



School of Computing and Communications

A Modeling Language for Multi-tenant Data Architecture Evolution in Cloud Applications

Assylbek Sagitzhanuly Jumagaliyev

M.Sc., International Information Technologies University, 2014

B.Sc., Suleyman Demirel University, 2011

Thesis submitted for the degree of
Doctor of Philosophy

29 April 2019

Abstract

Multi-tenancy enables efficient resource utilization by sharing application resources across multiple customers (*i.e.*, *tenants*). Hence, applications built using this pattern can be offered at a lower price and reduce maintenance effort as less application instances and supporting cloud resources must be maintained. These properties encourage cloud application providers to adopt multi-tenancy to their existing applications, yet introducing this pattern requires significant changes in the application structure to address multi-tenancy requirements such as isolation of tenants, extensibility of the application, and scalability of the solution. In cloud applications, the data layer is often the prime candidate for multi-tenancy, and it usually comprises a combination of different cloud storage solutions such as blob storage, relational and non-relational databases. These storage types are conceptually and tangibly divergent, each requiring its own partitioning schemes to meet multi-tenancy requirements. Currently, multi-tenant data architectures are implemented using manual coding methods, at times following guidance and patterns offered by cloud providers. However, such manual implementation approach tends to be time consuming and error prone. Several modeling methods based on Model-Driven Engineering (**MDE**) and Software Product Line Engineering (**SPL**) have been proposed to capture multi-tenancy in cloud applications. These methods mainly generate cloud deployment configurations from an application model, though they do not automate implementation or evolution of applications.

This thesis aims to facilitate development of multi-tenant cloud data architectures using model-driven engineering techniques. This is achieved by designing and implementing a novel modeling language, **CadaML**, that provides concepts and notations to model multi-tenant cloud data architectures in an abstract way. **CadaML** also provides a set of tools to validate the data architecture and automatically produce corresponding data access layer code. The thesis demonstrates the feasibility of the modeling language in a practical setting and adequacy of multi-tenancy implementation by the generated code on an industrial business process analyzing application. Moreover, the modeling language is empirically compared against manual implementation methods to inspect its effect on developer productivity, development effort, reliability of the application code, and usability of the language. These outcomes

provide a strong argument that the CadaML modeling language effectively mitigates the high overhead of manual implementation of multi-tenant cloud data layers, significantly reducing the required development complexity and time.

Declaration

I declare that the work in this thesis is my own work and has not been submitted either in whole or in part for the award of a higher degree elsewhere. Any sections of the thesis which have been published are clearly identified.

Thesis originally submitted on 29 April 2019

Recommended corrections submitted on 19 June 2019



.....
Assylbek Sagitzhanuly Jumagaliyev

©Copyright by Assylbek Sagitzhanuly Jumagaliyev.
All rights reserved.

Acknowledgements

I would like to express my sincerely gratitude to Yehia Elkhatib for his great patience, profound expertise, inspirational encouragement, generous professional support, and perceptive advice throughout my PhD. He has successfully guided me through the long and sometimes hard roads of PhD life. I have learnt so much from him and I feel lucky to have him as my advisor, mentor and friend.

Besides my supervisor, I would like to thank my examiners, Francois Taiani and Sumi Helal for their objective feedback, interesting suggestions, and constructive discussions we had during my viva.

My sincere thank also goes to my former supervisor, Jon Whittle, for his support and help at the beginning of my PhD, and for his contribution to my initial development as a researcher.

I thank my wonderful colleagues and fellow office mates who have been helpful and supportive. Specifically, a big thank to Abdessalam Elhabbash for reading my thesis and providing constructive feedback on some chapters. In addition, I am grateful to Zheng Wang, Rotsos Charalampos, Amit Chopra, Pete Sawyer, Gerald Kotonya, Jaejoon Lee, Ethem Bagci, Faiza Samreen, William Simm, Debbie Stubbs, Claire Oulton, Alex Powers, and Umar Armaya'U.

Special thanks go to my friends in Lancaster who have made my life enjoyable and unforgettable. I would like to thank Shahin, Afaf, Huyem, Aiman, Pang, Pear, Boyeun, Victoria, Peng, Hamed and Hunter.

Last but not least, I would like to thank my parents, Sholpan and Sagitzhan, for their encouragement and prayers. I would also like to thank my brothers, Askhat and Bakytzhan, and sisters, Shynar, Gaukhar, and Raushan, for their support and presence.

To my parents, Sholpan and Sagitzhan, along with brothers and sisters who are always supportive in all my endeavors.

Contents

Abstract	i
Declaration	iii
Acknowledgments	iv
Table of Contents	vi
List of Acronyms	x
List of Figures	xiii
List of Tables	xv
1 Introduction	1
1.1 Cloud Computing	2
1.1.1 Cloud Service Models	2
1.1.2 Multi-tenancy in SaaS	4
1.1.3 Domain-Specific Languages (DSLs)	5
1.2 Motivation	6
1.3 Challenges	7
1.4 Problem Statement	8
1.5 Research Aim & Objectives	10
1.6 Research Methodology	11
1.7 Contributions	12
1.8 Thesis Organization	14
1.9 Publications	15
2 Related Work	16
2.1 Overview	16
2.2 Manual Approaches	17
2.2.1 Schema-extension Techniques	18
2.2.2 Metadata-Driven Techniques	19

2.2.3	Multi-tenancy Enablement Layer	20
2.3	MDE Based Approaches	22
2.3.1	Unified Modeling Language Extensions	22
2.3.2	Domain-Specific Languages	23
2.4	SPL/ Based Approaches	24
2.4.1	Feature Modeling Based Techniques	24
2.4.2	Orthogonal Variability Modeling Language (OVM) Based Tech- niques	26
2.5	Hybrid & Other Modeling Approaches	27
2.6	Summary	29
3	Proposed Solution	31
3.1	Requirements	32
3.1.1	Concepts and Terminology Requirements	32
3.1.2	Meta-modeling Language Requirements	33
3.2	Methodology	34
3.3	Domain Analysis	35
3.3.1	Cloud Data Storage Types	35
3.3.2	Data Architecture Partitioning Schemes	38
3.3.3	Current Cloud Application Modeling Languages	38
3.3.4	Domain Analysis Output	39
3.4	Design	40
3.4.1	Language Exploitation versus Language Invention	40
3.4.2	Design Specification	41
3.4.3	CadaML Meta-model	41
3.5	Implementation	43
3.5.1	Graphical Domain-Specific Language (DSL) Implementation Frameworks and Tools	44
3.5.2	Comparing Graphical DSL Frameworks	46
3.5.3	Graphical Editor for CadaML	48
3.5.4	Validation Rules and Constraints	51
3.5.5	Code Generation	52
3.6	Reflection on Requirements	56
3.7	Summary	58
4	Application & Qualitative Evaluation of CadaML	59
4.1	Evolving from Single- to Multi-tenancy	60
4.1.1	Manual Evolution	60
4.1.2	Modeling using CadaML	61
4.2	Industrial Case Study: Background	64

4.2.1	Overview	65
4.2.2	Evolution Motivation & Challenges	66
4.3	Industrial Case Study: Implementation	67
4.3.1	Comparing the Data Partitioning Schemes	68
4.3.2	Evolving the Data Architecture	69
4.3.3	Evolving the Application Architecture	70
4.4	Qualitative Evaluation	72
4.4.1	Evaluation Methodology	72
4.4.2	Evaluation of the Feasibility	75
4.4.3	Evaluation of the Adequacy of Multi-tenancy Implementation	76
4.5	Discussion	77
4.5.1	Reflection on Challenges	77
4.5.2	Limitations	79
4.6	Summary	80
5	Experimental Evaluation	81
5.1	Evolving the Application	81
5.2	Experimental Design	82
5.2.1	Experiment Procedure	83
5.2.2	Participant Recruitment	84
5.2.3	Experimental Task	84
5.2.4	Exit Interview Questions	85
5.3	Participant Expertise	86
5.4	Participation Allocation	88
5.5	Modeling in CadaML	89
5.6	Evaluation of Productivity	91
5.7	Error Analysis	94
5.8	Exit Interview Results	95
5.8.1	Productivity	96
5.8.2	Quality of the Generated Code	97
5.8.3	Usability of CadaML	98
5.9	Discussion	99
5.9.1	Ease of Exploitation	99
5.9.2	Productivity of Developers	99
5.9.3	Code Generator	100
5.10	Threats to Validity	101
5.10.1	Construct Validity	101
5.10.2	Internal Validity	101
5.10.3	External Validity	101

5.10.4 Conclusion Validity	102
5.11 Summary	102
6 Discussions & Conclusions	104
6.1 Thesis Summary	104
6.2 Contributions	105
6.3 Reflection on Research Objectives	107
6.4 Limitations	108
6.5 Discussion	109
6.6 Future Work	113
Bibliography	113
A Comparing Data Storage Services of Cloud Providers	129
B Comparing Data Storage APIs of Cloud Providers	136
B.1 Relational Databases	136
B.1.1 Alibaba ApsaraDB	136
B.1.2 Amazon Relational Database Service (RDS)	137
B.1.3 Azure Structured Query Language (SQL) Databases	137
B.1.4 Creating connection and interacting with a database using Java Database Connectivity (JDBC) Application Program- ming Interface (API)	137
B.2 Non-relational Databases	138
B.2.1 Alibaba Table Store Service	138
B.2.2 Amazon Web Services (AWS) DynamoDB	139
B.2.3 Azure Table Storage	140
B.3 Blob Storage	140
B.3.1 Alibaba Object Storage Service (OSS)	140
B.3.2 Amazon Simple Storage Service (S3)	141
B.3.3 Azure Blob Storage	141
C Comparing Meta-modeling Environments	142
D Data Model for Alibaba Table Store	144
E Test Cases to Verify Multi-tenancy Implementation	145

List of Acronyms

ACM	Association for Computing Machinery
API	Application Programming Interface
APP	Application
AWS	Amazon Web Services
BPEL	Business Process Execution Language
CAML	Cloud Application Modeling Language
CadaML	Cloud Application Data Architecture Modeling Language
CDN	Content Delivery Network
CEUR	Central Europe
CRUD	Create, Read, Update, and Delete
CloudML	Cloud Modeling Language
DB	Database
DSL	Domain-Specific Language
DSPL	Dynamic Software Product Line
EC2	Elastic Compute Cloud
EF	Entity Framework
EGL	Epsilon Generation Language
EMF	Eclipse Modeling Framework
EOL	Eclipse Object Language
ER	Entity Relationship

CONTENTS

EVL	Epsilon Validation Language
FSTREC	Faculty of Science and Technology Research Ethics Committee
FaaS	Functions as a Service
GAE	Google App Engine
GCE	Google Compute Engine
GCP	Google Cloud Platform
GMF	Graphical Modeling Framework
GOPRR	Graph, Object, Property, Port, Relationship and Role
GWT	Google Web Toolkit
IaaS	Infrastructure as a Service
IBM	International Business Machines
IDE	Integrated Development Environment
IRB	Institutional Review Board
JDBC	Java Database Connectivity
JPA	Java Persistence Application Programming Interface
JPQL	Java Persistence Query Language
JSON	JavaScript Object Notation
LoC	Lines of Code
MDE	Model-Driven Engineering
MOCCA	Move to Clouds for Composite Applications
MODA-Clouds	Model-Driven Approach for Clouds
MSDK	Modeling SDK for Visual Studio
NoSQL	Not only Structured Query Language
OCL	Object Constraint Language
OS	Operating System

OSS	Object Storage Service
OVM	Orthogonal Variability Modeling Language
PaaS	Platform as a Service
RDS	Relational Database Service
S3	Simple Storage Service
SaaS	Software as a Service
SDK	Software Development Kit
SINTEF	Stiftelsen for Industriell og Teknisk Forskning
SLE	Service Line Engineering
SOA	Service-Oriented Architecture
SoaML	Service-Oriented Architecture Modeling Language
SPLE	Software Product Line Engineering
StratusML	Stratus Modeling Language
SQL	Structured Query Language
TOSCA	Topology and Orchestration Specification for Cloud Applications
UFPE	Universidade Federal de Pernambuco
UML	Unified Modeling Language
URL	Uniform Resource Locator
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
YAML	Yet Another Markup Language

