure that can express the structure of sub models. The Entity–Relationship–model is often used for ‘unified data models’, but is actually inadequate for that purpose [48]. The functional model and some object-oriented models are most suitable [51, 48]. Some common model structures will be discussed in section §2.4.

A canonical model unifies sub-models.

Unified models must compose data models with different structures.

## 2.4 MODEL STRUCTURES

Models describe the structure of (data, information, knowledge) content. Various models exist and the knowledge cloud should allow the expression of all structures and support transforming between them. An overview of the most common models and their qualities follows.

### 2.4.1 Hierarchical

The hierarchical information model was common in early databases and still is in file systems (with directories/folders as interior nodes and segments/files as leaf nodes). Hierarchies are often used to model registries, such as the Windows registry and LDAP-services (e.g., OpenLDAP).

Hierarchies structure content in a tree of *interior nodes* (that reference other nodes) and *leaf nodes* (that reference no other nodes). Each node has a single parent node, except the ‘root’ node which has none.

The major advantage of the hierarchical model is its *simplicity* [57]. However, there are significant disadvantages: it is difficult to represent many-to-many relationships; deletion of a node can delete its descendants; ‘conceptually symmetric’ queries<sup>1</sup> cannot always be answered equally easily; it is difficult to answer *ad*

Hierarchical: data in trees of nodes.

👍 Simplicity

🗨️ Navigational, unsuitable for *ad hoc* queries

<sup>1</sup> A model structure’s semantic relativism is the power of its operations to derive views (i.e., external schemata) with another conceptualization on the same content [48]. Conceptually symmetric queries yield semantically related views. For example: “find all *Xs* that contain *Y*” and “find all *Ys* that are contained by *X*” are conceptually symmetric because they query the same relation between *X* and *Y* (viz. *contains* (*Xs*, *Ys*)) but yield different results.



*hoc* queries. Core to these disadvantages is the model's navigational nature: users must traverse a specific 'path' to access data (cf. *access path dependence* [12]).

#### 2.4.2 Network

Network: data in full graph.

👍 Flexibility  
👉 Navigational, unsuitable for *ad hoc* queries

The network model was standardized by Committee on Data Systems Languages (CODASYL). It generalizes the hierarchical tree structure to full graphs and thus supports *multiple parentage*: any data item could have multiple parents [44].

Compared to hierarchies, networks permit cycles, more realistic models and more flexibility. However, the model is still navigational and thus unsuitable for *ad hoc* queries.

#### 2.4.3 Object-Oriented

Object-oriented: data as objects in a class type hierarchy.

👍 Encapsulation, inheritance, polymorphism, suitable for complex values

👉 Navigational, badly scalable

The object-oriented model was formally introduced by SIMULA 67; the SMALLTALK programming language subsequently introduced 'Object-Oriented Programming' as a paradigm. Although somewhat different than the initial design, the object-oriented model is the industry standard for programming languages.

The object model represents data as *objects*, structured as *classes* with operations (i.e., methods) and attributes (i.e., variables) [15]. Classes are arranged in a hierarchy of generalized super-types and specialized subtypes. Objects are identified as 'the same' by being the same instance, not by having equal attribute values (as would be the case with a relational model).

The major advantages of the object-oriented model are encapsulation, inheritance and polymorphism. Encapsulation protects a class's internal state from direct outside access, thereby improving robustness. Inheritance enables classes to extend and specialize others, thereby reusing existing definitions. Polymorphism permits synonymous operations to have different implementations for different object types while allowing clients to remain largely ignorant of these differences. The object model is suitable for modelling complexity (nested structures) but not for *ad hoc* queries [54].

Disadvantages of applying this model in a DBMS include poor performance (query optimization is complex) and lack of scalability. Like the network model, the object-oriented model is navigational and suffers the corresponding weaknesses [44].

#### 2.4.4 Relational

Relational: data as tables of rows and columns.

The relational model was introduced by Codd [12] to abstract from the 'internal representation' of data in databases and solve problems that haunted the (then dominating) hierarchical and network models. The relational model was revolutionary and is still the industry standard today. Popular DBMSs (MySQL, PostgreSQL, SQLITE) have a common language (i.e., SQL) to query for data in the managed relational databases.

The relational model is a mathematically sound model (i.e., its operations are

defined in a relational algebra). It represents content as *tuples* (i.e., rows), structured in a *relation* (i.e., table) on sets (i.e., columns). Tuples can cross-reference others by using (foreign) keys.

The major advantage of the relation model is *data independence*: the independence from changes in data types and data representations. Particularly, it reduces dependence on data ordering, indexing and access paths (i.e., it is not navigational). The relational model has powerful operations to derive different conceptualizations (i.e., views) from one source model and thus permits relations to be (logically) *symmetrically exploited* (cf. conceptually symmetric queries; 2.4.1). The relational model is suitable for *ad hoc* queries but not for modelling complexity [54] (e.g., nested structures).

Despite its improvements over older information models, the relational model has fundamental limitations [59]: it lacks object identities; one cannot distinguish relationships from entities (i.e., entities are modelled as relationships between an entity identifier (primary key) and its attribute values); it does not support complex values, collection types, inheritance or methods.

👍 Data independence, *ad hoc* querying

👎 No complex values or type hierarchies

### 2.4.5 Object-Relational

The object-relational model extends the relational model with support for complex values (like the object-oriented model does), abstract data types, inheritance and rule systems [44, 15]. Users can define and extend new data types and operations. Objects may be modelled as columns, rows or as nested values in relational tables. The object-relational model has a less severe impedance mismatch (section §2.2.2) than a pure relational model [55]. Object-relational DBMSs may use the query language SQL3 that extends SQL(2) with object-oriented features.

The object-relational model is suitable for *complex* data *with* queries. To contrast: the relational model is suitable for *simple* data with queries; the object model for complex data *without* queries [54].

Object-relational DBMSs scale well. However, the performance of ORDBMSs may be worse than pure object-oriented DBMSs if the client merely needs an object store (i.e., overkill). Additionally, query optimization and index structures may perform worse than with a purely relational DBMS [56].

Object-relational: data in object tables.

👍 Object-oriented features, *ad hoc* queries, scalable

👎 Performance

### 2.4.6 Entity-Relationship

The Entity-Relationship model [11] defines two distinct constructs to model information: *entities* and *relationships* to inter-associate entities.

The ER-model is supposed to be sufficiently expressive such that it shares most advantages of the network model and relational model, without the limitations.

The prime weakness of the ER-model is its very essence, namely: having two distinct basic structures [48]. This leads to arbitrary modelling; some designs may define some concept as an entity, others as a relationship.

Entity-relationship: data as entities or relationships.

👍 More expressive than network and relational models.

👎 Two first-class constructs

### 2.4.7 Functional

Functional: directed graph of sets (nodes) and total functions (arcs).

The functional model is a directed graph with *nodes* being sets (of values or entities) and *arcs* being functions (one-to-one, partial, total, onto). Functional models may be queried with languages like DAPLEX [50], FQL [9] or AmosQL [46].

Many model structures use *records* [36] to describe relations between entities: fixed sequences of values, like relations with tuples (relational structure) or nodes with children (hierarchical structure). Record-based structures are suitable for modelling information with the same kind of attributes. The more specialized or exception some information is, the less appropriate a record-based structure is<sup>2</sup>.

A functional structure generalizes from records and thus expresses exceptional relations and derived functions as naturally as stereotypical relations and primitive functions [50]. Derived functions (e.g., calculated functions, functions that adapt information from another model) are critical for modelling other conceptualizations of some source model [28]. Consequently, the functional model is sufficiently powerful to express other model structures [51] (e.g., relational or network), possibly enforced by ‘data policy definitions’.

👍 Can express all other models

## 2.5 RELATED WORK

The concept ‘knowledge cloud’ is related to earlier research. Sibley and Kerschberg [51] defined the *Generalized Data System* as a system that could accommodate any model structure (section §2.4). *Mediator Systems* [42] fuse information from heterogeneous information sources. We elaborate on a few technologies that inspire our work: to model knowledge and facilitate interchange, to structure storage, to centralize knowledge, and to compose knowledge from different sources.

### 2.5.1 Semantic web

The World Wide Web Consortium (W3C) started the Semantic Web project to “[provide] a common framework that allows data to be shared and reused across application, enterprise, and community boundaries” [k].

The integration of heterogeneous data sources into one “Web of Data” is of prime importance to the Semantic Web [19]. The technology stack of Semantic Web (which includes Linked Data and RDF) has already proved useful for the integration of medical and social information.

The Semantic Web aims to integrate data on the Web.

#### 2.5.1.1 Linked Data

Linked Data [5] is a standardized approach to structuring and referencing data on the WWW conform Semantic Web principles. The most fundamental rules are:

<sup>2</sup> Consider a database of car models and their attributes (number of doors, length, fuel consumption); describing a nuclear-powered flying car in (solely and exactly) those terms would be artificial at best.



- use Uniform Resource Identifiers (URIs) with the **http**-scheme to reference data;
- use standards (RDF, SPARQL);
- and refer to related data by their URIs.

### 2.5.1.2 Resource Description Framework (RDF)

The Resource Description Framework (RDF) standardizes [1] how relationships between ‘pieces of information’ can be described as triples of subject-predicate-object (i.e., a graph). Specifically, it permits different authors to use different names and descriptions of conceptually similar information. SPARQL is a language specification [m] for querying and manipulating RDF graphs.

RDF is not guaranteed to allow for all common data model structures to be represented, but it is commonly used for data interchange (in general) and with object models [n] and relational models [o]. Using RDF does not *solve* the common object-relational impedance mismatch (section §2.2.2) but replaces it by an object-RDF mismatch [16, 40]. Mappers, libraries and APIs to access RDF-models from (object-oriented) programming languages are available but they may not fully support the flexibility of RDF models [38].

### 2.5.2 Microsoft Windows Future Storage (WinFS)

Microsoft has attempted [p] to implement ‘structured storage’ since the 1990s. Its projects intended to abstract from physical storage and to provide more structure to (meta)data. It would allow for easier data exchange between applications and more complex queries. To that end, the projects attempted to supplement or replace the traditional byte-based hierarchical file system by database technologies and models. WinFS [q] is perhaps the most well-known project, once promoted as one of three fundamental new technologies in Windows ‘Longhorn’ (eventually released as Vista). The project ended with cancellation.

Attempts to implement structured file storage are old and mostly unsuccessful.

### 2.5.3 Horde

Horde [r] is a PHP groupware system. Groupware is the class of applications that facilitate Personal Information Management for groups of people: e-mails, calendars and appointments, tasks, address books, notes, file sharing etc. Groupware applications are often tightly-integrated and support various storage mechanisms and various interfaces for clients to access their information.

Like the knowledge cloud, groupware consolidates storage needs for clients and facilitates information exchange between users. Like a program, this groupware product is responsible for storage and (as such) not storage-agnostic.

It may facilitate information exchange with (different) client devices through domain-specific protocols (e.g., CalDAV for calendar and task data, CardDAV for address books, IMAP for e-mail). Such standardized protocols have clearly defined operations and are not context-adaptive.

Complex programs must support various protocols and storage mechanisms.

#### 2.5.4 OpenLink Virtuoso Universal Server

Data servers  
implement various  
storage mechanisms.

OpenLink Virtuoso Universal Server [s] allows clients to access and manipulate data from various sources and formats via a single endpoint (viz. the Virtuoso server) through one of many protocols/APIs. It also permits existing applications to extend the server.

Information should be  
tailored to the client  
program's model.

Virtuoso is a data server that implements various storage mechanisms; to establish a united data model and facilitate data interchange. It also allows the server to be extended by application logic. In these regards, Virtuoso is very similar to what the *back-end* of the knowledge cloud is supposed to be. Virtuoso does not seem to support tailoring and transforming information models to the needs of client programs as intended for the knowledge cloud..

#### 2.5.5 Amos II

Mediator systems  
combine pieces of  
information from  
multiple sources.

Amos II [18, 46] is described as a *distributed mediator system*. At its core, it is a *functional* and *object-relational* database management system with common database features (e.g., storage, recovery, transaction, the AmosQL query language). It implements a *functional* query language derived from DAPLEX [50], facilitates data integration between heterogeneous sources and optimizes query. Amos II can operate in a distributed multi-database configuration (e.g., as a federated database system; section §2.3.4).

We value many of the concepts Amos II applies; it may relatively easily fit the role of knowledge cloud in our proposed paradigm. It implements building blocks (mediators) to encode domain-specific knowledge about data and reconcile schematic incompatibilities between data sources, which is extremely useful to support context-adaptivity and storage-agnosticism.

# 3 | SOLUTION

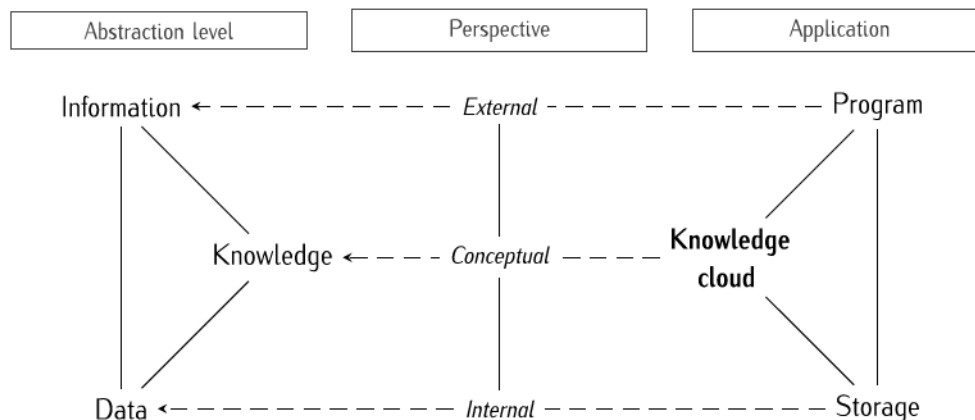
We have established the background ([chapter 2](#)) of knowledge cloud by describing various concerns the knowledge cloud would need to handle. In this chapter, we elaborate on the:

- §1. [Reason for the knowledge cloud](#): why is the knowledge cloud the right instrument to deal with the raised concerns?
- §2. [Qualities of the knowledge cloud](#): how does the knowledge cloud solve the problems of our current data management paradigm?
- §3. [Functional requirements](#): what needs must a knowledge cloud satisfy?
- §4. [Consequences of the new paradigm](#): what is the role of users, developers, data and programs in the new paradigm?

## 3.1 REASON FOR THE KNOWLEDGE CLOUD

We considered models at different abstraction levels, from different perspectives and for different applications, as shown by figure 5. We assigned a perspective and abstraction level to the knowledge cloud that would otherwise be left unassigned. For that reason, we reckon the knowledge cloud is not an optional practical tool but actually a critical service that satisfies a conceptual need which the current data management paradigm does not recognize.