List of Figures

1.1	Separation of responsibilities in different cloud service models [61].	2
1.2	Multi-tenancy patterns to deploy tenants and their resources.	4
1.3	Overview of the thesis design where chapters are mapped to research phases.	12
2.1	Overlap of approaches between SQL databases, Not only Structured Query Language (NoSQL) databases, and blob storage.	29
3.1	The development phases of CadaML following the methodology presented in [91].	34
3.2	Selected concepts and terminology for CadaML.	39
3.3	CadaML meta-model in UML class diagram.	42
3.4	The concrete syntax of CadaML implemented as the graphical editor with three parts: ① canvas ② palette, and ③ properties tab.	50
3.5	CadaML code generation process.	53
4.1	Comparing the implementation processes of the manual approach and that of CadaML.	60
4.2	The business process application architecture.	64
4.3	The Entity Relationship (ER) diagram of the business process application.	65
4.4	The evolution of the use case application from single- to multi-tenancy.	66
4.5	The evolved multi-tenant data architecture modeled in CadaML.	69
4.6	The first deployment scenario: the evolved architecture of the business process application that is deployed to services of AWS.	70
4.7	The second deployment scenario: the evolved architecture of the business process application where the application is deployed to services of AWS and the databases are deployed to a private cloud of the holding company.	71
5.1	Data storage in the business process application.	82
5.2	The flowchart of the experiment procedure.	83

5.3	The participants' self-reported programming experience in years. . . .	86
5.4	The number of participants based on self-reported experience in cloud application development and modeling.	86
5.5	The number of participants based on self-reported experience in exploiting cloud service providers, and their allocation.	87
5.6	The number of participants based on self-reported expertise level in years in cloud application development and modeling tools, and their allocation.	88
5.7	Object storage data architecture modeled in CadaML.	89
5.8	Non-relational data architecture modeled in CadaML.	90
5.9	Relational data architecture modeled in CadaML.	91
5.10	The distribution (median and interquartile range) of time taken by participants to finish the data layer implementation tasks using the 3 different datastore types. Using CadaML significantly reduces the development time. Note that only one participant attempted to accomplish any progress on SQL using manual implementation, hence the very narrow box on the far right.	93
5.11	Participant feedback regarding productivity: all agreed that CadaML helps reduce implementation time and difficulty, but not all agreed that it was sufficient on its own.	96
5.12	Participant feedback on reliability: generated code is of high readability and low frequency of errors; but with mixed perceptions about the ability of finding errors in the generated code.	97
5.13	Participant feedback on usability: CadaML is generally perceived to be intuitive and easy to use without restricting the developers' freedom of choice.	98

List of Tables

2.1	Total number of approaches and their application to different cloud storage types	17
3.1	Mapping cloud storage types to storage services of public cloud providers	36
3.2	Comparing concepts of non-relational databases to relational database	37
3.3	Comparing concepts of blob storage to filesystem	37
3.4	Comparing the DSL implementation frameworks	47
4.1	The distribution of test cases based on concerns under investigation per entity	77
4.2	The expanded test cases for <i>ProcessDefinition</i> and <i>Process</i> entities	78
5.1	The participants' self-reported expertise level in cloud APIs and storage services.	87
5.2	The participants' self-reported expertise level in programming with Java.	88
5.3	Time spent (in h:min:s) and completion rate (CR) by participants for each storage type through manual implementation	91
5.4	Time spent (in h:min:s) and completion rate (CR) by participants for each storage type using CadaML	92
5.5	The distribution of test cases planned for each storage type per participant.	94
5.6	The distribution of test cases that were planned and execute with their success and failure status.	94
A.1	Characteristics of Alibaba Object Storage Service (OSS)	129
A.2	Characteristics of Amazon S3	130
A.3	Characteristics of Azure Blob Storage	130
A.4	Characteristics of Google Cloud Storage	131
A.5	Characteristics of Alibaba Table Store	132
A.6	Characteristics of Amazon DynamoDB	133

A.7 Characteristics of Azure Table Storage	134
A.8 Characteristics of Google Cloud Datastore	135
E.1 The expanded test cases with their status for <i>Process Definition</i> entity	145
E.2 The expanded test cases with their status for <i>Task Definition</i> entity .	146
E.3 The expanded test cases with their status for <i>Process</i> entity	146
E.4 The expanded test cases with their status for <i>Task</i> entity	147

Chapter 1

Introduction

Cloud computing has become a major service provisioning paradigm as it provides powerful, reliable, and efficient platform to build and deploy cloud applications. It delivers on-demand, flexible and configurable computing resources over the Internet. Hence, instead of purchasing their own hardware and software infrastructure, cloud customers exploit computing resources over the network, and pay only for the resources they actually need and use. With the rapid growth of more efficient and affordable services offered by major cloud service providers such as Amazon, Google, and Microsoft, application providers are shifting to cloud environments.

Cloud computing resources can be provisioned as a virtual infrastructure, a platform, or predefined services. Most commonly application providers use computing resources delivered as a platform to develop and host their predefined services. These services range from email services to business oriented applications. Moreover, application providers tend to deploy multiple customers to a shared service with each customer enabled to configure or even customize the service based on its needs. Nevertheless, customers must be able to access and use a shared service as it is a dedicated one. Therefore, this thesis considers the challenges associated with sharing a single application across multiple customers. In particular, the thesis investigates the concerns related to sharing data, and propose a solution to address these concerns.

The remainder of this chapter is organized as follows. Section [1.1](#) defines the concept of cloud computing, explains different service models offered by cloud providers, discusses multi-tenancy patterns, and outlines how cloud computing could benefit from [DSLs](#). Section [1.2](#) highlights evolution motivations that trigger adoption of multi-tenancy, and Section [1.3](#) presents challenges associated with multi-tenancy. Sections [1.4](#) and [1.5](#) identifies the problem statement, and the research aim with objectives of the thesis, respectively. Meanwhile, Section [1.6](#) describes the followed research phases and methodology. Finally Section [1.7](#) emphasizes the contributions of the thesis, and Section [1.8](#) presents its structure.

1.1 Cloud Computing

In recent years, cloud computing has been widely exploited to deliver services over the Internet as it offers many advantages in comparison with existing traditional service providers. By definition, cloud computing is a paradigm that provides on-demand access to configurable computing resources to develop and deploy cloud applications [90]. The main factors that trigger application providers moving their applications to the cloud are the flexibility in resource provisioning and payment on the usage basis which lead to significant reduction in initial costs and transition from capital investments to operational expenses [90].

1.1.1 Cloud Service Models

As illustrated in Figure 1.1, there are three different service models [90] that allow outsourcing varying degrees of computing resources and hardware maintenance to a cloud provider. These models have their own benefits as well as differences in the amount of control over hardware and software resources they provide, and in the level of responsibility in managing them. The following describes each service model and its characteristics.

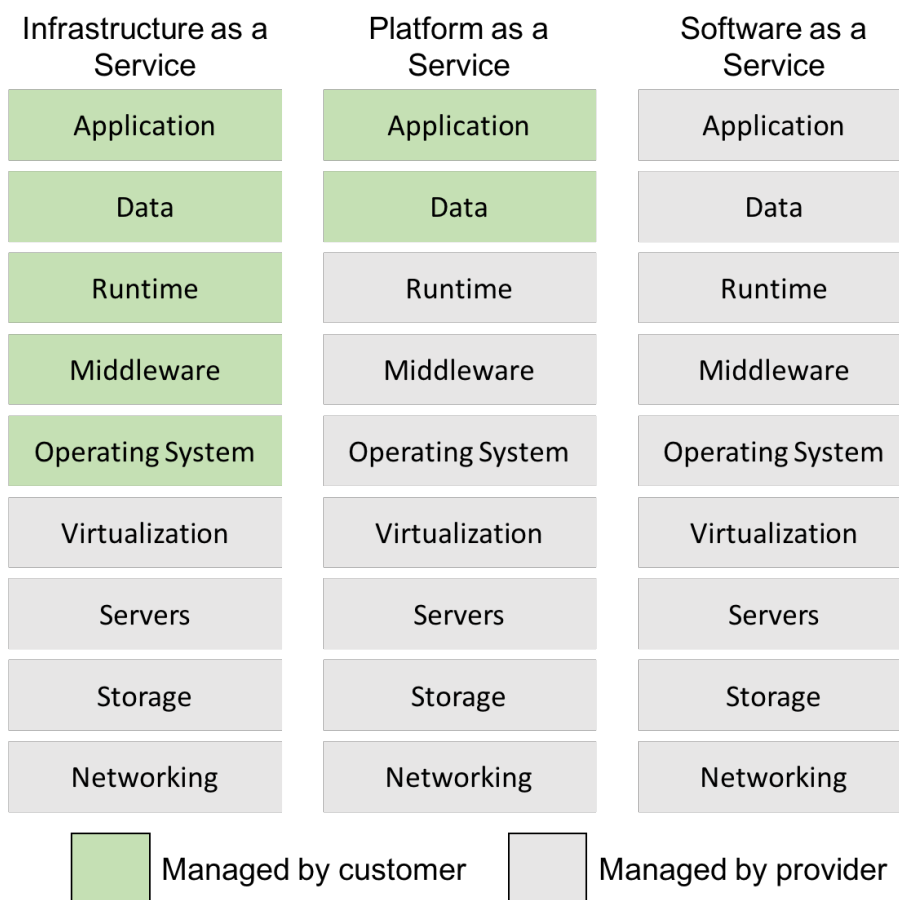


Figure 1.1: Separation of responsibilities in different cloud service models [61].

- **Infrastructure as a Service (IaaS)** model provides a pool of infrastructural resources such as servers, storage, and networking that are provisioned using virtualization technologies. In this model, cloud customers typically install a software stack which consists of an operating system, middleware and runtime environment to deploy their applications. Managing and maintaining the software stack is the responsibility of the customer, while managing the underlying infrastructure is handled by the cloud provider. The examples of IaaS provisioning services include AWS, Cisco Metacloud, DigitalOcean, Google Compute Engine (GCE), Linode, Microsoft Azure, and Rackspace.
- **Platform as a Service (PaaS)** model delivers operating systems and associated application services that facilitate development, testing, deployment and maintenance of applications without the need for investment in infrastructure and software environment. PaaS provides an environment to run cloud applications, services for data storage, and a number of additional services to fully support deployment of cloud applications. In order to benefit from the capabilities of the cloud platform, it is necessary to significantly evolve existing applications, or even implement a new one. As the evolution we refer to the modification of the application to adapt for the cloud environment. Most commonly exploited PaaS services comprise Apache Stratos, AWS Elastic Beanstalk, Force.com, Google App Engine (GAE), Heroku, OpenShift and Windows Azure.
- **Software as a Service (SaaS)** model enables customers to access applications running on a cloud infrastructure (*e.g.*, Cisco WebEx, Concur, GoToMeeting, Dropbox, and Salesforce). In this model, the application and associated data are hosted in the cloud and customers subscribe to the application over the Internet. This, in turn, eliminates the need to install and maintain the application, and it also removes the necessity to manage and control the underlying cloud infrastructure. Nevertheless, customers have no control over individual application capabilities, they are only enabled with limited user-specific application configuration settings.

Among these service models, IaaS has been mostly investigated to deploy and run cloud applications (*e.g.*, [100,121]), though maintaining a virtual infrastructure with supporting software stack incurs additional costs for application providers, and requires system administrators with sufficient skills [131]. In contrast, PaaS delivers a platform to develop and deploy SaaS applications while managing performance, availability, scalability and other infrastructure related concerns of computing resources. The combination of a full software stack and a managed platform significantly reduces the maintenance effort and upfront infrastructure investment for

application providers [115], and makes PaaS the most commonly used service model to develop and run SaaS applications [131]. Moreover, the demand for PaaS services is rapidly growing [33], and cloud service providers encourage building SaaS applications using PaaS services by offering a complete set of tools and design guidelines [17, 58, 105, 125]. Based on this, we investigate how SaaS applications can be implemented using services of the PaaS provisioning model and deployed on top of them.

1.1.2 Multi-tenancy in SaaS

In traditional application provisioning models each customer is deployed to its own independent application, database and software stack that are customized based on customer's requirements. However, this model has many drawbacks, such as inefficient resource utilization, limited scalability, and high maintenance effort and deployment costs [18, 19, 126]. In order to address these drawbacks, application providers have been adopting *multi-tenancy* pattern, where multiple *tenants* share an application with its supporting infrastructure while being able to configure the application for their needs [68]. In this context, a *tenant* is a group of users that belongs to an organization who has access with specific privileges to an application [74].

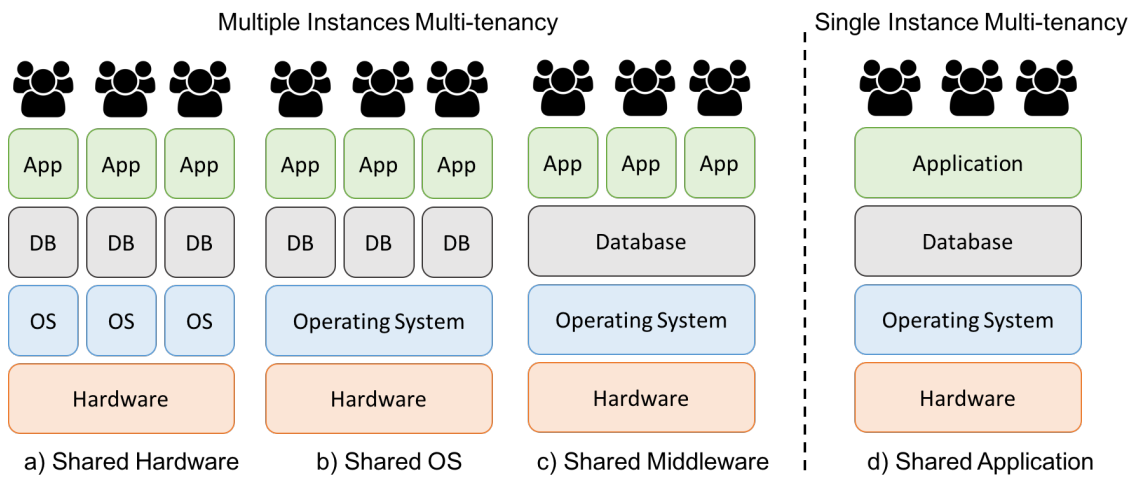


Figure 1.2: Multi-tenancy patterns to deploy tenants and their resources.

As illustrated in Figure 1.2, there are generally two resource sharing approaches in multi-tenancy [26]: *multiple instances multi-tenancy* and *single instance multi-tenancy*. In *multiple instances multi-tenancy*, each tenant has a dedicated application instance on a shared hardware, operating system, or middleware. Whereas in *single instance multi-tenancy*, tenants are served by a single application instance that runs on shared hardware and software infrastructure while the application distinguishes the requests and data of each tenant. However, in the latter approach

tenants must be able to configure and extend the application to their needs as it runs in a dedicated environment.

Multi-tenancy approaches offer different benefits and drawbacks, and choosing one approach over another depends on application requirements and architectural considerations. The *multiple instances* pattern (*i.e.*, *Shared Hardware* and *Shared Operating System (OS)*) provides more security compared to the *single instance* pattern as each tenant's application and data are completely separate from each other. The chance of accidental access other tenant's data is essentially eliminated. This type of isolation also prevents performance degradation which leads to higher reliability and provides tenants with full control over the environment. Nevertheless, cloud providers have limits on the number of application and database instances that can be created. Moreover, the *multiple instances* approach is usually costly and labor-intensive as it requires maintenance of multiple application and database instances, and it is inefficient in terms of resource utilization.

In contrast, the *single instance* approach addresses the limitations of the *multiple instances* technique. Primarily, tenants share the same application and database instances that dynamically scale on demand, hence, this approach can support a much larger number of tenants [17,26,29,105], and maximizes resource usage. Furthermore, application and database maintenance becomes easier since a single instance of each has to be maintained. All these factors significantly reduce overall operational and energy costs of resource provisioning and software maintenance [1,21,37,52,123], and encourage application providers to adopt this pattern for their existing applications. Throughout this thesis, we focus on the *single instance multi-tenancy* pattern, explore the key requirements, and consider the challenges introduced by this pattern.

1.1.3 Domain-Specific Languages (DSLs)

Multi-tenancy in cloud applications introduces a set of concerns that could be encapsulated into DSLs, specifically, to generate and/or maintain cloud application implementations [1]. A DSL is a concise, simple and expressive language that is focused on addressing problems of a particular domain [43]. DSLs can raise the level of abstraction by enabling specification of a model of an application directly using domain concepts, and improve productivity of developers during the application development by generating an application from the high-level model [85].

In software engineering, DSLs typically have a graphical interface, and provide notations and concepts to model a solution of a particular domain by domain experts [77]. DSLs can be implemented for interpretation or code generation [43]. Interpretation executes DSL models by simulating a runtime environment, while code generation produces high level language source code from a model. For example,

the Entity Framework (EF) designer from Microsoft¹ is used for visual modeling of persistent classes, generating a database schema from a model, and producing code in C# to interact with SQL Server databases.

In this thesis, we aim to implement a DSL to address multi-tenancy concerns in cloud applications. A DSL for multi-tenancy would possibly comprise concepts to represent tenants, their configurations, cloud providers, tenant database, tenant schema, and deployment specifications.

1.2 Motivation

Software evolution is an inevitable process in any software system that requires modification of software to adapt to changes [120]. The reasons that call for evolution to multi-tenancy include changes in business requirements, changes in hardware and software platforms, improving resource provisioning and application maintenance, or changes in the application delivery model. In this section, we describe some of the evolution motivations that typically lead to introducing multi-tenancy in existing applications.

Most importantly, provisioning and deploying dedicated application and database instances for each tenant is time consuming and labor-intensive [18]. Providing an application and a database for new tenants usually requires preparation of the deployment infrastructure, configuration of networks, installation of all necessary software, and ensuring proper functioning of the infrastructure. Nevertheless, tenant demands need to be met more quickly and efficiently than the traditional deployment procedures allow.

Another issue with traditional application provisioning is that much of the infrastructure is used inefficiently. As it is difficult to predict an application workload, application and database instances are deployed on servers that are configured to provide enough resources during high workload. Thus, the majority of servers are underutilized during normal workload [18, 19, 21, 37, 52, 73, 123]. This inefficiency could be optimized using better application deployment solutions.

A final issue is that maintenance and support of several web application and database instances require additional effort. For application providers, it is important to fix tenants' problems before they become expensive to resolve. Moreover, providing on-time and sufficient support is crucial to keep tenants satisfied with the application. The nature of cloud environments along with multi-tenancy patterns could address these issues by providing lower costs through economies of scale and resource sharing, automated management of the application resources, centralized

¹<https://docs.microsoft.com/en-us/ef/ef6/>

control over the application with its supporting infrastructure, and dynamic scalability.

1.3 Challenges

Despite its benefits, introducing multi-tenancy affects the overall application structure which requires application providers to address multiple challenges and find a balance between several architectural trade-offs [74]. The following multi-tenancy concerns along with design factors influence the application design, and they are listed in the order of importance.

- *Transparent tenant isolation:* When multiple tenants share the same application instance with its supporting infrastructure, isolation among different tenants is the highest priority. In a shared environment, each tenant must be able to access and use the application as if it is a dedicated one. Nevertheless, tenants must be able to view and edit only their own data. Hence, tenant isolation must be carefully considered in all layers of the application architecture from both functional and non-functional perspectives.
- *Configurability and extensibility:* In the traditional application provisioning model, each tenant is able to be served by customized application and database instances based on its requirements. In multi-tenancy, such customization is not applicable since customization of one tenant will impact the whole application. As a result, multi-tenant applications must provide a mean for tenants to configure and extend the application and database at run-time to cater for their needs without affecting other tenants sharing the application.
- *Scalability:* In multi-tenancy the application and database workload varies as several customers of multiple tenants interact with a single application and database. To avoid disruption and enable smooth running of the services, application providers must ensure scalability of the solution by dynamically creating and releasing application resources to match the performance requirements [59].
- *Ease of development and maintenance:* The transformation of an existing application into a multi-tenant SaaS service requires re-engineering the application architecture to support multi-tenancy concerns. This transformation process is seen as a major barrier by application providers [17, 26, 32, 127], thus, they are concerned that development and maintenance of multi-tenant applications will require a vast amount of effort.

These challenges influence all layers of an application architecture. This is especially true in the case of the data layer which is often the prime candidate for multi-tenancy [52], as other layers are typically stateless in cloud applications [11, 38]. Stateless layers do not store any tenant data since they are provided with each request [40]. Therefore, throughout this thesis, we focus on the data layer, analyze how these challenges influence the data layer of multi-tenant cloud applications that usually consists of different cloud data storage types, and investigate how these challenges can be addressed using modeling techniques.

1.4 Problem Statement

When building a multi-tenant application, one of the highest priorities for developers is to design a configurable and scalable data architecture that maximizes resource sharing across tenants, and one that is also efficient and cost-effective to implement and maintain [28]. However, cloud applications usually have a variety of data storage requirements and are often best served by a combination of multiple data storage options [110, 116, 125]. These storage options differ in storing and organizing data. Moreover, each storage option has its own partitioning and extensibility approaches to support multi-tenancy.

Cloud providers offer *relational databases*, *non-relational databases* and *blob/object storage* at the **PaaS** provisioning level for storing application data. *Relational databases* are appropriate for structured data with a well-defined schema where data is organized in tables, rows and columns, and a *primary key* identifies each row in a table. Relationships among tables, columns and other database elements are strongly defined in the data model. On the contrary, *non-relational databases* (also called **NoSQL**) are suitable for flexible data schemas and they support key-value store models. Data is also stored in tables, where a *partition key* determines the partition in which data will be stored, and a *row key* identifies data within each partition. In turn, *Blob/Object storage* is ideal for completely unstructured data such as documents, media files, or binary data. Data is stored in *buckets* as a blob, where a *key* uniquely identifies each blob (i.e., object or item) within a *bucket*.

In order to ensure isolation of tenant data, and scalability of the solution, a partitioning scheme is crucial when sharing application code and data across all tenants. Typically, each cloud storage type has its own partitioning techniques. For example, relational databases can be partitioned using one of the following ways. (i) *Separate databases*: each tenant is served by a dedicated database. (ii) *Shared database, separate tables*: all tenants are hosted by a single database with separate tables per tenant. (iii) *Shared database, shared tables*: all tenants share tables in a single database, with a tenant identifier is used to associate their records in each

table. Similarly, non-relational databases can be partitioned by either deploying *separate tables* per tenant, or *sharing tables* across all tenants. Whereas for blob storage all blobs belonging to a specific tenant can be stored in a dedicated bucket, or all tenant data can be stored in shared buckets.

Furthermore, cloud providers offer similar data storage services, though they provide different libraries and [APIs](#) to implement the data layer. For example, Alibaba Cloud requires manually creating an instance of a non-relational database, meanwhile for other cloud providers a database instance can be created using their [APIs](#). In addition, cloud providers use their own annotations for mapping classes and properties to tables and attributes in actual non-relational databases. As another example, a deployment region for blob storage can be assigned at run-time for Alibaba and [AWS](#), but for Azure and Google this must be specified when creating a storage credentials and creating a project in [GAE](#), respectively. More detailed characteristics of different cloud data storage types and comparison of [APIs](#) provided by major cloud providers are presented in Appendices [A](#) and [B](#).

Given these varying cloud data storage solutions, partitioning schemes and [APIs](#), manually implementing a multi-tenant data architecture can be highly time-consuming and error-prone [\[23, 45\]](#), especially for architectures utilizing more than one storage type. Recent research has aimed to generate multi-tenant cloud applications from high-level models in order to hide cloud-specific implementation details, *e.g.*, [\[20, 41, 88, 94, 97, 103\]](#). Several modeling languages have been proposed in this direction. As an example, an eXtensible Markup Language ([XML](#))-based modeling language is provided by Topology and Orchestration Specification for Cloud Applications ([TOSCA](#)) to define application components and their relationships [\[8\]](#). Another example, Cloud Application Modeling Language ([CAML](#)) is proposed to express cloud-based deployments directly in an application model [\[15\]](#). However, these modeling languages tend to focus less on managing multi-tenancy in the data layer, and instead focus on other aspects such as enabling configurable application functionality, capturing different functional and quality-of-service tenant requirements, and modeling variation of business logic implementation and deployment alternatives. Although there is a few approaches that tackle multi-tenancy at the data layer, none of them allow to capture multi-tenancy patterns in the data layer, produce the data access code from the model, or consider different cloud storage types with their partitioning peculiarities.