The knowledge cloud fills a conceptual gap.



**Figure 5:** Mappings (dashed edges) and interactions (solid edges) between model abstraction levels, perspectives and software applications.

To reinforce this vision, we elaborate on each of the concepts in the figure.

### 3.1.1 Model abstractions: data, information, knowledge

One can model entities and attributes at different levels of abstraction. The abstraction levels *data*, *information* and *knowledge* can be distinguished [1]. Higher levels imply more abstraction, more meaning or more value [47]. We may define the following abstraction levels (inspired by [2, 6, 60]):

Data is raw.	<p><b>data</b> are <i>raw</i> facts, like recorded descriptions of things, events, activities and transactions.</p> <p><i>Example:</i></p> <pre>&lt;person&gt;&lt;id&gt;wdrijfhout&lt;/id&gt;&lt;ft&gt;Emperor&lt;/ft&gt;&lt;/person&gt;</pre>
Information is data with meaning.	<p><b>information</b> is data with some <i>meaning</i> to the perceiver (a person, a program).</p> <p><i>Example:</i></p> <p>The person identified as <b>wdrijfhout</b> has job title <i>Emperor</i>.</p>
Knowledge is information with context.	<p><b>knowledge</b> is information with <i>context</i> to determine relevance; it is “anything that can be known or believed about a real or hypothetical world” (including “matters of fact” or “ways of reasoning”).</p> <p><i>Example:</i></p> <p>The person identified as <b>wdrijfhout</b> has job title <i>Emperor</i>, in the context of <b>wdrijfhout</b>’s fantasies in the year 2015.</p>

### 3.1.2 Model perspective: internal, external, conceptual

Early database management systems tended to enforce a tight coupling between how data is *accessed* by clients (i.e., the external schema) and how data is *stored* (i.e., the internal schema) and thus burdened the clients with irrelevant technical details. ANSI-SPARC recognized this lack of *data independence* [58] and proposed the three-schema<sup>1</sup> approach which includes the conceptual schema. The three-schema approach consists of the following models:

**internal models** represent the ‘physical’ *data* as seen by the information system. These models describe the storage of the conceptual model, such as performance optimization strategies, encoding mechanisms, access paths and the data model.

*Example:*

Entity PERSON is persisted as a database table; attribute NAME has a variable length (reserve space for 20 characters); attribute JOB is a foreign key to the table JOB; attribute NAME is indexed for fast look-up.

Entity JOB is persisted as a database table; attribute TITLE with variable length (reserve space for 30 characters); attribute SECURITY CLEARANCE LEVEL is an integer between 0 and 10.

<sup>1</sup> Over time, the word “model” superseded “schema”.

**external models** represent the 'logical' *information* as seen by a user application. These models are views of the conceptual model, tailored to distinct users/programs and their needs.

*Example:*

View USER has a NAME and an ISADMIN flag.

Only PERSONS with a JOB with SECURITY CLEARANCE LEVEL > 1 are USERS.

A SECURITY CLEARANCE LEVEL > 7 sets the ISADMIN flag.

**the conceptual model** represents *knowledge* [60] about some domain as relevant for the user 'community' as a whole. Its perspective is a relatively *stable* intermediate model to which other models (i.e., internal and external models) are mapped.

*Example:*

Entity PERSON has a NAME and optionally a JOB.

Entity JOB has a TITLE and a SECURITY CLEARANCE LEVEL.

### 3.1.3 Model applications: storage, program, cloud

The three-schema approach is typically applied to the architecture of a single information systems (e.g., a DBMS). We can apply the distinction of three schemata to the data management paradigm as a whole:

**storage applications** offer *data* persistence and retrieval services. They tend to be general-purpose and allow the user to manage arbitrary data. They generally mandate the use of a particular model structure (e.g., *relational* DBMSs, *hierarchical* file systems) and its constructs (e.g., relations and tuples, files and folders). Storage applications correspond to the internal models of the three-schema approach.

*Examples:*

DBMSs, file systems, networked file servers.

**program applications** are domain-specific *information* processing tools. Their domain-specific information model is embedded and instantiated by the program. The information model generally serves to support the user application's functions and the specific implementations. Program applications correspond to the external models of the three-schema approach.

*Examples:*

Mail clients, word processors and business administration tools.

**the knowledge cloud** maintains the *knowledge* on providing programs with relevant *information* from the *data* in storage. What facts (data) and meanings (information) are relevant, depends on the context (knowledge). Context may be shaped by queries, the involved applications, the user's identity and credentials, external configuration, performance requirements etc. The knowledge cloud corresponds to the conceptual model of the three-schema approach.

The weaknesses of data management paradigms without a knowledge cloud resemble the weaknesses of a ‘two-schema’ approach. Without a knowledge cloud to provide contextually-relevant knowledge, programs have to determine what is relevant and retrieve the data from storage themselves. Unsurprisingly, the program developer would not be particularly inclined or able to consider interoperability, lacking knowledge about the relevance of their information for other programs.

## 3.2 QUALITIES OF THE KNOWLEDGE CLOUD

To handle the various concerns a knowledge cloud has to deal with, it must be flexible and extensible. Therefore, the knowledge cloud enables two qualities of the new data management paradigm: storage-agnosticism (section §3.2.1) and context-adaptivity (section §3.2.2). These qualities are fundamental to our approach of solving the problems of the current data management paradigm (section §3.2.3).

### 3.2.1 Storage-agnosticism

The ANSI-SPARC three-schema approach [58] (cf. section §3.1.2) promotes *data independence* for users of database management systems. Data independence “insulates a user from adverse effects of the evolution of the database environment”; i.e., differences in physical storage of data do not affect the client. By virtue of data independence, applications can remain ignorant of the physical representation of information as data.

Storage-agnosticism permits programs to exchange information in their own model.

We propose *storage-agnosticism* to entail *data and information independence*. Storage-agnostic programs do not consider data models or abstract from them (cf. section §2.2.1). Storage-agnostic programs only use their information model and rely on the knowledge cloud for providing and storing those models using some data storage mechanism. This separation of concerns allows the knowledge cloud considerable freedom in *how* it manages storage and populates information models. In particular, it allows the knowledge cloud to consider interoperability (cf. section §2.2.3).

#### RELATION TO GUIDING EXAMPLE (section §1.6)

Storage-agnosticism allows Firefox and Internet Explorer to remain oblivious of the storage or origin of their bookmarks and favorites. This allows the knowledge cloud to aggregate bookmarks from Firefox and favorites from Internet Explorer, transform them and serve bookmarks as favorites to Internet Explorer and serve favorites as bookmarks to Firefox.

### 3.2.2 Context-adaptivity

The knowledge cloud needs to select and adapt data models from its storage services into *contextually relevant* information models (i.e., *knowledge* models; section §3.1.1). Context determines what is relevant and how the knowledge cloud should *adapt* the models accordingly; this is *context-adaptivity*. Context-adaptivity



is inspired by Context-Oriented Programming [31] and its actor-, environment- and system-dependent behaviour variations.

When a client program requests the knowledge cloud to provide information, the *context* could include the identity of the program. Additionally, it may include user identification and *authorization* credentials in order to include sensitive information and allow certain operations. Context may specify *performance* trade-offs to affect data compression and how many results are returned. Context may specify the use of *versioning* to make model mutations incremental (rather than destructive).

Crucially, client programs may specify *some* context but the knowledge cloud ultimately determines the whole context. The knowledge cloud may also consider user configuration, compatibility measures, security constraints and other contexts.

Context-adaptivity allows programs to use various contextually-enabled features in the knowledge cloud without necessarily being aware of it. This is useful for managing interoperability (cf. section §2.2.3) and cross-cutting concerns. In other words: client programs could be oblivious to authorization and versioning and still enjoy the benefits of security and automatic back-ups.

Context-adaptivity permits users to control information exchange.

#### RELATION TO GUIDING EXAMPLE (section §1.6)

Firefox may request knowledge in the domain 'bookmarks'. Context-adaptivity allows the knowledge cloud to consider the currently logged in user, the computer network and user preferences and *actually* yield knowledge in the domain 'not-deleted work-related bookmarks accessible to John, including bookmarks transformed from favorites'.

### 3.2.3 Benefits to data management

Using a knowledge cloud to enable storage-agnosticism and context-adaptivity should contribute to solving the four identified problems (see research questions in section §1.3) of our current 'primitive' data management paradigm, as shown in table 1.

**Table 1:** This table shows how storage-agnosticism and context-adaptivity affect the identified data management paradigm problems.

PROBLEM	STORAGE-AGNOSTICISM	CONTEXT-ADAPTIVITY
[Q1] <i>Programs are unnecessarily complex.</i>	[I] Absolves from storage-related concerns.	[II] Absolves from certain program-related concerns.
[Q2] <i>Data is badly interoperable.</i>	[III] Unifies models from heterogeneous data sources.	[IV] Transforms and adapts data to contextual requirements.
[Q3] <i>Users are not in control of their data.</i>	[V] Enables users to dictate data storage policies.	[VI] Enables users to dictate data usage and processing policies.
[Q4] <i>Developers must write inessential code.</i>	[VII] Obsoletes data models, storage APIs and exchange protocols.	[VIII] Centralizes reusable logic.

### 3.3 FUNCTIONAL REQUIREMENTS

The benefits we attributed to storage-agnosticism and context-adaptivity (table 1) imply abstract requirements of the knowledge cloud. We also considered various concerns in our background research (chapter 2) that the knowledge cloud must deal with. The following list of detailed requirements is derived from those background concerns and grouped by knowledge cloud benefits.

- I. In order to reduce program complexity, the knowledge cloud absolves from storage-related concerns. Therefore, it

—*with regard to data formats* (section §2.1.1)—

1. **Must** serialize and deserialize (application-specific and general-purpose) arbitrary data formats.
2. **Must** interface with file systems.
3. **Must** interface with database management systems.

—*with regard to storage interfaces* (section §2.1.2)—

4. **Must** interface with storage mechanisms to store information models.
5. **Must** derive storage mechanism queries (e.g., SQL) from context (e.g., a knowledge domain).
6. **Must** provide a knowledge cloud interfacing library.
7. **Should** implement knowledge cloud-adapters for existing libraries and language constructs.

—*with regard to consistency guarantees* (section §2.1.3)—

8. **Should** support transactional storage processing.
9. **Could** allow clients to specify desired consistency guarantees.
10. **Could** support multiple consistency models (ACID, BASE).
11. **Could** support distributed transactions.

- II. In order to reduce program complexity, the knowledge cloud absolves from program-related concerns. Therefore, it

—*with regard to program architecture* (section §2.2.1)—

1. **Must** be integrable with existing architectures.
2. **Should** not impose software-architectural constraints.
3. **Must** enable less complex architectures (compared to non-cloud-solutions).

—*with regard to impedance mismatch* (section §2.2.2)—

4. **Must** support customizable mappings between differently-structured models.
5. **Must** reconcile structural incompatibilities between models.

—*with regard to interoperability* (section §2.2.3)—

6. **Must** be a communication kernel between interoperating programs.



7. **Must** enable interoperability between interoperability-oblivious programs.
  8. **Must** transform between (program-specific) information models.
- III. In order to improve data interoperability, the knowledge cloud unifies heterogeneous data sources. Therefore, it
1. **Must** maintain a federation of data stores and a canonical data model (section §2.3.4).
  2. **Must** facilitate interchange between models from data stores (section §2.3.2).
- IV. In order to improve data interoperability, the knowledge cloud transforms and adapts data to requirements. Therefore, it
1. **Must** allow all common data model structures (section §2.4) to be represented.
  2. **Must** support transformations to facilitate interchange between models (section §2.3.3).
  3. **Must** maintain knowledge on the contextual relevance of information (section §2.3.1).
- V. In order to improve user control over data, the knowledge cloud enables users to dictate data storage policies. Therefore, it
1. **Must** allow users to install data sources.
  2. **Must** allow users to configure the relevance of particular data sources in specified contexts.
- VI. In order to improve user control over data, the knowledge cloud enables users to dictate data usage and processing policies. Therefore, it
1. **Must** allow users to install (model processing) agents (plug-ins, software components).
  2. **Must** allow users to configure the relevance of particular agents in specified contexts.
- VII. In order to reduce the amount of inessential code developers write, the knowledge cloud obsoletes data models, storage APIs and exchange protocols. Therefore, it
1. **Must** maintain the relation between (storage) data models and (program) information models.
  2. **Must** control data models, storage APIs and exchange protocols indirectly, as implied by information models.
  3. **Could** offer an interface to access raw data models, storage APIs and exchange protocols.

VIII. In order to reduce the amount of inessential code developers write, the knowledge cloud centralizes reusable logic. Therefore, it

1. **Must** allow programs to specify that certain centralized services are contextually relevant.
2. **Should** allow programs to suggest the installation of agents.

### 3.4 CONSEQUENCES OF THE NEW PARADIGM

A knowledge cloud with the aforementioned qualities and requirements drives the new data management paradigm we propose. The new paradigm would affect programs, data, developers and users as follows:

#### 3.4.1 Programs

[Q1]

Programs would ignore storage concerns but must support remote model changes.

Programs would define information models and provide a user interface to access that information. Programs would no longer explicitly consider storage but *must* now consider unexpected modifications to their information models (propagated by the knowledge cloud).

Programs would request the knowledge cloud to provide and manage instances of a particular information model. The definition of information models must be shared with the knowledge cloud. Programs might publish type declarations in Java code or a custom model description language. Future research must determine practical approaches.

To exchange information models, programs and knowledge cloud implementations may benefit from advanced programming language features, such as incremental compilation, reflection, remote procedure calls, inter-process communication, CORBA or shared memory. Future research must determine practical approaches.

#### 3.4.2 Data

[Q2]

Data would be managed by the cloud and be more interoperable.

Data would be decoupled from programs. Data would be read, written and interpreted by knowledge cloud components (knowledges, offers, feeds, agents) instead. The knowledge cloud also facilitates data interoperability by (on-demand) transformations.

The knowledge cloud is extensible and can include components to interpret various data formats from various sources with various technologies: knowledge from relational databases (possibly using object-relational mappers and SQL), knowledge from web services (possibly using JSON), knowledge from binary files (possibly using parsers).

### 3.4.3 Users

Users would need to manage a knowledge cloud by configuring data sources, installing components and adapting their availability to particular contexts. In particular, the user could install and configure components for his many on-line accounts (e-mail, instant messaging, address books, schedules). Programs that use the knowledge cloud could instantly use those accounts (indirectly) and would need no further configuration by the user.