In conclusion, we come up with the following research areas that need improvements:

- Current modeling approaches should consider and capture conceptual differences of available cloud data storage solutions.

- Current modeling approaches should capture varying data architecture partitioning and extensibility alternatives of different cloud data storage types.
- Current modeling approaches should provide a way to build a multi-tenant data architecture that abstracts from the implementation differences of different cloud data storage types.
- Current modeling approaches should expedite data layer implementation by reducing the development overhead and minimizing errors in the application code.

1.5 Research Aim & Objectives

This thesis aims to *facilitate implementation of multi-tenancy at the data layer of cloud applications*. Specifically, we investigate how applying **MDE** techniques could benefit to raise the level of abstraction when building multi-tenant data architectures, expedite the development process, and increase the productivity of developers. In **MDE**, models are the key artifacts throughout the engineering lifecycle and they represent abstract description of an application from a certain viewpoint. As the main contribution of the thesis, we propose a cloud data architecture modeling language (**CadaML**) to design a multi-tenant data architecture and generate source code from the model that is suitable for different cloud storage types. We set the following objectives to achieve our aim.

- R01:** *Analyze existing approaches that consider multi-tenancy challenges at the data layer of cloud applications.* This will help to assess the previous and current research in multi-tenancy, gain insights into the practical challenges of implementing multi-tenancy at the data layer, and explore methods for data collection and evaluation of modeling languages.
- R02:** *Provide a way to describe a multi-tenant cloud data architecture at an abstract level.* This is based on the realization that cloud providers offer their own libraries and **APIs** to implement the data layer. Hence, a data architecture designed using the modeling language should hide the implementation details by only representing data layer related components while also capturing data partitioning and extensibility to address the multi-tenancy challenges. From the data architecture, the modeling language should be able to produce the data access layer code that can be deployed to services of different cloud providers.
- R03:** *Reduce the development effort during the implementation of a multi-tenant cloud data architecture.* Since manual implementation of multi-tenancy is

time-consuming and error-prone, the modeling language should mitigate and reduce the development effort by (semi-)automating the data layer implementation process.

R04: *Improve reliability of the application code (specifically at the data layer).* For the same reason as in R03, the modeling language should reduce the number of errors in the application code by means of model validation and automated code generation.

R05: *Provide developers with a reasonable level of usability.* By proposing a modeling language, we aim to increase the productivity of developers, reduce the development effort and improve the quality of the data access layer code. However, the modeling language needs to be intuitive to exploit, suitable to implement multi-tenancy, and it should not require much effort or time to learn it.

1.6 Research Methodology

We divide the work in this thesis into three main research phases in order to systematically achieve the above research objectives, and aim.

- **Phase 1: Literature survey** of existing manual approaches along with model-based and model-driven modeling languages, tools and frameworks in order to define the overlap and identify the gap in the current research (Supportive of R01).
- **Phase 2: Development** of a modeling language which simplifies and expedites the implementation of multi-tenant data architectures of cloud applications (Supportive of R02 and R05).
- **Phase 3: Evaluation** of the modeling language in terms of its applicability to evolve an industrial single-tenant web application into a multi-tenant **SaaS**, productivity of real developers with varying abilities, reliability of the generated code, and usability of the language (Supportive of R03, R04, and R05).

As we propose a new modeling language that requires quantitative evaluation to investigate interaction of developers with the language, this thesis adopts a combination of constructive and empirical research, and it is designed based on the research phases as shown in Figure [1.3](#). We explore existing academic and industrial approaches that focus on addressing multi-tenancy concerns at the data layer by means of manual and modeling techniques (*Phase 1: Literature Review*). This is

followed by the implementation of the modeling language (*Phase 2: Development*), and subsequent evaluation (*Phase 3: Evaluation*).

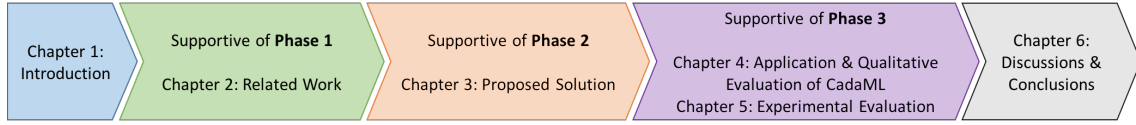


Figure 1.3: Overview of the thesis design where chapters are mapped to research phases.

Phase 1 reviews relevant approaches that tackle multi-tenancy in cloud applications to (i) outline the importance of the research scope being considered; (ii) identify weaknesses and limitations of the current research; and (iii) analyze different methodological approaches that have been applied for investigation of multi-tenancy concerns, data collection and evaluation of modeling languages.

Phase 2 involves identifying requirements for concepts and terminology that must be included in the modeling language, and meta-modeling language for the implementation of the language. It also involves defining the methodology for the development of the language which includes domain analysis, design and implementation. The requirements and methodology are based on the findings of *Phase 1* and analysis of the literature to develop **DSLs**.

Phase 3 evaluates the modeling language in terms of its *application* in a real-world application, *effectiveness* at facilitating *productivity* of developers and *reliability* of the data layer code, and *usability* of the language. The evaluation is carried out in two stages. The first stage inspects the applicability of the modeling language to design and implement multi-tenancy at the data layer using qualitative evaluation. This includes conducting a case study to evolve a data layer of an industrial web application to adopt multi-tenancy, and demonstration of the correctness of multi-tenancy implementation through a combination of manual review and JUnit testing. The second stage evaluates the benefits and drawbacks of the language through a controlled experiment with task analysis. This approach involves observing interaction of real developers with the modeling language, analyzing their performance and productivity, and collecting their experience through a structured questionnaire, an exit interview and open-ended questions.

1.7 Contributions

In this thesis, we propose CadaML that provides concepts and notations to design and implement the data layer of multi-tenant cloud applications as a combination of different cloud data storage solutions. Major contributions are grouped according to the three research phases and mapped to the research objectives they reflect.

Literature review

- C1. A literature search that covers current manual and modeling approaches, and patterns for developing multi-tenant cloud data architectures (R01).
- C2. Comparison of existing approaches to outline the overlap and gap in the state of the art (R01).

Development

- C3. A set of requirements for concepts and terminology for the modeling language formulated based on design methodology on **DSL** development (R02).
- C4. A set of requirements for a meta-modeling language to implement the modeling language (R02).
- C5. A domain model that includes common vocabulary of available cloud data storage solutions at the **PaaS** service level offered by four major public cloud providers and their partitioning alternatives (R02).
- C6. A meta-model of the language, its design and implementation that support the requirements specified in C3 and C4 and integrates them into a graphical modeling language to build multi-tenant data architectures (R02).
- C7. A set of deterministic validation rules to ensure consistency of a model created using the modeling language, and reliability of the model-to-code transformation (R03 and R04).
- C8. A code generation tool that uses a validated model to synthesize a data layer implementation with multi-tenancy management logic that corresponds to the specific data storage types and policies selected by the developer (R03 and R04).

Evaluation

- C9. A qualitative evaluation of the modeling language through a case study of an industrial web application. The aim of the evaluation is to assess the feasibility of the modeling language to design and implement the data layer of multi-tenant cloud applications (R03 and R05).
- C10. An empirical evaluation of the modeling language through an experimental user study. The case study of an industrial application is reused to further evolve the data layer as a combination of different cloud storage solutions to reduce the costs. We specifically observe the productivity of developers of varying abilities, the reliability of the generated code, and usability of the modeling language (R03, R04 and R05).

1.8 Thesis Organization

The thesis consists of six chapters. The following gives an overview of chapters and associates each chapter with resulted contributions.

- **Chapter 1 - Introduction** outlines the work of the thesis.
- **Chapter 2 - Related Work**
Chapter 2 presents and discusses current academic and industrial research works that are geared towards addressing multi-tenancy challenges at the data layer. The related researches are covered by dividing them under four categories: *manual methods*, *MDE based techniques*, *SPLE based techniques*, and *other modeling approaches*. This chapter also analyzes the overlap and identifies the gap in the state of the art (C1 and C2).
- **Chapter 3 - Proposed Solution**
Chapter 3 proposes CadaML to fill the gap in the current research. The chapter starts with specifying the requirements for the concepts and meta-modeling language, and it introduces the methodology for the development of CadaML. This is followed by domain analysis, design and implementation of the modeling language. Finally, the chapter validates the language implementation by reflecting on the specified requirements (C3–C8).
- **Chapter 4 - Application & Qualitative Evaluation of CadaML**
Chapter 4 describes the application of CadaML and presents a qualitative evaluation of the language through a case study. The application demonstrates the transformation of a data layer from single- to a multi-tenancy using the modeling environment of the language, and highlights the capability of the language to design different multi-tenancy options at the abstract level, and to produce the corresponding data access layer code from the model. The qualitative evaluation shows the evolution of an industrial business process analyzing web application from single-tenant on-premises to a multi-tenant service deployed to a public cloud. This chapter also discusses reflection on evolution challenges, and comments on the limitations of the performed case study (C9).
- **Chapter 5 - Experimental Evaluation**
Chapter 5 further evaluates CadaML through an experimental user study where real developers are asked to evolve the data layer of the use case application to deploy it on different cloud data storage. The aim of the evaluation is to assess productivity of developers against the manual method, reliability of the generated code, and usability of the modeling language. Primarily, the chapter defines the experimental design and evaluation methods that are exploited to

conduct the experiment. The chapter continues by describing the expertise level of participants in programming, cloud application development, cloud data layer development, modeling tools and their allocation. Further, the modeling process of the data layer using CadaML is explained. The chapter, then, illustrates the evaluation results, discusses the limitations of the experiment, and analyzes threats to validity (C10).

- **Chapter 6 - Discussions & Conclusions**

Chapter 6 summarizes the chapters with overall discussions and conclusions of the thesis.

1.9 Publications

The following presents my major contributions that are published in peer reviewed workshops and conferences. The publications are listed in chronological order and mapped to chapters of this thesis they most relate to.

1. Assylbek Jumagaliyev and Jon Whittle. Model-Driven Engineering for Multi-tenant SaaS Application Development. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructure & Platforms co-organized with Eurosys Conference 2016*, pages 1–2, 18-21 April 2016 [65] (Chapter 2)
2. Assylbek Jumagaliyev, Jon Whittle, and Yehia Elkhatib. Evolving Multi-tenant SaaS Cloud Applications Using Model-driven Engineering, In *Proceedings of 10th International Workshop on Models and Evolution co-organized with MODELS Conference 2016*, pages 60-64, 2-7 October 2016 [66] (Chapter 3)
3. Assylbek Jumagaliyev, Jon Whittle, and Yehia Elkhatib. Using DSML for Handling Multi-tenant Evolution in Cloud Applications, In *Proceedings of IEEE International Conference on Cloud Computing Technology and Science (Cloud-Com)*, pages 272-279, 11-14 December 2017 [67] (Chapters 3 and 4)
4. Assylbek Jumagaliyev and Yehia Elkhatib. CadaML: A Modeling Language for Multi-Tenant Cloud Application Data Architectures, accepted to *IEEE Cloud Conference 2019* [64] (Chapters 3)
5. Assylbek Jumagaliyev and Yehia Elkhatib. A Modelling Language to Support the Evolution of Multi-Tenant Cloud Data Architectures, accepted to *IEEE / ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS) 2019* (Chapter 3, 4, and 5)

Chapter 2

Related Work

This chapter provides detailed discussion of the previous and current approaches that consider multi-tenancy at the data layer. First, section 2.1 gives a high level overview of the approaches and their application to different cloud storage solutions. Then, manual approaches are described in section 2.2, followed by modeling techniques based on Model-Driven Engineering (MDE) and Software Product Line Engineering (SPLE) in sections 2.3 and 2.4, respectively. Section 2.5 presents modeling techniques that combine MDE and SPLE, and those that use other modeling approaches. Finally, section 2.6 concludes the chapter by analyzing the overlap and identifying the gap in current approaches.

2.1 Overview

Several approaches have been proposed so far to address multi-tenancy challenges in cloud applications. These approaches have very similar goals but tackle multi-tenancy concerns in different ways. They can be classified as **manual** implementations and modeling techniques based on MDE, SPLE, and *other modeling approaches*.

Manual implementations are aimed at partitioning and isolating tenant data through manual development. In turn, modeling techniques are geared towards expressing customization and configuration alternatives in an application structure, and describing deployment of application components on cloud services. Generally, MDE based techniques have been implemented as extensions for general purpose modeling languages or standalone DSLs. Meanwhile, SPLE based techniques have exploited well known modeling techniques such as feature modeling and Orthogonal Variability Modeling (OVM).

For the literature survey, the electronic databases (namely Association for Com-

Table 2.1: Total number of approaches and their application to different cloud storage types

	Total number of related work	SQL	NoSQL	Blob
Manual approaches	13	12	1	1
MDE based modeling languages	13	13	13	11
SPLE based modeling techniques	9	6	3	3
Other modeling approaches	8	8	3	3
TOTAL	43	39	20	18

puting Machinery (ACM) Digital Library, SpringerLink ¹, IEEE Xplore ², Central Europe (CEUR) Workshop Proceedings ³, Google Scholar ⁴, ScienceDirect ⁵, Scopus ⁶, and ResearchGate ⁷) have been used as primary searching sources. The search for publications was filtered from 2006 onward as cloud providers such as AWS, Google Cloud Platform (GCP) and Alibaba Cloud started officially launching their services from this year. The following terms were included in the search queries: ‘multi-tenancy’, ‘data architecture’, ‘data layer’, ‘modeling’, ‘domain-specific language’, ‘model-driven engineering’. The search was also limited to studies that are indexed by the keyword ‘cloud computing’. The publications from the search results were further filtered for relevance by reading their abstract and conclusion sections.

As presented in Table 2.1, in total 43 academic and industrial studies have been identified as relevant during the literature survey. Among these studies, manual approaches and MDE based modeling languages have 13 papers related to each, 9 papers refer to SPLE based modeling techniques, and the remaining 8 papers are based on other modeling approaches. Table 2.1 also emphasizes the application of approaches to different cloud storage types. Specifically, a total of 39 papers discuss multi-tenancy in relational databases, 20 cover non-relational databases, and 18 capture blob storage.

2.2 Manual Approaches

Enabling multi-tenancy in relational databases has been commonly achieved through manual implementation. In some approaches, multi-tenancy is implemented by means of schema-extension techniques and the use of metadata. Other approaches propose an additional layer to ensure data isolation of tenants and customizability

¹<http://www.springerlink.com>

²<http://ieeexplore.ieee.org>

³<http://ceur-ws.org/>

⁴<https://scholar.google.co.uk/>

⁵<http://www.sciencedirect.com>

⁶<http://www.scopus.com>

⁷<https://www.researchgate.net>

of the data layer.

2.2.1 Schema-extension Techniques

Schema-extension techniques have been introduced to extend the base schema in a database in order to support customization of the data layer. The base schema contains a set of core tables that is shared across all tenants, and extension schemas define extensions and specify the purpose of extension elements.

Three generic structures have been described in [9] to achieve customization of a single shared relational database: (i) *Universal table*, (ii) *Pivot table*, and (iii) *Chunk table*. A *universal table* stores a large number of generic *data* columns, a *tenant* column, and a *table* column. The *data* columns have a flexible type (*i.e.*, VARCHAR) that can be converted into other types during the runtime. The *tenant* column identifies a tenant that owns columns, while the *table* column defines a logical table. Thus, tenants can extend the same table in different ways. The same approach is also adopted by Salesforce.com [132]. However, one significant drawback of this approach is that the generic table has to store many *null* values that may lead to misinterpretation of these values, and inefficient usage of the database storage space [136].

A *pivot table* eliminates the need to handle many *null* values. In a *pivot table*, each field of the logical tables is mapped into a physical row. In addition to *tenant* and *table* columns, a *pivot table* includes *row* and *col* columns to specify a source field that a row represents and a value stored in that field. Nevertheless, a *pivot table* has more columns to store meta-data than actual data that also requires additional storage space.

A *chunk table* is an alternative to *pivot tables* but the *col* column is replaced by a *chunk* column. In a *chunk* table, a data is partitioned into *chunks*, and a *chunk* identifier is assigned to each *chunk*. In all three cases, query transformation is required to produce appropriate queries that operate over the generic structures.

Similarly, a database design technique proposes using *common tenant tables*, *extension tables*, and *virtual extension tables* in a single shared relational database [135]. *Common tenant tables* are physical tables that are shared across all tenants. In contrast, *extension tables* store meta-data to extend an existing physical multi-tenant database schema. Further, *virtual extension tables* are created at run-time from *extension tables*. It is worth to note that tenants can also create their own virtual database schema to meet their requirements that does not extend the existing schema. A combination of these three tables can be used to enable multi-tenancy at certain levels: (i) a tenant identifier can be used in *common tenant tables* to differentiate tenants rows, (ii) *common tenant tables* with *virtual extension tables*

can be combined to extend the physical database schema, or (iii) *virtual extension tables* can be used to create a tenant-specific virtual database schema.

In another approach [112], a tenant-aware schema inheritance concept has been presented. The schema inheritance defines *shared schema*, *virtual schema* and *tenant schema* types. A *shared schema* is static and all tenants share this schema. In turn, a *virtual schema* defines a core application schema that can be extended by each tenant through inheritance. As a result of extension, a *tenant schema* is driven from the *virtual schema*. Furthermore, a *tenant context* is introduced to associate each *tenant schema* with a corresponding tenant.

2.2.2 Metadata-Driven Techniques

In metadata-driven techniques, metadata is used to store configurations and extensions that are specific to each tenant. At run-time, an application retrieves these metadata and applies required configurations on a per-tenant basis.

In [87], an additional database is created to persist metadata that describes tenant-specific information such as tenant name, tenant domain, billing and tenant identifier. In this approach, a single shared database with shared schema has been adopted to support multi-tenancy at the data layer. Furthermore, a *tenant resolver* and a *data partitioner* are implemented. The *tenant resolver* is implemented as a filter that intercepts web requests, analyzes the requests and extracts the tenant identity from the requests. Next, the *tenant resolver* determines the tenant from the extracted tenant identity by querying the database, and stores the tenant on the user session. In turn, the *data partitioner* is implemented using object mapper frameworks, and it defines parameterized default scopes for database operations (*i.e.*, Create, Read, Update, and Delete (CRUD) operations). As a parameter the tenant identifier is set from the user session.

On the contrary, an existing application's data layer has been evolved into a single shared database with separate schemas per tenant [134]. This partitioning model is chosen as it requires less manual modifications and lowers operational costs compared to other data partitioning models for relational databases. As in [87], an additional database instance is deployed that is shared among all tenants. This database stores tenant information and configurations such as a tenant-specific schema address. When a tenant sends a request, the application connects to the additional database to retrieve the corresponding value of the schema address for that tenant and pushes it to a tenant's session.

Interestingly, both separate databases and a shared database with shared schema partitioning models for relational databases are supported by a metadata driven architecture called Cloudio [63]. The architecture also requires an additional database

(*i.e.*, *Multi-tenant Database Index*) that stores the data of which database corresponds to each tenant. Furthermore, a tenant identifier column is added to every existing table in the actual database to associate records in tables with an appropriate tenant. The data access layer also requires modifications to read configuration data from the *Multi-tenant Database Index* in order to route tenant requests to an associated database or records within. Finally, re-implementation of queries is needed to inject the tenant identifier.

Partitioning and tenant isolation in all three different cloud storage types have been discussed in [17]. Specifically, relational databases can be partitioned either by deploying: (i) a dedicated database per tenant with metadata associating each database with a corresponding tenant; (ii) a shared database with per tenant tables where table name contains tenant identifier, or each tenant is assigned with a different schema; or (iii) single shared database with shared schema where tenant identifier is added to rows in all tables. In non-relational databases, each tenant may have its own set of tables with tenant identifier included in table name, or tenants share all tables with tenant identifier included in the partition key. Meanwhile, blob storage can be partitioned either by creating a bucket per tenant where a bucket name comprises the tenant identifier, or all tenants sharing the same buckets with the tenant identifier included in the blob name.

2.2.3 Multi-tenancy Enablement Layer

A multi-tenancy enablement layer [24] includes a *tenant context* and an *interceptor*. A *tenant context* is an object that stores tenant information (*e.g.*, tenant identifier), whilst an *interceptor* is a web filter that captures tenant information from the *tenant context*. The *tenant context* is also used to propagate tenant information to perform data operations on a single shared database.

Another multi-tenancy enablement layer [26] has been proposed to provide on-demand customization and ensure isolation in security, performance, availability, and administration aspects in shared application and database by all tenants. Isolation of data is achieved by inserting the tenant-oriented filter into requests that require access to database. Hence, `SQL` queries are modified with a sub-statement (*i.e.*, ‘WHERE *tenant_id* IS xxx’) to retrieve tenant-specific data.

A data middleware has been described in [86] to enable tenant isolation and customization of a single shared database with shared schema. The main components of the middleware are the *abstract data model*, *query interceptor and parser*, `SQL request router`, and *data cloud node*. The *abstract data model* consists of the same set of tables and views from the actual physical database and it provides logical data isolation for tenants. When a tenant requests data from an application, the *query*

interceptor and parser intercepts queries, and formulates new queries with tenant information to interact with the *abstract data model*. Then, a **SQL** request router sends **SQL** request to different *data cloud nodes* to retrieve appropriate data from the database pool.

An additional data access layer [32] has been inserted between the business logic and the application's database pool to enable multi-tenancy at the data layer. Ideally, the additional layer is responsible for creation of new tenants in the database, query adaptation to isolate tenants' requests and load balancing to meet the changing application workload. A property is added to the data model to identify whether a table is multi-tenant, and a tenant identifier is included to multi-tenant tables. To achieve tenant isolation, the query generator module needs to be extended to filter tenants' requests by a tenant identifier. However, this approach only supports query adaptation.

An abstraction layer [98] has been implemented for back-end customization through injection of custom business logic. The abstraction layer separates *business objects* from the actual database, where a *business object* encapsulates business data and associated business logic. When a tenant requests a *business object* from an application, the abstraction layer intercepts a tenant identifier from the request, and retrieves a corresponding model from tenant-specific models. Note that each business object is mapped to exactly one tenant-specific model that consists of attributes, methods and relationships to other *business objects*. Then, a *run-time object* is created for the requested *business object*. During and after processing the request, *business objects* are stored in a relational database by a data mapper. However, it is not clear how tenant data are isolated and which multi-tenant data architecture is applied.

Unlike other approaches, a middleware architecture PERSIST [104] has been implemented to enable dynamic configuration of **NoSQL** storage requirements for multiple cloud storage providers. The middleware consists of four different layers where the core layer is the *data management middleware layer*. This layer uses Java Persistence Application Programming Interface (**JPA**) and Java Persistence Query Language (**JPQL**) to provide a data access in a cloud storage provider agnostic way. It also plays a role of a communication interface between **SaaS** applications and the middleware. Moreover, the middleware is capable of selecting the most suitable data storage solution based on tenant-specific meta-information regarding data storage policies. Nevertheless, the middleware incurs the same limitations as previous approaches, hence, it neither considers other cloud storage solutions nor different data partitioning alternatives to isolate tenants in non-relational storage.

On top of the lamented implementation overhead, manual approaches are quite limited. Despite the fact that most real-world cloud applications exploit different

cloud storage solutions [125], almost all manual approaches we surveyed implement multi-tenancy in relational databases. Particularly, these approaches address multi-tenancy in a single shared database without considering other multi-tenant data architectures for relational databases. Surprisingly, only one approach [17] considers partitioning and tenant isolation in all three cloud storage options.

2.3 MDE Based Approaches

Modeling multi-tenancy at the data layer of cloud applications has been proposed in [MDE] based modeling languages. Most of these languages aim to automate the provisioning and deployment of cloud services. A few languages are geared towards portability and interoperability of cloud applications across different cloud providers, and migration of existing applications to the cloud by generating deployment specification models. In general, modeling languages have been implemented as extensions for general propose modeling languages, or as independent [DSLs].