[Q3]

Users would control their data by configuring the knowledge cloud.

### 3.4.4 Developers

Developers need to relinquish control over various concerns they previously implemented directly (like storage I/O, authorization etc.). If they cannot relinquish control, they would need to implement custom knowledge cloud components. Development approaches that separate the information model from other application logic (like Domain-Driven Design [17]) must be applied as well, because the information model must be shared with the knowledge cloud.

[Q4]

Developers should relinquish storage control and implement information models with care.



# 4 | DESIGN

We explained (chapter 3) how the knowledge cloud is critical for our proposed data management paradigm. In this chapter, we describe:

- §1. **Abstractions**: what indirections should a knowledge cloud support?
- §2. **Architecture**: how does a knowledge cloud work?
- §3. **Concepts**: what are the concepts introduced to the knowledge cloud?

## 4.1 ABSTRACTIONS

By describing the knowledge cloud's primary goals we reveal the most fundamental abstraction requirements.

The knowledge cloud provides clients with relevant knowledge. Clients do not care *why* this knowledge was relevant (context-adaptivity) or *where* it originated (storage-agnosticism). Knowledge sources in the cloud exchange knowledge. Knowledge may depend on other knowledge (to transform, expand, filter, use etc.), thus establishing a dependence hierarchy.

We identify necessary abstractions for critical phrases (underlined) in the above description and name these abstractions for use in the design:

**knowledge cloud provides clients** Clients need an interface (CLOUD) to access knowledge in the cloud.

**relevant** We need a model to describe knowledge characteristics (TRAIT) and an abstraction to make assertions about some knowledge's characteristics (DOMAIN). Clients need an abstraction to specify (SPECIFICATION) what characteristics constitute relevance.

**knowledge** Clients need an abstraction (KNOWLEDGE) to access and mutate provided knowledge.

**context-adaptivity** We need abstractions to model the context (DOMAIN, TRAIT, SPECIFICATION) of client-cloud and server-cloud interactions (like knowledge requests) and adapt it (SCOPE, BUREAU).

**storage-agnosticism** We need an abstraction for information that is relevant in some context (KNOWLEDGE) and its providing storage mechanism (FEED).

**knowledge sources** We need an abstraction to provide knowledge (FEED, OFFER) and for creating and managing these knowledge providers (AGENT).

**exchange** We need an abstraction to request and provide knowledge (QUESTION, ANSWER) and to satisfy these requests and responses (NETWORK).

**other knowledge** We need to model knowledges' dependencies on other Knowledge (QUESTION).

**dependence hierarchy** We need to establish a model of knowledge's interdependencies (NETWORK) by repeatedly searching (ORCHESTRATOR) for knowledge that is available (FEED, OFFER) and could satisfy a known dependency (QUESTION, ANSWER).

### Relation to guiding example (section §1.6)

Web browsers are clients of the knowledge cloud. The knowledge cloud provides an interface to access knowledge in a relevant domain. Firefox uses the interface to access relevant knowledge (i.e., on bookmarks); similarly, Internet Explorer would access knowledge on favorites.

The browsers do not consider the origin of the provided knowledge (storage-agnosticism) nor the circumstances that made it relevant (context-adaptivity). The knowledge cloud may consider various knowledge sources (e.g., Firefox's bookmark database, Internet Explorer's favorites folder, a file with secret hyperlinks). To satisfy Firefox's request, it may choose to use the bookmark database, ignore the secret hyperlinks (lacking authorization) and transform the favorites into bookmarks (to facilitate data exchange between Firefox and Internet Explorer).

Transforming favorites into bookmarks implies a dependency of the new bookmark knowledge on the original favorite knowledge. Similarly, to combine multiple (bookmark) knowledges implies a dependency of the 'combined' knowledge on all its component knowledges. As knowledges build on other knowledges, they establish a dependence hierarchy.

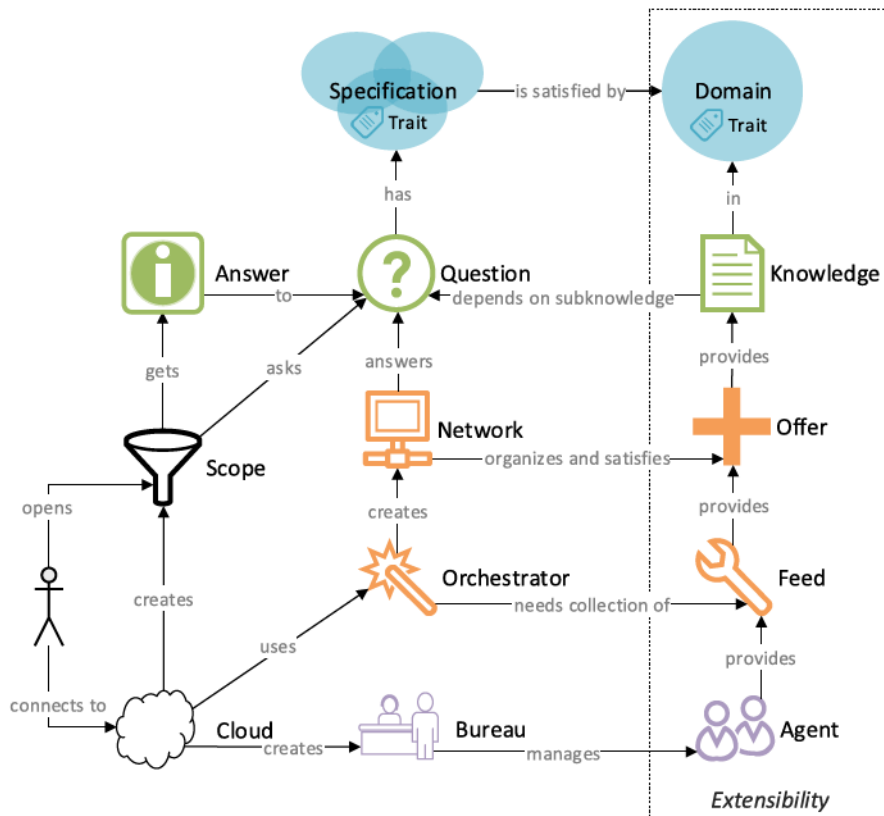
## 4.2 ARCHITECTURE

Figure 6 shows the proposed knowledge cloud architecture. We describe the concepts by walking through a common scenario in which a client asks for relevant knowledge from the cloud, from the perspectives of a client and a cloud server.

Note that the architectural design does not mandate any particular machine configuration; the design can be applied in one application, in a client-server configuration or possibly even a peer-to-peer configuration.

### 4.2.1 A client asks for knowledge in a domain

We will illustrate a typical interaction between a client application and cloud server. References to the guiding example (section §1.6) are shown in *italic*.



**Figure 6:** The proposed knowledge cloud architecture. The icon colours denote packages of related concepts. Lines denote a relation between concepts. The stick-figure represents the client. The dashed box 'Extensibility' marks the concepts that may be implemented or specialized by third-party 'plug-ins' to support storage mechanisms, web services, other data servers, transformations etc.

1. The client connects to a CLOUD (the public interface of the knowledge cloud) to initialize a session.  
*The browser uses some reusable library to establish a connection with the knowledge cloud (generally: a server on localhost).*
2. The client opens a SCOPE on the cloud.  
*The knowledge cloud has determined and set the context of the browser's interaction with the knowledge cloud: the name of the browser, the name of the user etc.*
3. The client builds a SPECIFICATION of the DOMAIN it wants the cloud to provide knowledge for. A specification is a simple logical expression of the traits a domain should exhibit. TRAITS are properties such as the knowledge schema, owner, source, credentials.  
*The browser Firefox would specify that relevant domains exhibit the traits "has bookmarks" and "for user John" and "is formatted as RDF" (a graph).*
4. The client commands the scope to get the KNOWLEDGE(s) with a domain that satisfies the specification; the scope formulates the corresponding knowledge QUESTION and uses the cloud to ANSWER it.

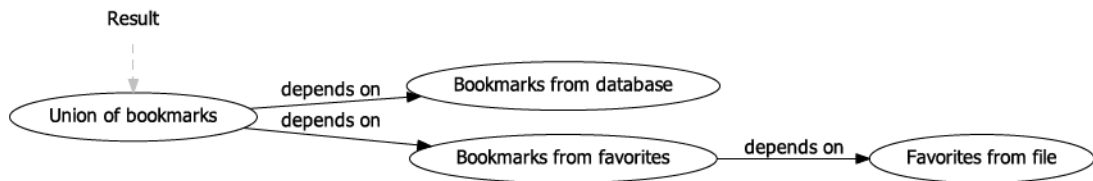


*The browser Firefox may get zero, one or more different 'bookmark collections'. Indeed, John may have multiple bookmark collections (one private, one work-related, back-ups). Some of these bookmark collections could even be stored as Internet Explorer favorites. Firefox may allow John to further restrict the specification ("no back-ups"). Additionally, Firefox may have specified to the knowledge cloud it wants all relevant knowledge "aggregated" as one virtual bookmark collection.*

#### 4.2.2 The cloud provides knowledge in a domain

1. The CLOUD maintains at least one BUREAU (a registry of AGENTS).  
*The bureau contains agents (plug-ins/modules) for storage systems (e.g., the file system and SQLite databases), for parsers (e.g., text-to-RDF), for generic model manipulations (e.g., aggregating multiple RDF-models into one) and for application-specific models (e.g., a transformer from bookmarks to favorites and vice-versa).*
2. When a client opens a SCOPE, a bureau prepares a collection of currently available feeds for that scope.  
*The agents yield feeds to recognize some domain specifications (e.g., "file bookmarks.sqlite") and to offer the corresponding relevant knowledge, if available.*
3. When the client asks the scope to answer a knowledge QUESTION, an ORCHESTRATOR for the given feeds is instantiated.  
*Firefox asks for exactly one knowledge model that "has Bookmarks" "of John".*
4. The orchestrator builds a NETWORK of knowledge OFFERS and their inter-dependencies. Offers contribute to answering some question, like the 'root' question the client asked. Offers may specify new questions if the offered knowledge depends on other knowledge. The orchestrator tries to find new offers for new questions, recursively.  
*The orchestrator asks the question to the available feeds. One feed offers this knowledge immediately from storage. Another feed (provided by the Favorite-Bookmark-Conversion-agent) offers this knowledge (transformed) and needs knowledge that "has Favorites" "of John". Another feed is able to combine both knowledges to provide the "exactly one" knowledge Firefox wanted. A network is built to describe the dependencies between all offers.*
5. Once the network is built, it builds the KNOWLEDGES (if any) that answer the root question.  
*A hierarchy of knowledge models is built from the dependence hierarchy in the network, as shown in figure 7. The knowledge cloud provides Firefox the top knowledge (the root node) and discards the network.*
6. The scope yields the knowledges to the client.  
*Firefox can now interact with the requested knowledge. Mutations propagate down/up through the knowledge hierarchy, but that is not Firefox's concern.*





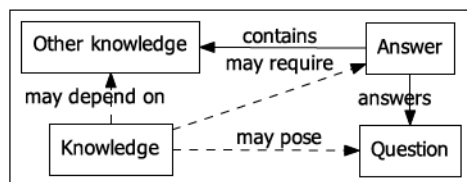
**Figure 7:** Example of the dependence hierarchy that could be constructed for a request for bookmarks.

## 4.3 CONCEPTS

For the concepts in figure 6, we will elaborate on their core responsibilities (with their collaborators) [4] and the possible implementation variations. Concepts are grouped in packages; for each package we elaborate on their relevance and design considerations.

### 4.3.1 Knowledges, questions and answers (green)

KNOWLEDGE represents an information model in some domain. Knowledge may depend on other knowledges with particular domains and amounts and express these requirements as QUESTIONS for which it demands ANSWERS. See figure 8.



**Figure 8:** Illustration of relations between KNOWLEDGE, QUESTION and ANSWER.

**RELEVANCE** Knowledge represents relevant information, abstracted from its storage mechanism. It is the primary unit of result the knowledge cloud exchanges (in questions and answers).

#### RELATION TO GUIDING EXAMPLE (section §1.6)

Knowledge models can be implemented at different abstraction levels in a trade-off with re-usability. Vendors (of applications or libraries) could implement specialized knowledge models ("InternetExplorerFavoriteFile") but would then also

need to implement the feed with (de)serialization and/or transformation logic for it. Other vendors may implement more reusable models (for example: generic RDF graph models) but they may be too 'raw' for applications to directly use. On the other hand, Mozilla Firefox actually uses the RDF data model internally [t].

#### CONSIDERATIONS

- Knowledge should be storage-agnostic (to clients) and yet transparently propagate mutations to dependent knowledge.
- The relevance of KNOWLEDGE is determined by the traits its domain exhibits (section §4.3.2).

##### 4.3.1.1 *Knowledge*

Represents a mutable information model that is relevant in some contextual DOMAIN.

#### RESPONSIBILITIES

- (Implementations) Expose some information model.
- Save/commit model mutations to dependent knowledges.
- Load/update the model with mutations from dependent knowledges.

#### IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- Implementations of KNOWLEDGE expose some information model. These implementations can be built on a generic framework (e.g., a generic graph model) or for particular client applications (e.g., domain-specific objects). To ease interoperability in the cloud, using an object-oriented or functional model is recommended.
- Implementing KNOWLEDGES with a generic data model (e.g., graph, relational [13]) generally requires implementations for plain instances (graphs, tables) and for operations thereon (set operations, projection, selection, join).
- Consider how mutations to some KNOWLEDGE are propagated to its dependent knowledges or storage mechanism.
- Practical use of the knowledge cloud may require access to KNOWLEDGE across processes/hosts. One could provide proxy implementations to transparently convert local KNOWLEDGE calls and events to remote queries and commands. To convert calls, one may use Remote Procedure Call-technology or write a custom adapter.

##### 4.3.1.2 *Question*

Expresses the need for some amount of KNOWLEDGES in some DOMAIN.

## RESPONSIBILITIES

- Describe the SPECIFICATION of the required DOMAIN.
- Describe the acceptable minimum and maximum amounts of relevant KNOWLEDGES.
- Determine if it is answered satisfactorily by some KNOWLEDGE candidates.

## IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- A simple implementation is answered by simply having the right amount of KNOWLEDGES satisfying the specification.
- More complex implementations could consider the 'cost' of using a particular KNOWLEDGE.

4.3.1.3 *Answer*

Presents KNOWLEDGE that is available to answer a particular QUESTION.

## RESPONSIBILITIES

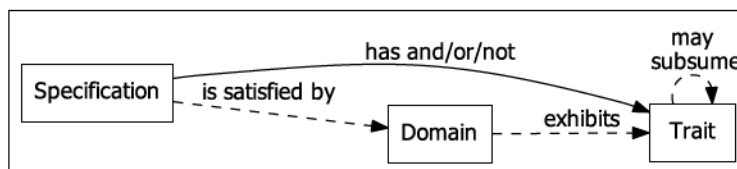
- Describe the QUESTION.
- Describe the answering KNOWLEDGE.

## IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- More complex implementations could declare a preference/priority order among ANSWERS.

## 4.3.2 Domains, traits and specifications (blue)

DOMAINS exhibit traits. TRAITS are characteristics. A domain SPECIFICATION determines what traits a domain must exhibit. See figure 9.



**Figure 9:** Illustration of relations between DOMAIN, TRAIT and SPECIFICATION.

**RELEVANCE** Domains, traits and specifications are used to characterize knowledge and determine its relevance in some context (such as a knowledge request).

#### RELATION TO GUIDING EXAMPLE (section §1.6)

A knowledge model of a simple file may have a domain with traits “file /home/john/bookmarks.dat”, “mimetype application/json”, “charset utf-8”. A book-mark’s knowledge model may have a domain exhibiting the traits “has Bookmarks”, “belongs to John”. A specification is a simple logical expression of such traits that some arbitrary domain ( $d$ ) may satisfy or not; e.g.,  $\text{hasBookmarks}(d) \wedge \text{belongsTo}(d, \text{John})$ . Clients (like Firefox) and feeds (like transformers) ask for knowledge by constructing such a specification.

#### CONSIDERATIONS

- Developers of agents, feeds and knowledges will not consider all possible traits. Therefore, supporting an open-world assumption in implementations of domains, traits and specifications may be desirable.  
One could use a three-valued logic to allow some domain to *truly* exhibit, *not* exhibit or *maybe* exhibit some trait. However, an eventual (binary) decision as to whether some knowledge is relevant or not is required. Modelling uncertainty through a ‘maybe’ state can be useful; e.g., to log the incompatibility, ask the user to decide, apply heuristics.
- Knowledges can depend on (i.e., ‘ask for’) other knowledges. The superior knowledge specifies the required domains of its dependent knowledges. One may require the reverse (i.e., inferior knowledges expressing requirements on their superiors) for modelling authorization (using information-flow control [29]) or relevance (to demote redundant or unreliable knowledge).  
A knowledge hierarchy has a corresponding instantiation order of individual knowledges. One could consider supporting temporal logic [23] to specify that some trait must hold for *some* or *for each* later knowledge (where ‘later’ means hierarchically superior).

##### 4.3.2.1 Trait

Is a characteristic (fact, property) of some subject, as exhibited by a DOMAIN or asserted by a SPECIFICATION.

#### RESPONSIBILITIES

- Determine if this TRAIT subsumes another.
- Ensure that equal TRAITS subsume each other.

#### IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- The knowledge cloud provides implementations for DOMAIN and SPECIFICATION. The knowledge cloud provides a few basic TRAITS (to describe know-

ledge schema and model type). User applications and knowledge cloud plug-ins may define traits.

- Implementations generally model a property and a value range. Generally, instances with some value range imply similar instances with a value sub-range.
  - The property could be specified explicitly (with a simple identifier attribute) or implicitly (by the class identity).
  - The value range may be undefined/singular (singleton `TRAIT`), permit absent values (`TRAITS` with actual values subsume similar ones with missing values) or be an actual value range or enumeration.

#### 4.3.2.2 *Domain*

Makes assertions about some subject's characteristics (i.e., exhibited `TRAITS`).

##### RESPONSIBILITIES

- Determines if a given `TRAIT` is exhibited.
- Provides a characterizing `SPECIFICATION`.

##### IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- A simple implementation (`SETDOMAIN`) maintains a set of affirmed traits. Traits that cannot be implied by any contained trait are determined to be not expressed (assuming a closed world).
- More complex implementations could express conjunctions, disjunctions and negations (like a `SPECIFICATION`).

#### 4.3.2.3 *Specification*

Expresses the characteristics (`TRAITS`) a candidate object (a `DOMAIN`) must exhibit to qualify as 'satisfying'. The specification pattern `[u]` is a propositional logic boolean expression to separate assertions *about* candidate objects from the candidate objects itself. The pattern is useful for selecting, validating and constructing objects.

##### RESPONSIBILITIES

- Determine if a candidate `DOMAIN` satisfies the specified conditions.

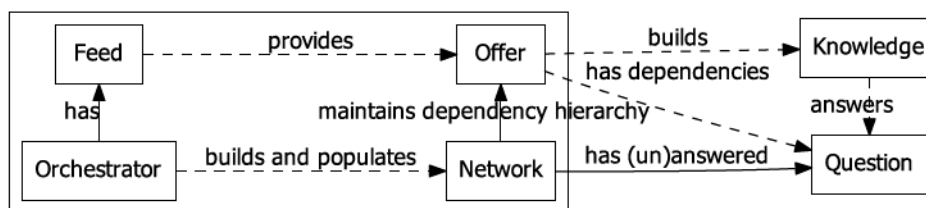
##### IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- Apart from conjunction, disjunction and negation other operators could be implemented.

- The specification pattern can be extended to support *subsumption* and partial satisfaction. In that case, instead of asking if a specification is satisfied by some domain, one could ask if the specification is *subsumed* by that domain's characterizing specification.

#### 4.3.3 Feeds, offers, networks and orchestrator (orange)

FEEDS make OFFERS for knowledges in certain domains. The ORCHESTRATOR builds NETWORKS of offers to satisfy a particular knowledge request. See figure 10.



**Figure 10:** Illustration of relations between FEED, OFFER, NETWORK and ORCHESTRATOR.

**RELEVANCE** FEEDS expose (OFFERS for) knowledge from particular knowledge sources. The ORCHESTRATOR builds a NETWORK of all relevant offers to satisfy a particular knowledge request.

#### RELATION TO GUIDING EXAMPLE (section §1.6)

These components are internal to the cloud; client applications will not directly interact with them.

If a ~~webbrowser~~web browser (say, Microsoft Internet Explorer) requires a custom knowledge model interface (InternetExplorerFavoriteFile), its developers would most likely want to provide implementations of the knowledge model and the feeds that offer it. A knowledge cloud may enable client applications to (remotely) install new feeds and knowledges to the cloud (if given authorization) or require manual installation by the user.

#### CONSIDERATIONS

- The ORCHESTRATOR should be deterministic; i.e., it should produce the same NETWORK for the same knowledge request and set of FEEDS every time.

##### 4.3.3.1 Feed

Provides OFFERS for knowledges that are relevant to (i.e., satisfy) some domain specification.

## RESPONSIBILITIES

- Provide relevant OFFERS for some domain specification.

## IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- For each implementation of knowledge, a FEED-implementation to instantiate the knowledges should be available as well.

4.3.3.2 *Offer*

Declares the availability of some knowledge.

## RESPONSIBILITIES

- Describe the knowledge structure.
- Describe the dependent knowledge QUESTIONS.
- Provide KNOWLEDGE (given answers to the dependent questions).

## IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- Each implementation of FEED generally requires an implementation of OFFER (to actually provide the offered knowledge) as well.

4.3.3.3 *Network*

Maps questions to their satisfying knowledge offers and builds the corresponding KNOWLEDGE hierarchy.

## RESPONSIBILITIES

- Maintain mapping of QUESTION to a set of satisfied OFFERS.
- Determine if some OFFER is satisfied.
- Determine if some QUESTION is answered.
- Provide KNOWLEDGES that answer some QUESTION.

## IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- A simple implementation could maintain a mapping of QUESTIONS to OFFERS for each root question.
- A more complex implementation may have a more reusable mapping, such that NETWORKS are not built anew for each request.

4.3.3.4 *Orchestrator*

Populates a NETWORK with OFFERS from a FEED that contribute to the answering of a 'root' knowledge question.

## RESPONSIBILITIES

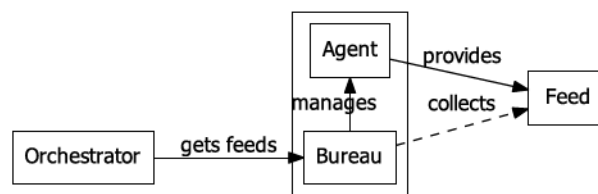
- Create a NETWORK for answering some root knowledge QUESTION.
- Prevent dependency loops among knowledges. Knowledge should never depend on itself.

## IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- To accept a knowledge OFFER, its own dependent questions need to be answered (by other offers). The space of requested offers and their dependencies can easily explode. A brute-force breadth-first search may not suffice. To solve larger spaces, one may benefit from constraint logic programming [34].

## 4.3.4 Agents and bureaus (purple)

AGENTS provide knowledge cloud components. BUREAUS manage a registry of agents and collect the agents' components. See figure 11.



**Figure 11:** Illustration of relations between AGENTS and BUREAU.

**RELEVANCE** The AGENT is an installable module or plug-in of cloud components by a trusted partner (a system administrator, an application vendor); such an abstraction allows for configuration and extension of a knowledge cloud.

The BUREAU defines an AGENT cooperation sphere; this abstraction permits selective availability of agents (e.g., to separate of mutually-incompatible AGENTS, to quarantine classified and unsafe AGENTS).

## RELATION TO GUIDING EXAMPLE (section §1.6)

There may be an agent for “Microsoft Internet Explorer” and one for “Mozilla Firefox” to provide feeds that expose client-specific knowledge (possibly as custom model implementations; e.g., InternetExplorerFavoriteFile). There may be other agents with feeds that transform knowledge models (favorites versus bookmarks); such agents could be built by anyone.



## CONSIDERATIONS

- The AGENTS may use background threads to discover knowledge sources.
- AGENTS may need to execute private initialization/disposal instructions when they are (de)registered at some BUREAU.

4.3.4.1 *Agent*

Represents a module with knowledge cloud components.

## RESPONSIBILITIES

- Provide a collection of FEEDS.

## IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- A simple implementation may provide pre-instantiated feeds.
- Implementations that adapt external systems (e.g., a file system, a DBMS, another cloud) for the cloud will generally instantiate one feed that lazily checks if a relevant knowledge source exists and can be offered.

4.3.4.2 *Bureau*

Manages a registry of AGENTS and collect their feeds.

## RESPONSIBILITIES

- Maintain a registry of AGENTS.
- Provide a collection of FEEDS.

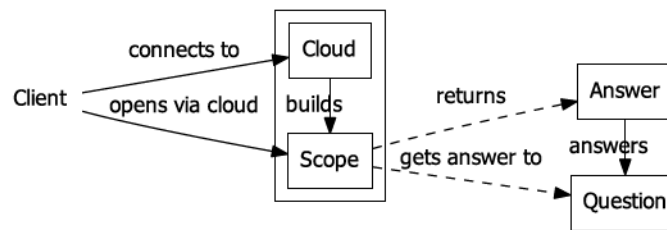
## IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- A simple implementation contains an arbitrary collection of AGENTS.
- More complex implementations may impose constraints on registered AGENTS (e.g., by vendor, class signature or some authorization mechanism).

## 4.3.5 Clouds (black)

The CLOUD is the public interface for clients to open a SCOPE for retrieving knowledge. See figure [12](#).

**RELEVANCE** To hide the inner workings of the CLOUD for clients, the knowledge cloud exposes a public interface. The SCOPE insulates the client from server-volatilities: it snapshots the CLOUD's configuration.



**Figure 12:** Illustration of relations between CLOUD and SCOPE.

#### RELATION TO GUIDING EXAMPLE (section §1.6)

The web browsers use the public interface to query the knowledge cloud for knowledge with their respective information models (bookmarks vs. favorites).

#### CONSIDERATIONS

- We should avoid forcing an architectural distinction between client and server components. Without changes to the high-level design, the CLOUD interface should be suitable for in-memory, inter-process and inter-machine implementations. While a singleton cloud per host is proposed, various concerns (e.g., performance, testability, extensibility, security, reliability) may demand that client programs run a private cloud in their own memory. Developers and programs could then still benefit from some abstractions (like knowledge retrieval and orchestration) the knowledge cloud design provides.

##### 4.3.5.1 *Cloud*

Provides a SCOPE on the cloud and its available knowledge sources.

#### RESPONSIBILITIES

- Open a SCOPE on currently available feeds.

#### IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS

- An in-memory implementation runs in the local process's memory. A proxy implementation may transparently convert local calls to calls on a remote server (or process). To convert calls, one may use Remote Procedure Call-technology or write a custom adapter using some protocol (e.g., SQL, SPARQL). Implementations may use a bureau (or multiple) to maintain configurations of interoperating agents.

##### 4.3.5.2 *Scope*

Provides the knowledge answers that satisfy some knowledge question.

**RESPONSIBILITIES**

- Provide ANSWERS that satisfy some QUESTION.

**IMPLEMENTATION VARIATIONS AND SPECIALIZATIONS**

- SCOPES may adapt the client's question to consider other contexts (e.g., user identity and authorization, application identity, manual configuration).



# 5 | VALIDATION

We identified problems our current data management paradigm suffers ([chapter 1](#)) and proposed the knowledge cloud as the solution ([chapter 3](#)). We also provided an architectural design of this knowledge cloud ([chapter 4](#)). Our efforts to validate our proposal and design consist of:

- §1. [Demonstration: a prototype](#) to show the practical application of a knowledge cloud and validate its architectural design;
- §2. [Theoretical case-study: Horde](#) to illustrate the potential gains of using a knowledge cloud on complex software;
- §3. [Results](#) of our efforts to validate the knowledge cloud's ability to solve the identified problems.

## 5.1 DEMONSTRATION: A PROTOTYPE

Our guiding example (section [§1.6](#)) describes the challenge of maintaining one hyperlink collection across various browsers with distinct domain models (specifically Firefox's *bookmarks* and Internet Explorer's *favorites*). We proposed using a knowledge cloud to maintain this relation. As implementing a practical solution is beyond the scope of this research project, we demonstrate the use of a knowledge cloud with the implementation of a knowledge cloud prototype and a few demonstration client programs.

We implement a knowledge cloud prototype and demonstration programs.

### 5.1.1 Purpose

The purpose of our demonstration is to:

- validate the architectural design of the knowledge cloud;
- illustrate how a knowledge cloud facilitates (better) data interoperability;
- illustrate how a knowledge cloud improves user control;
- assess the feasibility and practical effort of making cloud-enabled programs.

5.1.2 Scope

Our demonstration is limited in scope.