2.3.1 Unified Modeling Language Extensions

[CAML] [15] has been proposed as an extension for Unified Modeling Language ([UML]) to express cloud-based deployments directly in [UML] models. Using this extension, a deployment topology is described in terms of [CAML] *Library*. Then, the deployment topology is refined by applying a dedicated [CAML] *Profile* to map deployments with cloud provider specific offerings. [CAML] *Library* uses common cloud modeling concepts that capture computing services (*e.g.*, operating system, web server, application container), cloud storage options, and cloud services (*e.g.*, load balancer, queue). [CAML] *Profile* comprises services from [GAE] and [AWS]. In [CAML], different cloud storage options (*i.e.*, block storage, blob storage, relational databases, and key-value storage) with consistency kinds (*i.e.*, strict or eventual) are captured in [CAML] *Library*.

Another [UML] profile [49] has been designed for modeling multi-cloud applications. *Cloud artifacts* are introduced in the profile to represent application components that can be deployed in a cloud platform. During the application modeling process, each cloud artifact requires a cloud-agnostic service type. The profile also allows to represent non-functional requirements for application components in terms of *property*, *operator*, and *value*. The model is later refined to represent cloud provider specific service instances, and processed by a model transformation engine to generate deployment plan with all cloud-related information. Using this [UML] profile, the data layer can be represented as a cloud artifact with a cloud storage option (*i.e.*, file storage, relational storage, and [NoSQL] storage) assigned as a service

type.

2.3.2 Domain-Specific Languages

An XML-based modeling language [8] has been provided by TOSCA to define application components and their relationships. Similarly, Cloud Modeling Language (CloudML)-Stiftelsen for Industriell og Teknisk Forskning (SINTEF) [14] has been proposed as a standalone DSL in the PaaSage⁸ project to express deployment specification of application components. In both approaches, the data layer of an application can be described as a separate component with database properties. Specifically, TOSCA XML enables to specify a database engine, a virtual machine to deploy the database, and an operating system that runs on the virtual machine. In contrast, CloudML-SINTEF captures other database properties such as a database engine, a data structure type associated with the database engine, and consistency of the storage type.

Furthermore, CloudML-SINTEF has been evolved in Model-Driven Approach for Clouds (MODA-Clouds)⁹ project to allow describing a data architecture associated with the applications data layer. The data architecture is expressed in terms of an Entity Relationship (ER) diagram and refined by a meta-model that specifies functional and non-functional data properties. Then, the data architecture is refined by cloud storage types (*i.e.*, distributed file system, NoSQL databases, and blob storage) offered by cloud providers in a cloud provider independent way. Later, the data architecture can be further refined by cloud provider-specific data structures to generate data definition scripts.

Cloudify DSL¹⁰ is also based on TOSCA to describe an application with its resources (*e.g.*, infrastructure, middleware, application code, scripts, tool configuration, metrics, and logs). The application descriptions are stored as blueprints in Yet Another Markup Language (YAML) documents. In addition to database related properties, Cloudify DSL allows to specify life cycle operations (*e.g.*, configure, create, start and stop) in a generic manner. Similarly, using Zephyrus [34] the data layer can be specified as an application component with quality constraints (*e.g.*, maximum number of components, minimum number of replicas).

Another XML-based modeling language, CloudML-Universidade Federal de Pernambuco (UFPE) [47], allows to describe the data layer in terms of cloud resources and services with their requirements. Stratus Modeling Language (StratusML) [51] also captures an application structure as a composition of cloud services. In StratusML a developer can specify a storage group that will be used to persist application's data

⁸<http://www.paasage.eu>

⁹<http://www.modaclouds.eu>

¹⁰<https://docs.cloudify.co/4.1.0/blueprints/spec-dsl-definitions/>

and describe different data partitioning strategies. The similar approaches are also supported by Holmes [57] and Blueprint [99]. Though the latter approach enables expressing service-based applications from a combination of different services from different cloud service models (*i.e.*, IaaS, PaaS, and SaaS). Thus, an application can be hosted by a combination of different cloud storage types from different cloud providers.

Move to Clouds for Composite Applications (MOCCA) [79] and CloudDSL [119] have been proposed to create deployment models to support migration of existing application to the cloud. MOCCA [79] provides a way to re-architect application components into groups of components where each group can be provisioned separately by different cloud providers. Whilst, CloudDSL [119] uses a common cloud vocabulary, for describing cloud IaaS services, to model an application architecture as an interconnection of cloud services. In both approaches, cloud storages are captured as services with storage related configurations such as region to deploy the storage, storage type and security rules to access the storage.

Almost all of the proposed modeling languages allow to capture different cloud storage services to generate deployment configurations. This certainly helps to automate deployment of application components to the cloud. However, none of the approaches explicitly enable to model a multi-tenant data architecture, or to produce source code from it. In addition, partitioning and implementation peculiarities of different cloud storage types have not been considered.

2.4 SPLE Based Approaches

SPLE is another commonly adopted software engineering paradigm to model multi-tenancy in cloud applications that focuses on the development of software products from reusable core assets [78]. In SPLE based approaches, feature modeling [69] and OVM [102] techniques have been widely exploited to express different functional and quality requirements of tenants in an application structure. The main focus of these approaches is to support multi-tenancy by enabling customizability and configurability of application components.

2.4.1 Feature Modeling Based Techniques

Feature modeling is a domain analysis technique to identify common and variable aspects of an applications [69]. It provides concepts to visually represent an application as a hierarchy tree of *features*. A *feature* is a functionality or characteristic of an application, and it can be identified as a *common*, *optional*, *alternative* or *at-least-one-of (OR)*. *Common features* compose a core of an application, and the

rest features represent variable functionality or quality that can be included in an application.

Feature modeling has been applied in [96], to capture configurable and customizable parts of an application. From a feature model, an application can be implemented as (i) *application-based binary*, or (ii) *feature-based binary*. In the former approach, multiple application instances are developed where each instance groups tenants with a different set of configurations and customizations. Whereas in the latter approach, an application is built using Service-Oriented Architecture (SOA) and application components are configurable to support multi-tenancy. In addition, a hybrid approach has been proposed that leverages benefits of both *application-based* and *feature-based* approaches.

A similar approach has also integrated feature modeling with SOA to re-architect an existing application to support multi-tenancy [124]. Feature modeling is applied to represent common core artifacts and customizable features of an application during domain analysis. While SOA is exploited to implement and deploy application components as microservices to support scalability. The data layer multi-tenancy is achieved by deploying a separate database instance per tenant, where a *tenant parameter* is added to requests to specify a tenant-specific database. The significant drawback of these approaches is that feature modeling is only used to analyze and classify configuration and customization options, while the application itself is manually implemented after the analysis and classification. Moreover, data layer related configurability and customizability alternatives have not been captured in an application structure.

Feature modeling has been extended in [25] to complement features with *attributes*. An *attribute* is a measurable characteristic such as price, cloud provider, and availability. The main goal of the extension is to allow selection of cost-effective cloud services for deployment. In this approach, relational and non-relational cloud storage services are grouped into a persistence feature, and blob storage services are captured as a file storage feature. Though, different partitioning and extensibility options of different cloud storage types have not been considered.

In another approach, Dynamic Software Product Line (DSPL) techniques [75] have been applied to realize a single shared multi-tenant SaaS application. In general, DSPL allows to implement an application that adapts its behavior at runtime through configurations [54]. In this approach, a *structural model* of the application is represented using service-oriented DSPL. A *structural model* provides a general abstraction of the services and their relationships that compose an application. In addition, *feature model* is exploited to enable creation and reconfiguration of services for each tenant. However, the data layer has been considered in neither *structural model* nor *feature model*.

Service Line Engineering (SLE) method has been described in [130] to model customizable multi-tenant applications. SLE is based on SPLE techniques, but, it proposes a single shared customizable application instance to meet different tenant-specific requirements. In SLE, functional and non-functional application requirements are initially modeled using feature modeling. From the feature model, a service line architecture is designed to describe all possible variations in the multi-tenant application. These variations are presented to tenants during the provisioning process. Thus, tenants select and parametrize features which are transformed into tenant-specific software configurations. In this method, a data layer could potentially be represented in a feature model with all different cloud storage types and their partitioning alternatives.

2.4.2 OVM Based Techniques

OVM is similar to feature modeling, except that it only captures variable functional and quality requirements of an application in terms of *variation points* with *variants* [102]. A *variation point* is a configurable functionality or quality, and a *variant* is an available variation option. As in feature modeling, *variation points* and *variants* can be either *mandatory*, *optional* or *alternative*.

In [94], OVM technique has been suggested to explicitly model different functional and quality features of an application in a separate view. The features are classified as *external* and *internal* features. The *external* features represent different requirements introduced by tenants, whereas the *internal* features are alternatives to implement different requirements imposed by the external features. In this approach, data segmentation patterns (*i.e.*, shared database and separate database per tenant) are captured as quality features and presented as configuration options for tenants during tenant on-boarding process. Based on selected configuration options, the application performs necessary deployments actions for each tenant.

OVM technique has also been exploited in [118] to define customizability alternatives in a separate model. The main difference of this approach from [94] is that the OVM is combined with Service-Oriented Architecture Modeling Language (SoaML). The role of the OVM model is to represent different application configuration options for tenants at run-time, where SoaML is used to define implementation alternatives during the application development. Such approach allows modeling different business process workflows, services that perform business processes and components that implement services. However, multi-tenancy at data layer was neglected because the author does not consider it as a customization point at run-time.

Another approach [128] that proposes OVM technique allows tenants to cus-

customize application's user interface, workflow, data layer, quality of service requirements, and services associated with the application. Multi-tenant data architectures for relational databases (*i.e.*, separate databases, shared database with separate schemas, and shared database with shared schema), and other database related options (*e.g.*, data storages, encryption, and schema enhancement) are captured as data layer customizability alternatives. During on-boarding process, a tenant selects from available customizability options and an application stores this information as a tenant-specific configuration.

Similarly, a multi-tenant application is modeled from three perspectives using **OVM** [62]: (i) *functional specification*, (ii) *realization and deployment alternatives*, and (iii) *device accessibility options*. Tenants select preferred options from these models, and a configuration descriptor is generated for each tenant that is used for deployment. In this approach, data layer related configurations are captured in *realization and deployment alternatives* perspective. The configurations include cloud storage types and deployment alternatives (*i.e.*, single instance, multiple instance, geo-specific).

OVM technique has been extended in [6] to model applications at the *meta-* and *base-levels*. *Meta-level* represents metadata associated with available configuration options, while *base-level* reflects functional and implementation alternatives within an application. Using this technique, multi-tenant data schemes could potentially be captured in a *meta-level* model, and implementation details of each storage type can be expressed in a *base-level* model. However, the extension mainly focuses on other application layers than the data layer.

In general, the presented modeling techniques are aimed to generate a tenant configuration by expressing application's different functional and quality requirements as configuration options during tenant on-boarding process. Nonetheless, these techniques require manual application implementation from a model. Furthermore, different cloud storage options and multi-tenant data partitioning models are only captured by a few approaches.

2.5 Hybrid & Other Modeling Approaches

Although **MDE** and **SPL** provide well-known techniques to model an application structure, some modeling approaches integrate the strengths of both to model multi-tenancy. In contrast, a few approaches exploit other modeling techniques to express customizability options in an application and validate their compatibility.

A framework for modeling multi-tenant cloud services from multiple architectural perspectives has been described in [88]. The framework is based on **MDE** principles and combines the strength of feature modeling, **UML** and **SoaML**. Feature modeling

is used to capture application functionality and quality requirements. Meanwhile, [UML](#) and [SoaML](#) are exploited to define different implementation and deployment options. Firstly, a *core model* is created that contains common architectural artifacts that must be present for every tenant. Then, a *tenant model* is created for each tenant that consists of the *core model* with tenant-specific artifacts. As a result of merging the *core model* with the *tenant models*, a *multi-tenant model* is generated that is capable to meet requirements of all tenants. However, the framework does not consider the implementation of multi-tenancy from the data layer perspective.