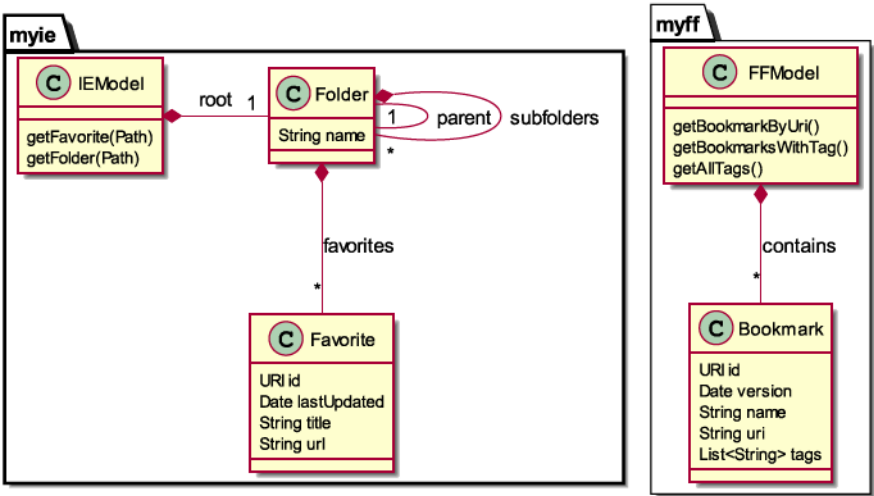
The scope our demonstration is limited to illustrate how the knowledge cloud facilitates interoperability between client programs. In particular, our demonstration does *not* consider:

- interactions between cloud and storage. Our demonstration maintains data models in-memory and does not *persist* models. Production-ready knowledge cloud *will* need to access storage.
- inter-process communication (IPC) between clients and cloud. Our demonstration runs the knowledge cloud and client programs in a single process. Production-ready knowledge clouds *will* find IPC relevant and may benefit from various existing IPC-libraries [v] or alternative technologies (RPC, CORBA).

5.1.3 Method & Implementation

Our demonstration also illustrates data interoperability.

Our demonstration involves a prototypical implementation<sup>1</sup> of the knowledge cloud architecture ('Klowid') and two cloud-enabled client programs. One client program ('MyFF') uses *bookmarks* (in reference to Firefox); another ('MyIE') uses *favorites* (in reference to Internet Explorer), as shown in figure 13. These models are similar but not trivially so; for example: bookmarks are *tagged* with multiple categories whereas favorites are *contained* by a single folder in a hierarchy.



**Figure 13:** The favorite and bookmark domain models for client programs MyIE and MyFF, respectively. The knowledge cloud will facilitate the exchange of information between the different models.

We implement two knowledge cloud configurations.

The knowledge cloud has a configuration of implemented knowledges, offers, feeds and agents; these implementations provide or synchronize (transform) favorite and bookmark models. We implement two cloud configurations:

<sup>1</sup> The source code for this project is available on-line via <http://klowid.eu/>.

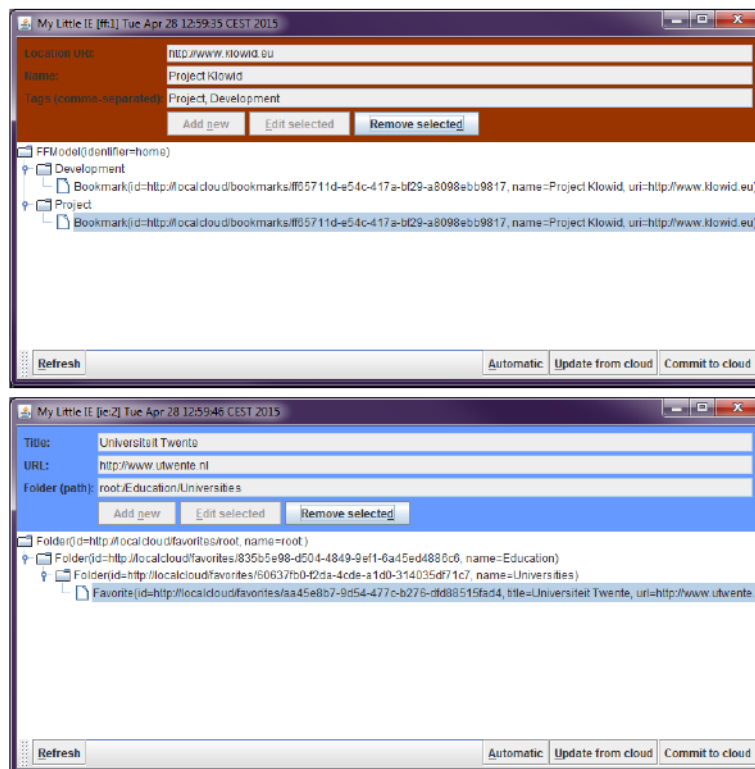
- one applies transformation rules and uses RDF (section §2.5.1.2) and Apache Jena [w];
- one applies direct manipulation of Java objects and is called the POJO-approach ('Plain Old Java Objects').

As required by the proposed new paradigm, the client program implementations are oblivious to the knowledge cloud configuration and accept either.

#### 5.1.3.1 *Demonstration client programs*

Our demonstration client programs are simple bookmark/favorite managers (shown in figure 14). These client programs do not concern themselves with storage and *only* interact with their bookmark/favorite model and the knowledge cloud as follows:

1. client programs query the cloud for knowledge with a specified domain (i.e., 'has BookmarkModel' and 'has FavoriteModel' respectively);
2. client programs freely mutate the bookmark/favorite model (e.g., per instruction by the user);
3. client programs commit and update knowledge model changes to/from the cloud.



**Figure 14:** Screenshots of 'MyFF' and 'MyIE' demonstration programs.

### 5.1.3.2 Knowledge cloud prototype

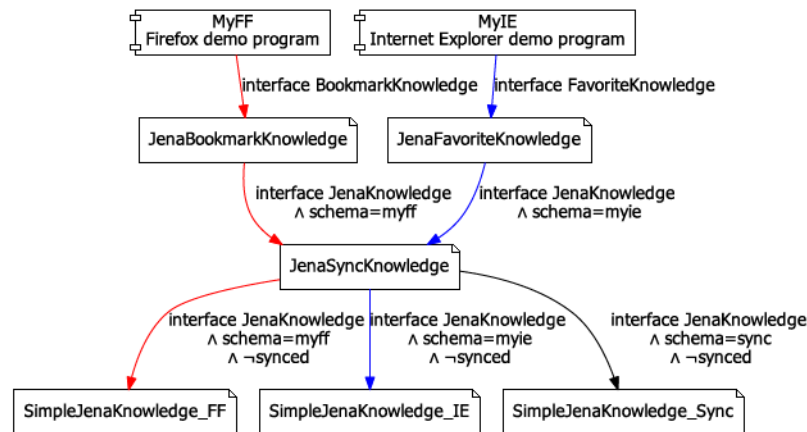
Different knowledge cloud configurations may answer the same knowledge question differently.

Our knowledge cloud prototype implements all components of our architectural design (section §4.2). Our two configurations (RDF-Jena versus POJO) have distinct implementations of knowledge, offers, feeds and agents. Different configurations allow the knowledge cloud to construct different hierarchies of different knowledge instances to satisfy the same client-specified domain. Both configurations have knowledge implementations that simply maintain some bookmark/favorite model in memory and implementations that maintain a correspondence between some favorite knowledge and some bookmark knowledge.

### 5.1.3.3 Configuration A: transformation using RDF and Apache Jena

The RDF-Jena-configuration uses graph storage and transformation.

We implemented agents, feeds, offers and knowledges to transform bookmarks to/from favorites via RDF models. By using generic RDF (graph) models, we could use existing technologies: graph-based inference engines (to transform models), Object-RDF-(de)serialization frameworks (to possibly save and load bookmark/favorite models from disk). In this configuration, the knowledge cloud orchestrates knowledge as shown in figure 15.



**Figure 15:** The knowledge hierarchy for the two client programs that would be orchestrated with a RDF-Jena cloud configuration. Edges denote dependencies on knowledge and are labelled by the domain specification. **Red** edges denote bookmark-modelled information; **blue** edges denote favorite-modelled information.

RDF graphs consist of a set of triples (*subject*, *predicate*, *object*) in which each component is some node (a literal, a Uniform Resource Identifier, a blank node<sup>2</sup>). Apache Jena implements different RDF graph models. Some implementations maintain triples in memory or in storage. Other implementations build on other models to *infer* triples or *combine* graphs.

<sup>2</sup> RDF defines blank nodes as nodes with *identity* but without URI. Blank nodes refer to anonymous resources and may only be used as *subject* or *object*.



Jena's inference graph model binds to another model and uses a reasoner [x] to infer the addition/removal of RDF triples from the source model. Jena implements reasoners for the RDF Schema and OWL standards as well as a 'generic rule reasoner' for custom transformations.

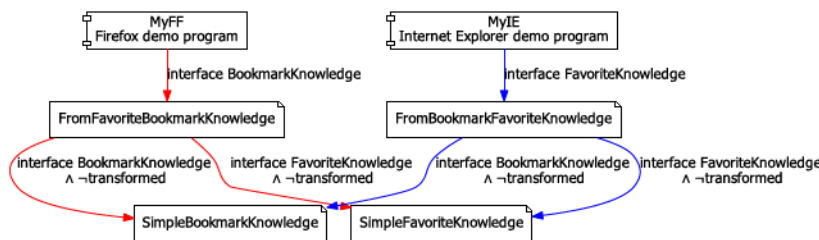
We configured a generic rule reasoner with custom transformation rules (see section §A.1 on page 71) to synchronize RDF-representations of favorites and bookmarks (see section §A.2 on page 72). We used the Jenabean library [y] to (de)serialize Java bookmark/favorite objects to/from RDF.

Unfortunately, this approach turned out to be invasive (model had to be adapted for Jenabean), buggy, incomplete (deletions are not supported), bloated (transformation file is large), slow and unstable (infinite loops).

#### 5.1.3.4 Configuration B: synchronization using POJO

In response to the (unexpectedly many) difficulties with the RDF-Jena-approach, we implemented agents, feeds, offers and knowledges to transform bookmarks to and from favorites with Java code. In this configuration, the knowledge cloud orchestrates knowledge as shown in figure 16.

The POJO-configuration uses Plain Old Java Objects to store and transform information.



**Figure 16:** The knowledge hierarchy for two client programs that would be orchestrated with a POJO cloud configuration. Edges denote dependencies on knowledge and are labelled by the domain specification. Red edges denote bookmark-modelled information; blue edges denote favorite-modelled information.

This approach turned out to work very well; it is complete, fast, maintainable and easy to implement.

#### 5.1.4 Discussion

Given the purpose of our demonstration (section §5.1.1), we conclude the following.

##### 5.1.4.1 Architecture design is adequate (for our demonstration)

For (at least) the limited scope of this demonstration, the architecture is valid. The prototypical implementation of the knowledge cloud architecture (see figure 6) satisfies the needs of this demonstration. In particular, our demonstration shows that the architecture has adequate abstractions to support completely different knowledge cloud configurations (i.e., the RDF-Jena versus POJO approaches).

Our architecture supports different configurations well.

Our demonstration is too artificial to conclude overall architectural adequacy, but it does permit us to present our architecture as a solid initial design for future implementations. We suspect storage-related concerns (such as database transactions) and realistic imperfections (failing connections, misbehaving agents or clients) may require adaptations to the architecture we designed.

#### 5.1.4.2 *Knowledge clouds support data interoperability*

The demonstration shows that the knowledge cloud supports data interoperability by facilitating transformations between heterogeneous models with different knowledge cloud configurations. The flexibility of knowledge cloud configurations turns out to be of significant value.

Our first approach (RDF-Jena; figure 15) to facilitate data interoperability failed. We assumed that RDF's general-purpose nature and the availability of tools and libraries would *ease* transformation between models, but this assumption was wrong. Our second approach (POJO; figure 16) was to simply 'adapt' one information model into the other with Java code, thus avoiding serialization, deserialization, RDF-models, inference engines etc. The second approach was successful and easier.

Programs have information models (e.g., in-memory Java objects) with logic to maintain consistency and integrity (e.g., default values in getters, validation logic in setters, bidirectional relations etc.). *Data-level* transformations (like the RDF-Jena approach) operate on raw *data representations* of the information models, bereft of that information model logic. *Information-level* transformations (like the POJO approach) access and manipulate information models like the client programs do and reuse that logic. Therefore, information-level transformations have many benefits over data-level transformations.

Data-level transformations resemble traditional approaches to data interoperability (without a knowledge cloud): distinct programs share a common data format and re-interpret the stored data into their own (heterogeneous) information models. Information-level transformations avoid data formats altogether and directly manipulate in-memory information models. As maintainer of information models, the knowledge cloud is particularly well-suited for this approach to data interoperability.

#### 5.1.4.3 *Knowledge clouds improve user control*

The user configures the knowledge cloud to specify how data is managed and stored, thereby maintaining control over his data. The user would directly benefit from any configurational flexibility. Our demonstration shows this flexibility: the knowledge cloud may use completely different configurations (RDF-Jena versus POJO) to provide information models to programs, without notice or change to those client programs.

A knowledge cloud configuration requires implementations of knowledge, offers, feeds and agents to transform and provide model instances (from storage). Knowledge cloud configurations vary greatly in the required levels of effort, boilerplate code and overall quality.

The RDF-Jena-configuration failed; the POJO-configuration succeeded.

Transforming information is better than transforming data.

Client programs are not concerned or affected by the knowledge cloud configuration.

Knowledge cloud configurations are not created equal.

#### 5.1.4.4 *Cloud-enabled programs are feasible (for our demonstration)*

The demonstration confirms that implementing cloud-enabled programs is feasible. Cloud-enabled programs must query a knowledge cloud for knowledge, commit changes to the knowledge and handle (remote) updates to the knowledge.

Simple cloud-enabled programs are feasible.

Client programs get knowledge models from the cloud; the cloud ensures this knowledge remains up-to-date by propagating future changes to the client. Programs must expect the models they get from the knowledge cloud to change beyond their control. Managing this volatility may be difficult; we do not specify how programs should handle updates. As the scope of our demonstration is limited, the feasibility of implementing more complex cloud-enabled *practical* programs (*with* support for storage and inter-process communication) is uncertain. The demonstration does adequately illustrate the concrete application of the concept ‘knowledge cloud’.

## 5.2 THEORETICAL CASE-STUDY: HORDE

Our proposed data management paradigm re-assigns storage logic from programs to the cloud. To study the feasibility (and wisdom) of this approach, we should apply this reassignment on existing complex software. Because adapting source code for this experiment is infeasible, we simulate it by studying the modules and their interdependencies of an existing software program, and determining which modules should be implemented in the cloud (rather than in the program).

We simulate the separation of storage and programs.

We hypothesise that using a knowledge cloud for complex programs could significantly reduce program complexity. We choose a program to represent the class of complex programs with rich information models: Horde. We analyse the Horde groupware product (section §2.5.3) because it is open-source, complex, modular and has many features common to other programs. Moreover, the application domain (Personal Information Management) is very suitable for management by a knowledge cloud.

Hypothesis: the knowledge cloud reduces program complexity.

### 5.2.1 Method

Our experiment involves analysing individual Horde-modules and decide for each whether the module should...

We decide whether a Horde-module should be in the cloud.

- be *cloudified*: functionally re-implemented in the cloud; (E.g., modules that implement protocols, data formats or database access.)
- be *duplicated*: be implemented by both the cloud and Horde; (E.g., components for exception handling, translations, DNS, HTTP etc.)
- be *retained*: remain implemented by only Horde. (E.g., user interface modules.)

We consider the dependencies of modules.

By cloudifying modules, we move program logic (and the corresponding complexity) from programs into the cloud. At worst, this extraction only *moves* complexity—which is still desirable. At best, some complexity may be easier to implement in the knowledge cloud or become redundant altogether. Modules may have dependencies on other modules. Cloudifying one module requires the recursive cloudification (or duplication) of its dependencies (i.e., the transitive closure). Duplicating a module requires the duplication of its dependencies.

We mark modules for cloudification or duplication.

To guide our analysis of Horde-modules and their interdependencies, we built a tool. This tool visualizes a module dependency tree as a Graphviz [z] graph, as shown for an artificial program in figure 17. The tool also allows us to *mark* modules for cloudification (red) or duplication (pale blue), as shown in figure 18. Of the remaining ‘unmarked’ modules (generally white), the tool marks those that depend on cloudified modules (green) or are depended on by cloudified modules (orange); these modules are candidates for subsequent cloudification or duplication.

### 5.2.2 Results

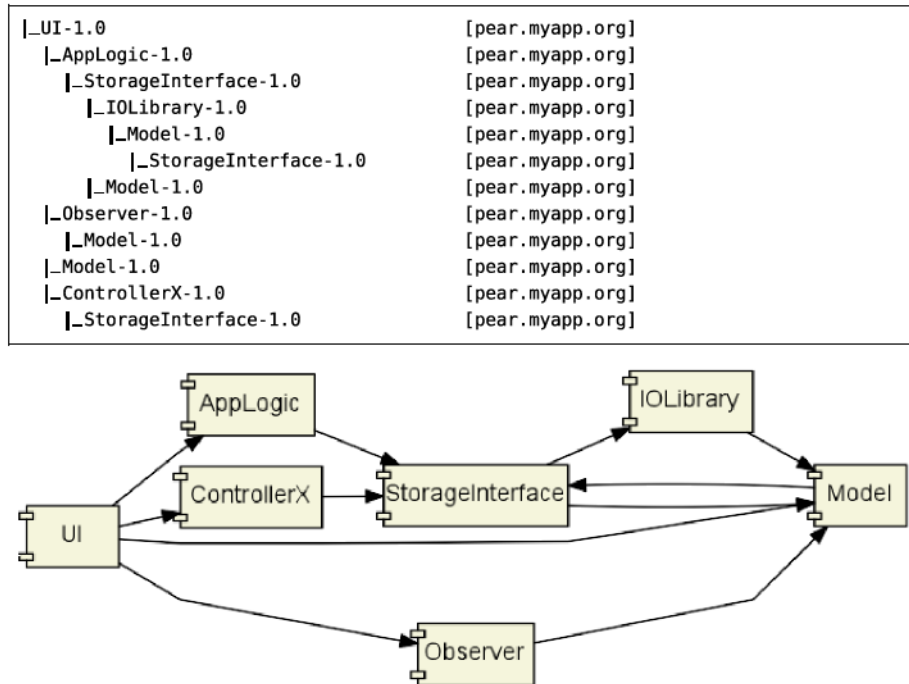
We extracted Horde’s module dependency tree using a Horde developer tool [aa]. We marked Horde-modules for cloudification or duplication to our satisfaction in 11 iterations. For each iteration...

- we marked a module for cloudification (generally a module that was marked as candidate (green) in a previous iteration);
- and recursively cloudified or duplicated all its dependencies (marked orange).

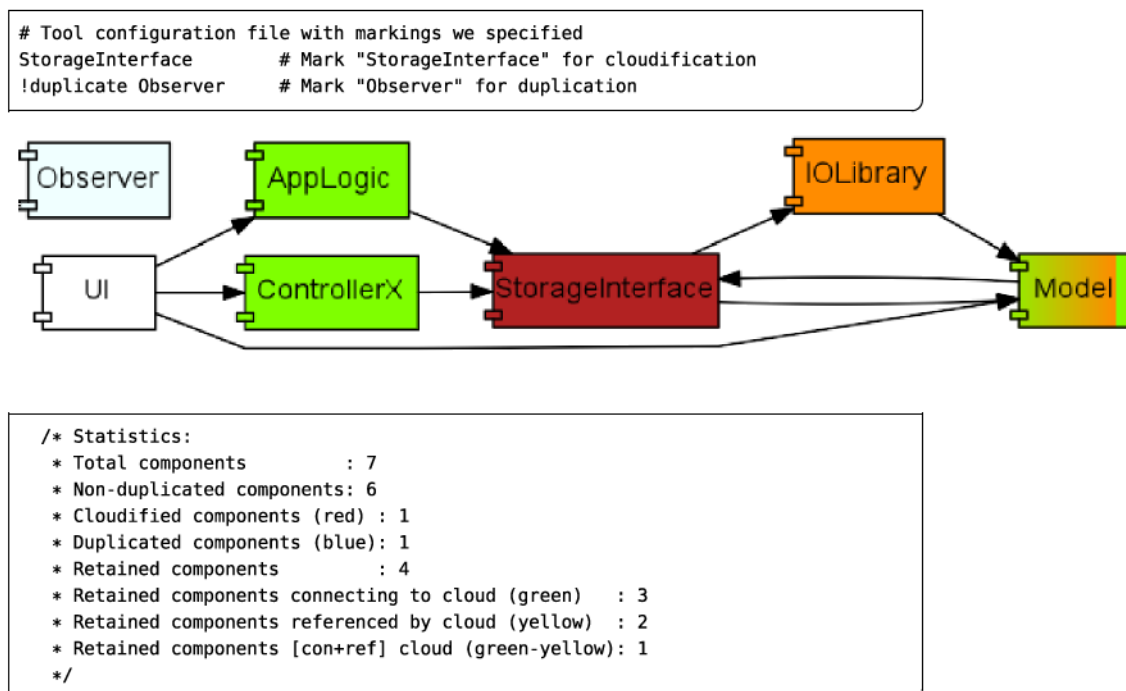
The results of each iteration are summarised in table 2. The graph of Horde-modules with markings that results from the final iteration is shown in figure 19.

Iteration	1	2	3	4	5	6	7	8	9	10	11
All modules:	147	147	147	147	147	147	147	147	147	147	147
Non-duplicated modules:	137	124	123	122	121	121	120	120	119	116	116
Cloudified modules:	1	25	28	32	33	34	37	38	39	47	49
Duplicated modules:	10	23	24	25	26	26	27	27	28	31	31
Retained modules:	136	99	95	90	88	87	83	82	80	69	67
Retained modules connecting to cloud:	14	24	22	20	19	18	15	14	13	12	12

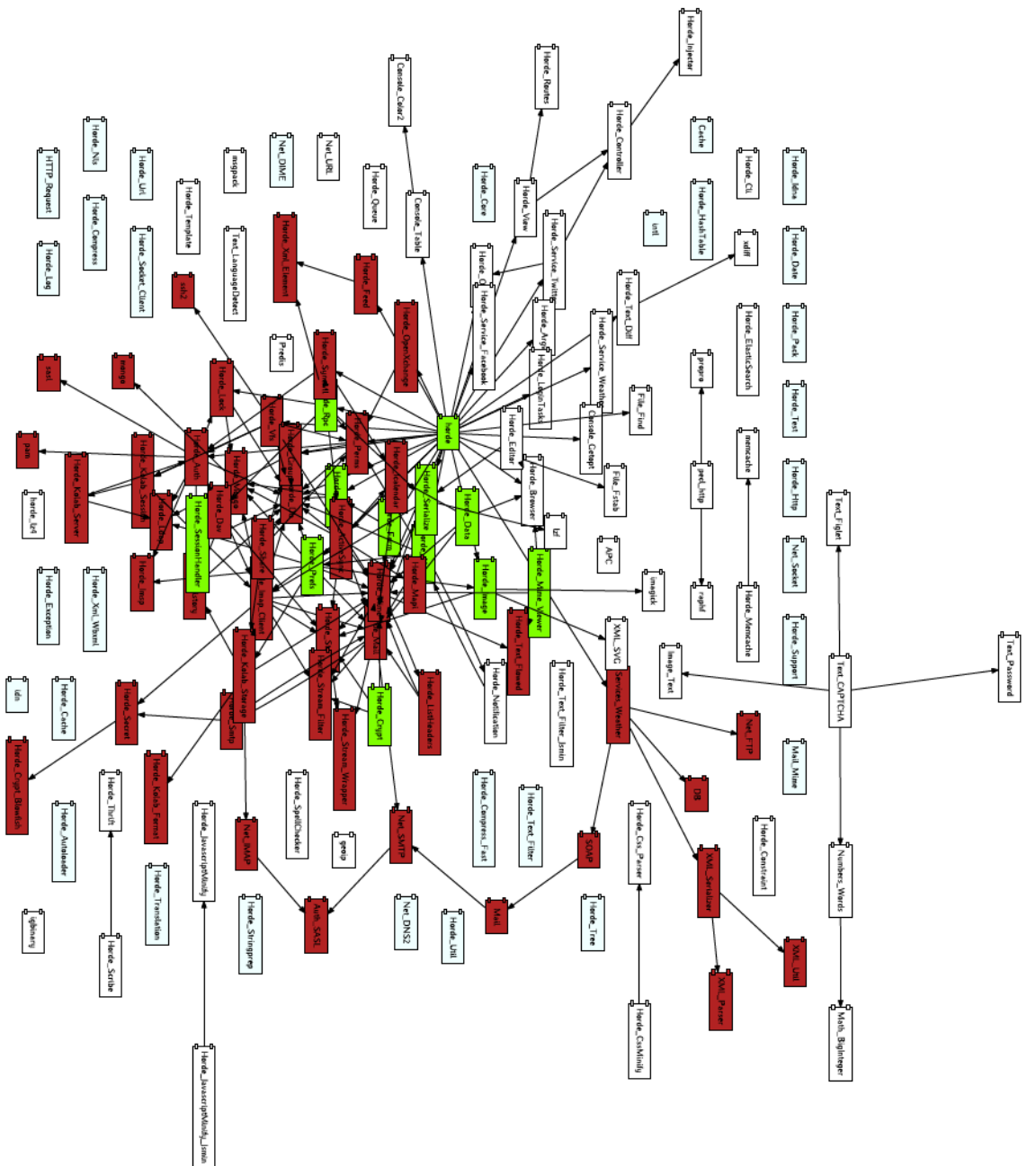
**Table 2:** Summary of the different iterations.



**Figure 17:** Our tool transforms a module dependency tree into a Graphviz graph.



**Figure 18:** Our tool visualizes modules marked for cloudification or duplication and marks the incoming and outgoing dependencies of cloudified modules.



**Figure 19:** Dependency graph for the final cloudification list, showing the set of cloud modules (red) and set of Horde-modules that connect to the cloud (green). Beige modules are Horde-modules without particular relevance to the cloud; the white modules that were excluded from the analysis.



### 5.2.3 Discussion

Our module analysis proposes that of Horde's 147 modules (100%), 31 (21%) be duplicated—these modules offer generic functionality and are not particular to either the knowledge cloud or Horde. Of the remaining modules, 67 (46%) would be retained by Horde while 49 modules (33%) could be extracted from Horde and re-implemented in the cloud. Given that those cloudifiable modules tend to implement complex reusable features (e.g., protocols, services, data formats), Horde's complexity may be reduced *significantly*. To cloudify those 49 modules, 12 modules (retained by Horde) would have to be adapted to connect to and use the cloud. Some of those 12 retained modules already implement intercommunication services (e.g., Remote Procedure Call, session handlers).

We hypothesised that using a knowledge cloud for complex programs (like Horde) could significantly reduce program complexity. We find that our analysis supports our hypothesis. Horde was chosen to represent the class of complex programs with rich information models. Therefore, we are confident other programs may benefit from complexity reduction by using a knowledge cloud as well.

Horde could extract 33% of its modules to the knowledge cloud.

Program complexity is indeed likely to be significantly reduced.

## 5.3 RESULTS

We investigated the the potential use of a knowledge cloud practically with a demonstration using a knowledge cloud prototype (section §5.1) and theoretically with a case-study (section §5.2). Their results allow us to assess to what extent our proposed knowledge cloud yields the claimed benefits (section §3.2.3). (Our validation efforts are too limited in scope to consider the detailed functional requirements in section §3.3.)

### 5.3.1 Programs may be less complex

Our validation efforts confirm the knowledge cloud can reduce program complexity in some cases.

[Q1]

#### 5.3.1.1 *Because storage-agnosticism absolves from storage-related concerns*

Mostly true, but with significant unknowns.

requirement I

The case-study shows that Horde (an enterprise-ready groupware application) has 49 modules (33% of 147) that may be extracted to the cloud. Most of these modules handle storage-related complexity: protocols, data formats, services. The case-study indicates that programs can indeed expect a significant complexity reduction from using the cloud.

We have not validated whether our architecture is adequate for managing transactions and consistency guarantees (section §2.1.3) and therefore cannot claim if all common storage-related concerns can be dealt with by our proposed architectures.



### 5.3.1.2 *Because context-adaptivity absolves from program-related concerns*

requirement II

Undetermined for tested cases; unknown for the rest.

The demonstration shows that the knowledge cloud can successfully implement program-related concerns (such as storage abstraction and interoperability), completely transparent to the client programs.

The demonstration programs are artificial and knowledge cloud implementation is a prototype. We implemented two knowledge cloud configurations with opposite results. One configuration *introduced* undesirable complexity while the other was nearly trivial to implement. The demonstration clearly illustrates that using the cloud is feasible for *some* programs and *some* concerns in *some* configurations but the scope of our demonstration is too limited to claim more.

### 5.3.2 Data is more interoperable

[Q2]

Our validation efforts confirm the knowledge cloud can improve data interoperability.

#### 5.3.2.1 *Because storage-agnosticism unifies models from heterogeneous data sources*

requirement III

Partly true, partly unknown.

The demonstration shows two approaches to synchronize heterogeneous data models (bookmarks versus favorites). One approach failed and another succeeded, which reinforces the value of storage-agnosticism to programs and the knowledge cloud.