The framework has been implemented in [\[97\]](#) with a set of evolution rules and model-to-model transformations to manage the evolution process. The evolution process comprises on-boarding a new tenant, customizing existing tenants, and removing tenants. The implementation extends the framework by expressing different multi-tenant data partitioning options for every functional part of an application that interacts with the data layer. Moreover, the extension includes model-to-text transformations to produce a source code from the multi-tenant model. One of the main drawbacks of this approach is that a tenant can choose preferred workflows and multi-tenant data models for each selected application functionality. This requires more development effort to implement such application code. Furthermore, the implementation details of data partitioning alternatives and different storage types that can be potentially used in cloud applications have not been described.

In [\[105\]](#), an application is modeled as a combination of web services based on [SOA](#) and orchestrated by a Business Process Execution Language ([BPEL](#)). To support multi-tenancy, configuration options are introduced for tenants, and a *tenant context* is used to keep tenant-specific information (*i.e.*, tenant identifier, or authentication information). In addition, services are distinguished as *non-configurable* and *configurable services*. *Non-configurable services* behave in the same way for all tenants, while *configurable services* can be customized on a per tenant basis to meet tenant requirements. Services are further classified in three instance types: *single instance*, *arbitrary instance*, and *multiple instance*. The database services offered by cloud providers can also be modeled following this approach. Nevertheless, authors aim to generate tenant-specific deployment scripts from the application model.

A metagraph modeling tool has been proposed to manage different configuration and customization alternatives of multi-tenant [SaaS](#) applications [\[82\]](#). A *metagraph* is a graphical structure that represents relationships between sets of elements. Using this tool, a multi-tenant application is described as a *metagraph* that comprises *component units* and their relationships, where a *component unit* can be elements of data schema, application logic and user interface. Similar approach has been described in [\[117\]](#). Despite the fact that metagraph tool is beneficial to analyze and validate relationships between component units, it does not allow to generate other

artifacts from the application model.

Customization of application services, business processes and data layer has been modeled based on Temporal Logic of Actions [84]. Another approach [81] has exploited directed graphs to model customization of relationships in data layer, services, business processes and user interface components. The data layer customization is modeled in terms of data objects with data fields and their interrelations. The main contribution of these approaches is that they verify compatibility of tenant-specific customizations and compliance with rules imposed by SaaS provider.

2.6 Summary

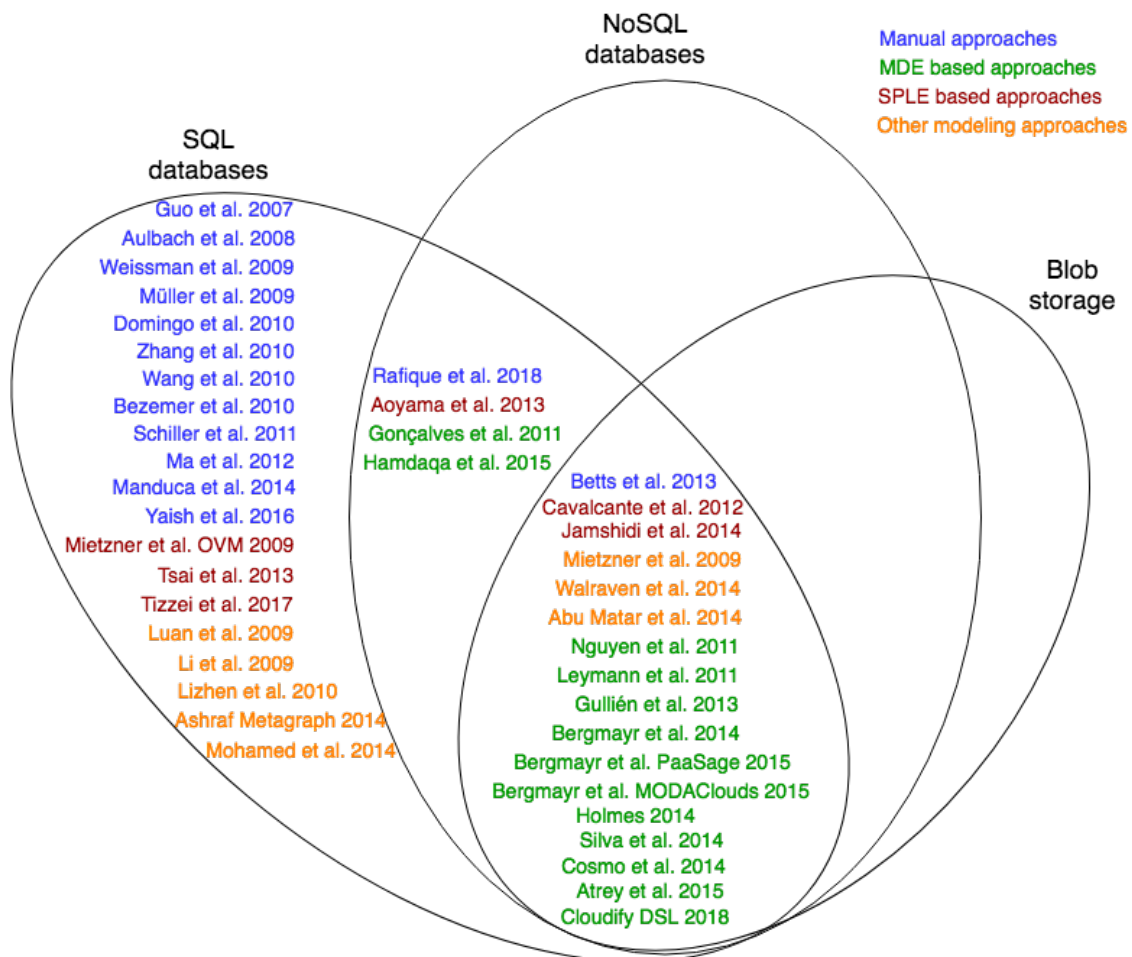


Figure 2.1: Overlap of approaches between SQL databases, NoSQL databases, and blob storage.

Current approaches pursue very similar goals but differ in scope, thus, provide partial overlapping as presented in Figure 2.1. In particular, manual approaches dominantly discuss multi-tenancy concerns in relational databases. Similarly, nearly half of the modeling techniques based on SPLE and other modeling approaches sup-

ports relational databases. Whereas the other half captures all three cloud storage options. Most of the [MDE](#) based modeling languages also cover different cloud storage types. Note that three modeling approaches comprise only relational and non-relational databases.

The majority of the manual implementation proposals partition a single shared database and isolate tenant data in it. On the contrary, [MDE](#) based modeling languages allow to describe different cloud storage types as application components and their deployment configurations. Meanwhile, [SPLE](#) based modeling techniques capture different customization and configuration alternatives in an application. Modeling techniques based on other modeling approaches also tackle defining customization and configuration options in an application, but also provide a way to validate their compatibility.

Figure [2.1](#) also emphasizes that non-relational databases and blob storage have not at all been considered separately. There are a few reasons that may contribute to this. First, non-relational databases are captured as an alternative for relational databases. Second, blob storage services are commonly used as a supplementary storage to persist backup and media files. Finally, non-relational databases and blob storage are presented as available cloud storage types in addition to relational databases. Nevertheless, most real-world cloud applications are implemented using a combination of various data storage solutions as each cloud data storage type works better and more efficiently for different tasks [\[125\]](#).

Interestingly, most of the latest approaches propose modeling techniques. This is due to the fact that modeling techniques provide an abstraction layer that allows to model application components in a cloud provider independent way. Furthermore, modeling techniques offer model-to-model and model-to-text transformations to enable (semi-)automation of cloud application development and deployment [\[22\]](#).

Nevertheless, all of the proposed approaches require manual implementation. Even in modeling techniques, a model of an application is only used to represent different configuration and customization alternatives during tenant on-boarding process, or to describe deployment of application components on cloud services. Moreover, none of the existing modeling languages consider conceptual and implementation differences, or partitioning and tenant isolation in available cloud storage types. This was also emphasized in [\[68\]](#) as an important issue that requires wider research discussions. Hence, there is a strong need for a modeling language that would enable modeling a multi-tenant data architecture as a combination of different cloud storage solutions in an abstract way. The modeling language should also provide tools to transform a data architecture into application code to (semi-)automate application development process.

Chapter 3

Proposed Solution

Introducing multi-tenancy affects all layers of the application structure, particularly, in terms of development and evolution overheads, and the data layer is no exception. Multi-tenancy at the data layer requires a data architecture to maintain data separation of different tenants. The data architecture typically also needs to be extensible to support tenant-specific customizations. A further complication is the tendency to store data in several storage types [125], *i.e.*, relational databases, non-relational databases, and blob storage. These different storage types are conceptually diverse, with each having its own partitioning and extensibility approaches to support multi-tenancy. In essence, these concerns can be encapsulated into a DSL for generating and/or maintaining cloud application implementation [1].

There have been some approaches in this direction that are discussed in Chapter 2. In brief, existing DSLs allow to model deployment specification of the data layer on cloud storage services, and to generate deployment descriptors from the model. However, none of the modeling languages provide a way to model a multi-tenant data architecture as a combination of different cloud storage types, or produce data access layer code from the data architecture.

Therefore, Cloud Application Modeling Language (**CadaML**) has been proposed as the main contribution of this thesis to fill the gap in the current research. **CadaML** provides concepts and notations to design a data architecture of multi-tenant cloud applications in a cloud provider independent manner. Moreover, the concepts and notations enable to explicitly define different cloud data storage solutions offered at the PaaS service level in a single data architecture model, and to specify data partitioning options for each data storage. **CadaML** also provides model validation and code generation capabilities. The model validation ensures compliance of a model created using **CadaML** with semantics of the meta-model and additional custom constraints. While the code generation produces executable data access layer code for three major cloud service providers.

This chapter, firstly, describes requirement specifications in Section 3.1 regarding

CadaML concepts and a meta-modeling language that will be used to implement it. Then, Section 3.2 presents the methodology applied to develop CadaML. The methodology proposes domain analysis, design and implementation phases that are presented in Sections 3.3, 3.4, 3.5, respectively. Finally, Section 3.6 validates the implementation of CadaML against the requirements presented in Section 3.1, and Section 3.7 concludes this chapter.

3.1 Requirements

While the development of CadaML requires a thorough analysis of the problem domain, there are also requirements that should be identified and specified beforehand. In particular, it is important to specify two sets of requirements, one regarding concepts and terminology that will be used in CadaML, and another related to a meta-modeling language to develop and deploy CadaML. The former requirements are applied when capturing domain concepts, while the latter requirements are considered when selecting a meta-modeling language to implement CadaML. The requirements are defined based on the requirements analysis and design guidelines [44] that are formulated through the experiences for enterprise modeling, and motivated to fill the gap identified in Chapter 2.

3.1.1 Concepts and Terminology Requirements

The prospective applications and users of CadaML are multi-tenant cloud applications, and cloud data layer architects and developers, respectively. Thus, CadaML should offer concepts and notations that are simple, comprehensive and convenient to model a data architecture of multi-tenant cloud applications.

In general, concepts and notations of a modeling language should provide *ontological clarity* and *ontological completeness* [44]. *Ontological clarity* demands that each concept maps to exactly one concept of the ontology. While *ontological completeness* demands that a modeling language covers all basic concepts to represent elements of the target domain. In correspondence with these ontological demands, the following requirements related to concepts and notations of CadaML are composed.

- CR1: The concepts and notations of CadaML should correspond to terminology that cloud data layer architects and developers are familiar with. Commonly, existing terminology of the target domain are reconstructed, and graphical notations are used to illustrate corresponding meaning of concepts. Using familiar domain concepts and their representations in CadaML will help the

prospective users to ease understanding and applying them properly when exploiting the language.

- CR2: Semantics of the concepts and notations of CadaML must be invariant within the scope of the language's application. Semantic invariance eliminates ambiguity and ensures explicitness of concepts and notations of CadaML.
- CR3: The concepts and notations of CadaML should be expressive enough to extract other target representations from the model. The target representations may include application code, documentations or any textual artifact that are characterized by various semantic distinctiveness. Hence, the model should contain all the required information by the target representations in order to generate complete artifacts.

3.1.2 Meta-modeling Language Requirements

There are many meta-modeling languages and tools that support implementation and exploitation of DSLs. For CadaML, choosing one meta-modeling language over another needs consideration of the following requirements.