The demonstration does *not* maintain (a federation of) *external* data sources; all information models are maintained in-memory. External data sources are fundamental to production-ready knowledge clouds and it is unknown how well the knowledge cloud could expose data from external data sources and deal with their volatile content and availability.

#### 5.3.2.2 *Because context-adaptivity transforms and adapts data to requirements*

requirement IV

Mostly true, partly unknown.

The demonstration presented two approaches to model synchronization; each approach was implemented as a configuration of knowledges, offers, feeds and agents in the knowledge cloud. With either configuration, the knowledge cloud built correct knowledge hierarchies to provide the client programs with tailored information models. Neither the knowledge cloud prototype nor the client programs required adaptation to the chosen approach.

Our knowledge cloud prototype implements our architecture design. We have not validated if production environment conditions (e.g., with more complex or conflicting requirement specifications) can be properly resolved with our current architecture.

### 5.3.3 Users have more control of their data

Our validation efforts confirm the knowledge cloud can improve the user's control over their data. [Q3]

#### 5.3.3.1 *Because storage-agnosticism enables users to dictate data storage and storage policies*

Mostly true, partly unknown.

requirement [V](#)

The demonstration shows the knowledge cloud supports significant configurational flexibility by supporting two different configurations (RDF-Jena versus POJO). The user controls and configures the knowledge cloud and therefore directly benefits from this flexibility.

We have not investigated what requirements the user has in practice and whether the demonstrated configurational flexibility is sufficient to meet them. We also have not investigated if our architecture permits more complex (and possibly conflicting) configurations.

#### 5.3.3.2 *Because context-adaptivity enables users to dictate data usage and processing policies*

Mostly true, partly unknown.

requirement [VI](#)

For each knowledge cloud configuration in our demonstration, there exist knowledge implementations to simply provide an information model from memory and implementations to synchronize two heterogeneous models. The user can disable or install implementations and thereby control how information are used and processed in the knowledge cloud.

Our demonstration does not support the installation of custom agents by users.

### 5.3.4 Developers should write less inessential code

Our validation efforts suggest the knowledge cloud can reduce the amount of inessential code developers write. [Q4]

#### 5.3.4.1 *Because storage-agnosticism obsoletes data models, storage APIs and exchange protocols*

Probably true.

requirement [VII](#)

The case-study shows the knowledge cloud effectively absolves client program Horde from many storage-related concerns. Horde would no longer need to implement data models, storage APIs and exchange protocols as this logic would re-implemented in the knowledge cloud instead.

While Horde's storage-related components may be obsoleted, other programs may have storage-related requirements that the knowledge cloud might not easily satisfy (e.g., streaming multimedia content, peer-to-peer protocols).

#### 5.3.4.2 *Because context-adaptivity centralizes reusable logic*

requirement VIII

Probably true.

The case-study shows the knowledge cloud could extract (i.e., centralize) 49 storage-related Horde-modules. These modules implemented data formats, conversions, protocols, services—most of them re-usable for satisfying requirements of other client programs.

#### 5.3.5 Conclusion

We implemented our knowledge cloud design as a prototype and we investigated how programs use the knowledge cloud with a theoretical case-study and practical demonstration. Our validation efforts were limited in scope but nonetheless insightful. In particular, we found that using a knowledge cloud may shrink programs significantly. We also found that the knowledge cloud configuration greatly affects the difficulty of implementing program-related concerns (such as synchronizing heterogeneous information models). Some configurations require non-trivial and cumbersome implementations (RDF-Jena) while others (POJO) are almost trivial.

Our validation experiments confirm the potential of the knowledge cloud.

Our validation efforts confirm the *potential* of our proposed data management paradigm and the knowledge cloud to our satisfaction. Our validation efforts have not uncovered critical flaws or insurmountable obstacles. However, our validation efforts can only show the presence, not the absence of weaknesses. We are particularly wary of how the complexity that the knowledge cloud imports from programs (i.e., ‘cloudified modules’) should be managed and how practical non-artificial conditions are handled by our architecture. The best validation of the knowledge cloud (in concept and architectural design) would be an implementation for production-use. We are optimistic that such an implementation reinforces the supposed advantages of the knowledge cloud and improves upon its definition and our design.

Further research is necessary.

# 6 | CONCLUSION

A data management paradigm is a set of *concepts* and *thought patterns* on data management. Today, this paradigm requires programs to directly access and manage storage; the consequences are significant:

- Programs are unnecessarily complex;
- Data is badly interoperable;
- Users are not in control of their data;
- Developers must write inessential code.

To ameliorate these problems, we proposed a new data management paradigm with a central information provision and exchange service: the *knowledge cloud*. In the new paradigm, programs connect to the knowledge cloud to access information in a model the program specifies. The knowledge cloud transforms between models and manages their storage, invisibly to the program.

The knowledge cloud enables two fundamental qualities of the new paradigm: storage-agnosticism and context-adaptivity.

**storage-agnosticism** permits users and programs to remain ignorant of *storage concerns*. Rather than managing data (in some format from some data source), a storage-agnostic program specifies an information model and expects the information from the knowledge cloud to conform.

**context-adaptivity** permits other parties to invisibly manipulate the interaction between programs and storage. By interception and adaptation, other parties (contexts) could support *cross-cutting concerns* (authorization, data transformation) without the program noticing.

We investigated various concerns the knowledge cloud needs to handle: storage-related concerns, program-related concerns and cloud-specific concerns. We found there is a lot of related work on modelling, transforming, accessing and integrating data.

We elaborated on our proposal for a knowledge cloud by reasoning for its existence, how it enables *storage-agnosticism* and *context-adaptivity*, what requirements it should satisfy and what the roles of programs, data, users and developers are in the new data management paradigm.

We designed a generic knowledge cloud architecture with fundamental concepts, which serves as a reference framework for future comparisons and implementations of knowledge clouds.

We validated our proposal for a new data management paradigm by simulating the effects of the proposed paradigm in a case-study of existing software. We validated the design and demonstrated the use of the knowledge cloud by implementing a knowledge cloud prototype and demonstration programs.

[chapter 1](#) on page 1

[chapter 2](#) on page 7

[chapter 3](#) on page 19

[chapter 4](#) on page 29

[chapter 5](#) on page 45

## 6.1 MAIN RESEARCH FINDINGS

The goal of this research project was to present and substantiate our vision for a data management paradigm in which programs and storage are separated by a knowledge cloud. Our research question was:

[Q0] on page 4

“What are the effects for users and developers if programs interacted with a single knowledge cloud for information access and storage, rather than with various storage mechanisms separately?”

Our validation efforts *confirm the potential* of the proposed knowledge cloud and the new data management paradigm to ameliorate the identified problems of the current paradigm for users, their data, developers and their programs. By using a knowledge cloud...

- [Q1] ● programs *may* be less complex;
- [Q2] ● data *is* more interoperable;
- [Q3] ● users *have* more control of their data;
- [Q4] ● developers *should* write less inessential code

Our validation efforts inspire confidence, but additional research on the extent and conditions of these benefits is necessary.

## 6.2 FUTURE WORK & RECOMMENDATIONS

We recommend the next step to be the implementation of a production-ready knowledge cloud and the implementation of productive programs to use this knowledge cloud. These efforts should yield insights in various practical and fundamental matters, such as:

- How should client processes interface with the cloud?  
(Some form of Inter-Process Communication, shared memory, system library, CORBA or network protocol? How do programs share information models?)
- How should human users manage and interface with the knowledge cloud?
- How deep can or should the knowledge cloud be integrated in operating systems?
- How can a federation of data sources be managed?
- How should atomic (database) transactions be expressed and managed?
- How can the knowledge orchestration process be improved?
- How should the user control the knowledge cloud?  
(Should there be a standard knowledge model for ‘storage policies’? Should there be an on-line marketplace to download agents, models and transformers? Should client programs be able to request the installation of agents?)

- How should the knowledge cloud handle and recover from errors?
- How can the knowledge cloud manage the complexity it imports from programs?
- How can the knowledge cloud maintain security and system stability with various interacting agents and knowledge models?





## NON-SCIENTIFIC REFERENCES

- [a] Ramesh Srinivasan (Winhelponline). Preserve the Order of Internet Explorer Favorites When Transferring to Another PC. 2008-08-18.  
URL <http://www.winhelponline.com/blog/preserve-order-ie-favorites-transferring-to-another-pc/>.
- [b] Mozilla. Run your own Firefox Accounts Server. Retrieved 2015-03-15.  
URL <http://docs.services.mozilla.com/howtos/run-fxa.html>.
- [c] Oracle. Java SE Technologies – Database. Retrieved 2015-03-14.  
URL <http://www.oracle.com/technetwork/java/javase/jdbc/index.html>.
- [d] Marget Rouse (TechTarget). Database-agnostic. 2011-02.  
URL <http://searchdatamanagement.techtarget.com/definition/database-agnostic>.
- [e] Martin Fowler. FluentInterface. 2005-12-20.  
URL <http://martinfowler.com/bliki/FluentInterface.html>.
- [f] Microsoft. LINQ (Language-Integrated Query). Retrieved 2015-03-14.  
URL <http://msdn.microsoft.com/en-us/library/bb397926.aspx>.
- [g] Martin Fowler. NosqlDefinition. 2012-01-09.  
URL <http://martinfowler.com/bliki/NosqlDefinition.html>.
- [h] Tony Marston. What is the 3-Tier Architecture?. 2012-10-14.  
URL <http://www.tonymarston.net/php-mysql/3-tier-architecture.html>.
- [i] Jeffrey Palermo. The Onion Architecture: part 1. 2008-07-29.  
URL <http://jeffreypalermo.com/blog/the-onion-architecture-part-1/>.
- [j] Ted Neward. The Vietnam of Computer Science. 2006-06-26.  
URL <http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx>.
- [k] W3C. W3C Semantic Web Activity. 2013-12-11.  
URL <http://www.w3.org/2001/sw/>.
- [l] W3C Working Group. RDF 1.1 Primer. 2014-06-24.  
URL <http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/>.
- [m] W3C. SPARQL 1.1 Overview. 2013-03-21.  
URL <http://www.w3.org/TR/sparql11-overview/>.

[n] W3C. A Semantic Web Primer for Object-Oriented Software Developers. 2006-03-09.

URL <http://www.w3.org/2001/sw/BestPractices/SE/ODSD/>.

[o] W3C. A Direct Mapping of Relational Data to RDF. 2012-09-27.

URL <http://www.w3.org/TR/rdb-direct-mapping/>.

[p] Paul Thurrott. Windows Storage Foundation (WinFS) Preview. 2005-08-30.

URL [http://web.archive.org/web/20070901005702/http://www.winsupersite.com/showcase/winfs\\_preview.asp](http://web.archive.org/web/20070901005702/http://www.winsupersite.com/showcase/winfs_preview.asp).

[q] Richard Grimes (Microsoft). Revolutionary File Storage System Lets Users Search and Manage Files Based on Content. 2004-08-27.

URL <https://msdn.microsoft.com/en-us/magazine/cc164028.aspx>.

[r] The Horde Project. Retrieved 2015-03-14.

URL <http://www.horde.org/>.

[s] OpenLink. Virtuoso Universal Server. Retrieved 2015-03-14.

URL <http://virtuoso.openlinksw.com/>.

[t] Mozilla. Resource Description Framework (RDF). 2007-09-27.

URL <http://www-archive.mozilla.org/rdf/doc/>.

[u] Eric Evans and Martin Fowler. Specifications. Retrieved 2015-02-16.

URL <http://martinfowler.com/apsupp/spec.pdf>.

[v] Daniel Gredler. Java Remoting: Protocol Benchmarks. 2008-01-07.

URL <http://daniel.gredler.net/2008/01/07/java-remoting-protocol-benchmarks/>.

[w] Apache Jena. Retrieved 2015-03-20.

URL <http://jena.apache.org/>.

[x] Apache Jena. Reasoners and rule engines: Jena inference support. Retrieved 2015-03-20.

URL <http://jena.apache.org/documentation/inference/>.

[y] thewebsemantic. jenabean. Retrieved 2015-03-20.

URL <https://code.google.com/p/jenabean/>.

[z] Graphviz. Graph Visualization Software. Retrieved 2015-03-16.

URL <http://www.graphviz.org/>.

[aa] Horde. Doc/Dev/Component/Components. 2015-03-16.

URL <http://wiki.horde.org/Doc/Dev/Component/Components>.

## BIBLIOGRAPHY

- [1] Akash Mitra. Classifying data for successful modeling, March 2012. (Cited on page [20](#).)
- [2] Debra M. Amidon. *Innovation Strategy for the Knowledge Economy*. Routledge, 1997. ISBN 9780750698412. (Cited on page [20](#).)
- [3] Peter Bailis and Ali Ghodsi. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Queue*, 11(3):20:20–20:32, March 2013. ISSN 1542-7730. doi: 10.1145/2460276.2462076. (Cited on page [9](#).)
- [4] K. Beck and W. Cunningham. A Laboratory for Teaching Object Oriented Thinking. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '89, pages 1–6, New York, NY, USA, 1989. ACM. ISBN 0-89791-333-7. doi: 10.1145/74877.74879. (Cited on page [33](#).)
- [5] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data—the story so far. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(3):1–22, 2009. (Cited on page [16](#).)
- [6] David Boddy, Albert Boonstra, and Graham Kennedy. *Managing Information Systems: An Organisational Perspective*. Pearson Education, January 2005. ISBN 9780273686354. (Cited on page [20](#).)
- [7] Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, page 338–347, 2006. (Cited on page [12](#).)
- [8] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: resourceful lenses for string data. In *ACM SIGPLAN Notices*, volume 43, pages 407–419. ACM, 2008. 00114. (Cited on page [12](#).)
- [9] Peter Buneman and Robert E. Frankel. FQL: a functional query language. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 52–58. ACM, 1979. 00186. (Cited on page [16](#).)
- [10] Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974. (Cited on page [8](#).)

- [11] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976. (Cited on page 15.)
- [12] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. (Cited on pages 8 and 14.)
- [13] Edgar F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. (Cited on page 34.)
- [14] Robert C. Daley and Peter G. Neumann. A general-purpose file system for secondary storage. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, page 213–229, 1965. (Cited on page 1.)
- [15] Ramakanth Subrahmanya Devarakonda. Object-relational database systems — the road ahead. *Crossroads*, 7(3):15–18, March 2001. ISSN 1528-4972. doi: 10.1145/367884.367895. (Cited on pages 14 and 15.)
- [16] Vadim Eisenberg. Programming Applications over the Semantic Web, September 2011. 00000. (Cited on page 17.)
- [17] Eric Evans. *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004. ISBN 9780321125217. 00712. (Cited on page 27.)
- [18] Gustav Fahl and Tore Risch. Amos II introduction. *Tutorial, Dept. of Information Science, Uppsala University, Sweden*, 1999. (Cited on page 18.)
- [19] Lee Feigenbaum, Ivan Herman, Tonya Hongsermeier, Eric Neumann, and Susie Stephens. The semantic web in action. *Scientific American Magazine*, 297(6):90–97, 2007. (Cited on page 16.)
- [20] J. Nathan Foster, Michael B. Greenwald, Christian Kirkegaard, Benjamin C. Pierce, and Alan Schmitt. Exploiting schemas in data synchronization. In *Database Programming Languages*, page 42–57, 2005. (Cited on page 12.)
- [21] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17–es, May 2007. ISSN 01640925. doi: 10.1145/1232420.1232424. (Cited on page 12.)
- [22] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2003. ISBN 9780321127426. (Cited on page 9.)
- [23] Antony Galton. Temporal Logic. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Fall 2008 edition, February 2008. (Cited on page 36.)

- [24] Seth Gilbert and Nancy Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, 33(2): 51–59, June 2002. ISSN 0163-5700. doi: 10.1145/564585.564601. (Cited on page 9.)
- [25] Jim Gray and others. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, pages 144–154, 1981. (Cited on pages 8 and 9.)
- [26] Thomas R. Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993. (Cited on page 11.)
- [27] Theo Haerder and Andreas Reuter. Principles of Transaction-oriented Database Recovery. *ACM Comput. Surv.*, 15(4):287–317, December 1983. ISSN 0360-0300. doi: 10.1145/289.291. (Cited on page 8.)
- [28] Michael Hammer and Dennis McLeod. The Semantic Data Model: A Modeling Mechanism for Data Base Applications. In *Proceedings of the 1978 ACM SIGMOD International Conference on Management of Data*, SIGMOD '78, pages 26–36, New York, NY, USA, 1978. ACM. doi: 10.1145/509252.509264.00200. (Cited on page 16.)
- [29] Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. *Proc. of the 2011 Marktoberdorf Summer School*. IOS Press, 2011. (Cited on page 36.)
- [30] Dennis Heimbigner and Dennis McLeod. A federated architecture for information management. *ACM Trans. Inf. Syst.*, 3(3):253–278, July 1985. ISSN 1046-8188. doi: 10.1145/4229.4233. (Cited on page 12.)
- [31] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008. (Cited on page 23.)
- [32] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric Lenses. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 371–384, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926428.00064. (Cited on page 12.)
- [33] C. Ireland, D. Bowers, M. Newton, and K. Waugh. A Classification of Object-Relational Impedance Mismatch. In *First International Conference on Advances in Databases, Knowledge, and Data Applications, 2009. DBKDA '09*, pages 36–43, March 2009. doi: 10.1109/DBKDA.2009.11. (Cited on page 10.)
- [34] Joxan Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 111–119. ACM, 1987. (Cited on page 40.)
- [35] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008. ISSN 01676423. doi: 10.1016/j.scico.2007.08.002.00587. (Cited on page 12.)



- [36] William Kent. Limitations of record-based information models. *ACM Transactions on Database Systems (TODS)*, 4(1):107–131, 1979. 00281. (Cited on page 16.)
- [37] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) version 1.2, February 2015. 00000. (Cited on page 12.)
- [38] Eyal Oren, Benjamin Heitmann, and Stefan Decker. ActiveRDF: Embedding SemanticWeb data into object-oriented languages. *Web Semantics: Science, Services and Agents on the World Wide Web*, 6(3), March 2011. ISSN 1570–8268. 00000. (Cited on page 17.)
- [39] Aris M. Ouksel and Amit Sheth. Semantic interoperability in global information systems. *ACM Sigmod Record*, 28(1):5–12, 1999. (Cited on page 10.)
- [40] Richard F. Paige. *Theory and Practice of Model Transformations: Second International Conference, ICMT 2009, Zürich, Switzerland, June 29–30, 2009, Proceedings*. Springer Science & Business Media, June 2009. ISBN 9783642024078. 00000. (Cited on page 17.)
- [41] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering, 1995*, pages 251–260, March 1995. doi: 10.1109/ICDE.1995.380386. (Cited on page 10.)
- [42] Yannis Papakonstantinou, Serge Abiteboul, and Hector Garcia-Molina. Object fusion in mediator systems. In *VLDB*, volume 96, pages 413–424, 1996. (Cited on page 16.)
- [43] M. Papazoglou, S. Spaccapietra, and Zahir Tari. *Advances in Object-oriented Data Modeling*. MIT Press, 2000. ISBN 9780262161893. (Cited on page 10.)
- [44] Jim Paterson, Stefan Edlich, Henrik Hörning, and Reidar Hörning. Comparing the object and relational data models. *The Definitive Guide to db4o*, page 31–46, 2006. (Cited on pages 14 and 15.)
- [45] Dan Pritchett. BASE: An Acid Alternative. *Queue*, 6(3):48–55, May 2008. ISSN 1542–7730. doi: 10.1145/1394127.1394128. (Cited on page 9.)
- [46] Tore Risch, Vanja Josifovski, and Timour Katchaounov. *Functional data integration in a distributed mediator system*. Springer, 2003. (Cited on pages 16 and 18.)
- [47] Jennifer Rowley. The wisdom hierarchy: representations of the DIKW hierarchy. *Journal of Information Science*, 33(2):163–180, January 2007. ISSN 0165–5515, 1741–6485. doi: 10.1177/0165551506070706. (Cited on page 20.)
- [48] F. Saltor, M. Castellanos, and M. García-Solaco. Suitability of datamodels as canonical models for federated databases. *SIGMOD Rec.*, 20(4):44–48, December 1991. ISSN 0163–5808. doi: 10.1145/141356.141377. (Cited on pages 13 and 15.)

- [49] Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, number 903 in Lecture Notes in Computer Science, pages 151–163. Springer Berlin Heidelberg, 1995. ISBN 978-3-540-59071-2, 978-3-540-49183-5. 00609. (Cited on page 12.)
- [50] David W. Shipman. The functional data model and the data languages DAPLEX. *ACM Trans. Database Syst.*, 6(1):140–173, March 1981. ISSN 0362-5915. doi: 10.1145/319540.319561. (Cited on pages 16 and 18.)
- [51] Edgar H. Sibley and Larry Kerschberg. Data architecture and data model considerations. In *Proceedings of the June 13–16, 1977, national computer conference*, page 85–96, 1977. (Cited on pages 13 and 16.)
- [52] Perdita Stevens. A landscape of bidirectional model transformations. In *Generative and transformational techniques in software engineering II*, pages 408–424. Springer, 2008. 00092. (Cited on page 12.)
- [53] Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software & Systems Modeling*, 9(1):7–20, January 2010. ISSN 1619-1366, 1619-1374. doi: 10.1007/s10270-008-0109-9. 00103. (Cited on page 12.)
- [54] Michael Stonebraker, P Brown, and D Moor. Object-relational DBMS—the next wave. *Informix Software (now part of the IBM Corp. family), Menlo Park, CA*, 1995. (Cited on pages 14 and 15.)
- [55] Muralidhar Subramanian, Vishu Krishnamurthy, and Redwood Shores. Performance challenges in object-relational DBMSs. *IEEE Data Eng. Bull.*, 22(2):27–31, 1999. 00136. (Cited on page 15.)
- [56] Tore Risch. Introduction to object-oriented and object-relational database systems, May 2004. (Cited on page 15.)
- [57] D. C. Tsichritzis and Frederick H. Lochovsky. Hierarchical data-base management: A survey. *ACM Computing Surveys (CSUR)*, 8(1):105–123, 1976. (Cited on page 13.)
- [58] Dennis Tsichritzis and Anthony Klug. The ANSI/X3/SPARC DBMS framework report of the study group on database management systems. *Information Systems*, 3(3):173–191, 1978. ISSN 0306-4379. doi: 10.1016/0306-4379(78)90001-7. (Cited on pages 20 and 22.)
- [59] Umeå universitet. Limitations of the relational model, 2005. (Cited on page 15.)
- [60] Mike Uschold. Knowledge level modelling: concepts and terminology. *The knowledge engineering review*, 13(01):5–29, 1998. (Cited on pages 10, 11, 12, 20, and 21.)







The following listings are related to section §5.1.3.3 ([Configuration A: transformation using RDF and Apache Jena](#)).

## A.1 JENA TRANSFORMATION FILE

Example of RDF graph data (in the Turtle format) and a transformation rule set (in a Jena-particular format). This example transforms a bookmark with two tags to two equivalent Favorites.

```
1 @prefix klowid: <http://eu.klowid/> .
  @prefix sync: <http://eu.klowid.demo.hyperlinks2.sync/> .
  @prefix ff: <http://eu.klowid.demo.hyperlinks2.myff/> .
  @prefix ie: <http://eu.klowid.demo.hyperlinks2.myie/> .
5
  [ bookmark_declare: ... ]           // 1+2 statements
  [ favorite_declare: ... ]           // 1+2 statements
  [ folder_declare: ... ]             // 1+2 statements
10
  [ favorite_path_components_root: ... ] // 4+2 statements
  [ favorite_path_components_subfolders: ... ] // 6+2 statements

  [ dangling_favorite_sync: ... ]     // 6+3 statements
  [ dangling_favorite: ... ]          // 4+2 statements
15
  [ folder_creates_tag:               // 5+2 statements
    (?fd rdf:type ie:Folder)
    noValue(?fd sync:tag)
    (?fd ie:name ?name)
    (?fd sync:path ?path)
    regex(?path "root:(.*)" ?tag)
20
    ->
      print("folder_creates_tag" ?fd ?tag)
      (?fd sync:tag ?tag)
25 ]
  [ tag_creates_folder: ... ]         // 9+8 statements

  [ bookmark_creates_favorites:      // 13+10 statements
    (?b rdf:type ff:Bookmark)
    (?b ff:tags ?tags)
    (?tags ?_ ?tag)
    isDType(?tag, xsd:string)
    (?fd sync:tag ?tag)
    (?fd rdf:type ie:Folder)
    (?fd ie:favorites ?favorites)
    (?b ff:uri ?uri)
    (?b ff:name ?name)
    (?b ff:version ?version)
    uriConcat(?b "#" ?tag ?btag)
    noValue(?f sync:with ?btag)
    uriConcat(?b ".favorite#" ?tag ?f)
40
    ->
```

```

    print("bookmark_creates_favorites" ?b ?f)
    (?f sync:with ?btag)
    (?f sync:with ?b)
    (?b sync:with ?f)
    (?f ie:lastUpdated ?version)
    (?f ie:title ?name)
    (?f ie:url ?uri)
    have(?favorites ?f)
    (?f ie:folder ?fd)
    (?f rdf:type ie:Favorite)
  ]
[ favorites_create_bookmark: ... ]      // 10+11 statements
55 [ bookmark_updates_favorites: ... ]   // 8+4 statements
[ favorites_update_bookmark:           // 8+4 statements
    (?f rdf:type ie:Favorite)
    (?f sync:with ?b)
    (?b rdf:type ff:Bookmark)
    (?f ie:lastUpdated ?ft)
    (?b ff:version ?bt)
    greaterThan(?ft ?bt)
    (?f ie:url ?url)
    (?f ie:title ?title)
65 ->
    print("favorites_update_bookmark" ?f ?b)
    (?b ff:version ?ft)
    reset(?b ff:uri)      (?b ff:uri ?url)
    reset(?b ff:name)     (?b ff:name ?title)
70 ]

// Deleting not supported

```

## A.2 RDF MODELS

Example of RDF graph data (in the Turtle format) and a transformation rule set (in a Jena-particular format). This example shows the source and transformation result of a Bookmark with two tags into two equivalent Favorites.

### A.2.1 Source RDF-model

```

1 <http://localcloud/bookmarks/2d6d09f9-db1f-460a-94d1-c062b347ac9c>
  a          ff:Bookmark ;
  ff:name    "Project Klowid"^^xsd:string ;
  ff:tags    [ a      rdf:Seq ;
5              rdf:_1 "Project"^^xsd:string ;
              rdf:_2 "Development"^^xsd:string
              ] ;
  ff:uri     "http://www.klowid.eu"^^xsd:string ;
  ff:version "2015-04-25T13:28:00.861Z"^^xsd:dateTime .
10
tw:javaclass a owl:AnnotationProperty .

ff:Bookmark a      <http://www.w3.org/2000/01/rdf-#Class> , rdfs:Class ;
tw:javaclass "eu.klowid.demo.hyperlinks2.myff.Bookmark" .

```

## A.2.2 Inferred RDF-model

```

1 <http://localcloud/bookmarks/2d6d09f9-dbf-460a-94d1-c062b347ac9c.favorite#Development>
    a                ie:Favorite ;
    ie:folder        <http://localcloud/sync/Development> ;
    ie:lastUpdated   "2015-04-25T13:28:00.861Z"^^xsd:dateTime ;
5    ie:title        "Project Klowid"^^xsd:string ;
    ie:url           "http://www.klowid.eu"^^xsd:string .

<http://localcloud/bookmarks/2d6d09f9-dbf-460a-94d1-c062b347ac9c.favorite#Project>
10    a                ie:Favorite ;
    ie:folder        <http://localcloud/sync/Project> ;
    ie:lastUpdated   "2015-04-25T13:28:00.861Z"^^xsd:dateTime ;
    ie:title        "Project Klowid"^^xsd:string ;
    ie:url           "http://www.klowid.eu"^^xsd:string .

15 <http://localcloud/sync/Project>
    a                ie:Folder ;
    ie:favorites     [ rdf:_1 <http://localcloud/bookmarks/2d6d09f9-dbf-460a-94d1-c062b347ac9c
                        .favorite#Project> ] ;
    ie:name          "Project"^^xsd:string ;
    ie:parent        <http://localcloud/favorites/root> ;
20    ie:subfolders   [] .

<http://localcloud/sync/Development>
    a                ie:Folder ;
    ie:favorites     [ rdf:_1 <http://localcloud/bookmarks/2d6d09f9-dbf-460a-94d1-c062b347ac9c
                        .favorite#Development> ] ;
25    ie:name        "Development"^^xsd:string ;
    ie:parent        <http://localcloud/favorites/root> ;
    ie:subfolders    [] .

<http://localcloud/favorites/root>
30    a                ie:Folder ;
    ie:favorites     [ a rdf:Seq ] ;
    ie:name          "root:"^^xsd:string ;
    ie:subfolders    [ a      rdf:Seq ;
                        rdf:_1 <http://localcloud/sync/Project> ;
35                        rdf:_2 <http://localcloud/sync/Development>
                        ] .

```