- MR1: A meta-modeling language should support implementation of graphical DSLs. In essence, a DSL can be implemented with either graphical or textual interface [43]. For CadaML, a graphical interface is preferred as visual representation eases the understanding and modeling a data architecture. However, a graphical interface requires mapping domain concepts to corresponding graphical notations that can be efficiently handled by a meta-modeling language.
- MR2: A meta-modeling language should provide a meta-modeling environment that supports realization of a model editor for CadaML. The implementation of a model editor requires a major development effort which can be facilitated by an effective support from a meta-modeling environment.
- MR3: A meta-modeling language should provide a model validation tool to keep a model consistent by enforcing constraints and validation rules. A validation tool allows specifying additional constraints to resolve ambiguities at the model level, and to ensure semantic correspondence of generated artifacts with target representations.
- MR4: A meta-modeling language should provide a model-to-text transformation tool to generate code from a model. Code generation requires parsing a model to an application code which can be implemented in a general-purpose programming language such as Java, or in a model transformation tool provided

by a meta-modeling environment. Compared to general-purpose programming languages, model transformation tools can offer advantages, such as syntax, to efficiently manipulate model elements which eases the implementation of the code generator.

These requirements are important to guide the design of concepts, and selection of a suitable meta-modeling language for CadaML. They will also be revisited to validate the CadaML implementation decisions.

3.2 Methodology

DSL development requires defining an *abstract syntax*, *semantics* and a *concrete syntax* of a modeling language [48]. The *abstract syntax* describes elements that compose a modeling language, and composition rules of those elements. The *semantics* of the language define the meaning of elements and their relationships. The *concrete syntax*, in turn, determines a language interface for language users which can be either graphical or textual [43]. The *abstract syntax* for graphical **DSLs** are defined by *meta-models*, whereas for textual **DSL** it is defined by *grammars*.

For CadaML, once the requirements are specified, the implementation is planned by following the **DSL** development methodology provided in [91] as it enfolds existing literature on **DSL** development methodologies (*i.e.*, [31, 60, 109, 122]), and provides generic patterns and approaches to systematically develop and deploy **DSLs**. The development methodology includes *domain analysis*, *design*, and *implementation phases* as presented in Figure 3.1. This section describes these phases in regards with implementation of CadaML.

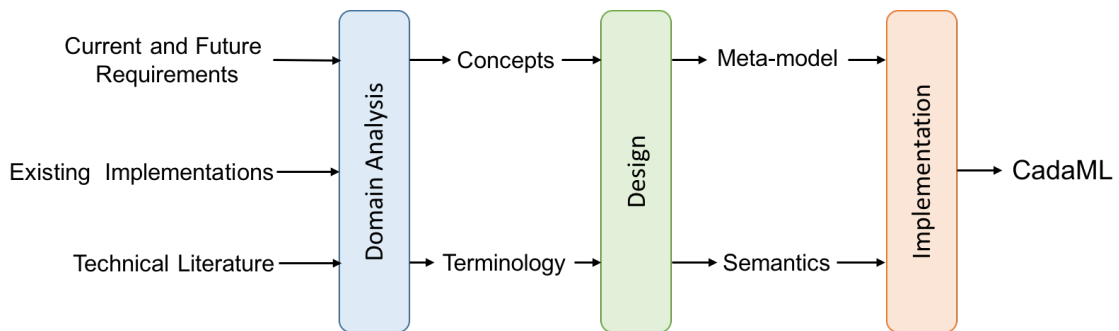


Figure 3.1: The development phases of CadaML following the methodology presented in [91].

During the *domain analysis phase*, the domain knowledge related to different cloud storage types and multi-tenancy patterns are collected, which are then composed into concepts and notations that will be used in the *design phase*. The domain knowledge are gathered from various resources such as technical literature, existing

implementations, and current and future requirements. As an outcome of the *domain analysis phase*, a *domain model* is produced that consists of the concepts and terminology used in the target domain.

The *design phase* includes identifying the relationship of CadaML to existing modeling languages and specifying its design. The relationship identification is needed to figure out whether to implement CadaML based on an existing language or to build it as an independent modeling language. The design specification is important to formulate domain concepts used in CadaML. Then, the *meta-model* and *semantics* of CadaML are defined, being derived from the *domain model* composed during the *domain analysis phase*.

In the *implementation phase*, firstly, the *meta-model* of CadaML is mapped to a graphical *concrete syntax*. For CadaML, the graphical interface is preferred because of the following benefits. First, visual representation of a data architecture makes designing database elements and relationships among them more convenient. Second, it is easier to find and correct errors in a graphical model [55]. Finally, visualization of a model allows non-developers to get an overview of a data architecture and intuitively develop an understanding of the data layer design. Then, constraints and validation rules are specified to ensure consistency of a model in CadaML. Finally, model-to-text transformation is defined to produce data access layer code from the model.

3.3 Domain Analysis

The objective of the *domain analysis phase* is to capture concepts and notations that are abstract enough to provide a unified representation of different types of cloud data storage services offered by major cloud service providers. To achieve this, primarily, the commonality and variability in concepts and terminology to describe available cloud data storage solutions by public cloud service providers are analyzed. Then, peculiarities of cloud data storage partitioning techniques by industrial and academic studies are considered. Finally, characteristics of current modeling languages that support cloud application development are investigated.

3.3.1 Cloud Data Storage Types

As a first step of the domain analysis, available storage types of widely used cloud providers (namely Alibaba Cloud, [AWS], [GAE], and Microsoft Azure) have been compared and analyzed. Specifically, storage services at the [PaaS] provisioning level are considered because they provide on-demand scalability, and they are the most commonly exploited by developers [131].

In general, cloud providers offer similar data storage solutions under different names that can be grouped into relational databases, non-relational databases and blob storage. Table 3.1 presents a high-level mapping of these storage types to available storage services of each cloud provider.

Table 3.1: Mapping cloud storage types to storage services of public cloud providers

	Relational Databases	Non-relational Databases	Blob Storage
Alibaba	ApsaraDB for RDS	Table Store	Object Storage Service
AWS	Amazon RDS	DynamoDB	S3
Azure	Azure SQL Databases	Table Storage	Blob Storage
GAE	Cloud SQL	Cloud Datastore	Cloud Storage

Despite offering varying services, cloud providers adhere to the same core principles of storage organization, but use different terminology and concepts. As an example, analyzed non-relational databases support schemaless data model, though a concept that defines a single data in a database differs based on the cloud provider. As another example, cloud providers exploit various terms to represent an unstructured data item in a blob storage.

Relational Database Services

Relational database services offered by cloud providers have all the capabilities and functionality of a traditional relational database, with a few additional features. Like a traditional relational database, these services are appropriate for structured data with a well-defined schema. Data is organized in *tables*, *rows* and *columns/fields*, and a *primary key* identifies each *row* in a *table*. Relationships among *tables*, *columns* and other database elements are strongly defined in the data model. As opposed to a traditional database, cloud based relational databases are fully managed by cloud providers that makes them easy to set up, maintain, manage, and scale.

Non-relational Database Services

While non-relational database services have many of the same characteristics as relational database services, they differ from them in the way they describe relationships between data objects. A comparison of concepts of non-relational cloud services and relational databases are presented in Table 3.2.

As in relational databases, most of the non-relational databases organize data in *tables*. However, non-relational databases are schemaless as they do not require *rows* of the same *table* to have a consistent set of *columns* except a *primary key*. Hence, *rows* of the same *table* can have different *columns*, and different *rows* can have

Table 3.2: Comparing concepts of non-relational databases to relational database

SQL Concepts	Cloud Providers			
	Alibaba	AWS	Azure	GAE
Table	Table	Table	Table	Kind
Row	Row	Item	Entity	Entity
Column	Column	Attribute	Column	Property
Primary Key	Primary Key	Partition Key	Partition Key	Key
Composite Primary Key	Primary Keys	Partition Key and Sort Key	Partition Key and Row Key	Does not support

columns with the same name but different value types. A *primary key* (also called *partition key*) determines the partition in which data will be stored. In addition, most of the cloud providers support a *composite primary key* as a combination of *partition key* and *row key*, where *row key* identifies data within each partition.

Non-relational databases also provide on-demand scaling to maintain high performance when they receive more traffic. On the contrary, queries that can be executed are more restrictive than those allowed on a relational database. Specifically, non-relational databases do not support join operations, inequality filtering on multiple columns, or filtering on data based on results of a subquery.

Blob Storage Services

Blob storage services allow to store unstructured data such as documents, media files, or binary data, in the cloud. Blob storage is also referred to as object storage, and it can be compared to filesystem. The comparison is presented in Table 3.3.

Table 3.3: Comparing concepts of blob storage to filesystem

Concepts of Filesystem	Cloud Providers			
	Alibaba	AWS	Azure	GAE
Folder	Bucket	Bucket	Container	Bucket
File	Object	Object	Blob	Object
File name	Object name	Key	Blob Name	Key

In blob storage, *buckets* are the basic containers to store and organize data. Data is stored as *blobs* or *objects*, where an *object/blob name* or a *key* uniquely identifies each *blob* within a *bucket*. Unlike directories and folders, *buckets* cannot be nested.

3.3.2 Data Architecture Partitioning Schemes

A partitioning scheme is crucial to ensure isolation of tenant data, and scalability of the solution when sharing application code and data across all tenants. Typically, each cloud storage type has its own partitioning techniques that are described in this subsection.

Relational database partitioning schemes are summarized through analyzing academic and industrial work. Meanwhile, partitioning schemes for non-relational databases and blob storage are classified based on guidance and patterns from cloud providers (e.g., [17, 58]).

In general, relational databases can be partitioned using one of the following three ways. (i) *Separate databases*: each tenant is served by a dedicated database; (ii) *Shared database, separate tables*: all tenants are hosted by a single database with separate tables per tenant. A tenant identifier can be included in the table name, or a different database schema can be used for each tenant; (iii) *Shared database, shared tables*: all tenants share tables in a single database, with a tenant identifier is used to associate their records in each table.

Non-relational databases can be partitioned in one of two ways: *separate tables* or *shared tables*. In the former, each tenant's data is stored in tenant-specific tables with a tenant identifier as part of table names. In the latter, all tenant data is stored in shared tables and a tenant identifier is included in partition keys to associate rows with a tenant.

Separate buckets and *shared buckets* are the main partitioning techniques for blob storage. In *separate buckets*, all blobs belonging to a specific tenant are stored in a single bucket where a tenant identifier is included in the bucket name. In contrast, *shared buckets* stores all tenant data in the same buckets, but includes tenant identifiers in the blob names.

3.3.3 Current Cloud Application Modeling Languages

Current modeling languages that support cloud applications usually allow to describe a deployment specification of the data layer to different cloud storage types. However, concepts and terminology used by modeling languages differ from each other. Consequently, there are no standardized concepts to represent different cloud storage services. For example, a UML profile [49] comprises *file storage*, *relational storage* and *NoSQL storage* as available cloud storage services, whilst a UML extension CAML [15] captures two types of data structures (namely, *key value* and *relational*) that are further refined by cloud storage services of GAE and AWS. As another example, a few DSLs (e.g., CloudifyDSL¹ and CloudML/UFPE [47]) use

¹<https://docs.cloudify.co/4.1.0/blueprints/spec-dsl-definitions/>

distributed file system, *NoSQL databases*, and *blob storage* concepts to define storage components of an application. Interestingly, *CloudML, SINTEF* [14] exploits concepts of entity relationship diagram to allow modeling a data architecture of cloud applications. Among those various terminology that represent different cloud storage services, most commonly used concepts are *relational database/storage*, *NoSQL database/storage*, and *blob storage*.

3.3.4 Domain Analysis Output

As a result of analyzing the above state of the art in the domain of cloud data storage, different types of data storage solutions and their partitioning alternatives have been composed into a domain model which is illustrated in Figure 3.2. The domain model presents a unified modeling view across different data services from the surveyed four major cloud service providers. In the domain model, the existing cloud data storage terminology is reused and reconstructed, where needed, to provide unambiguous and expressive concepts that correspond to the requirements described in Section 3.1.1. This model forms the basis of CadaML implementation as a *meta-model* is derived from it, subsequently, a graphical syntax is generated from the *meta-model*.

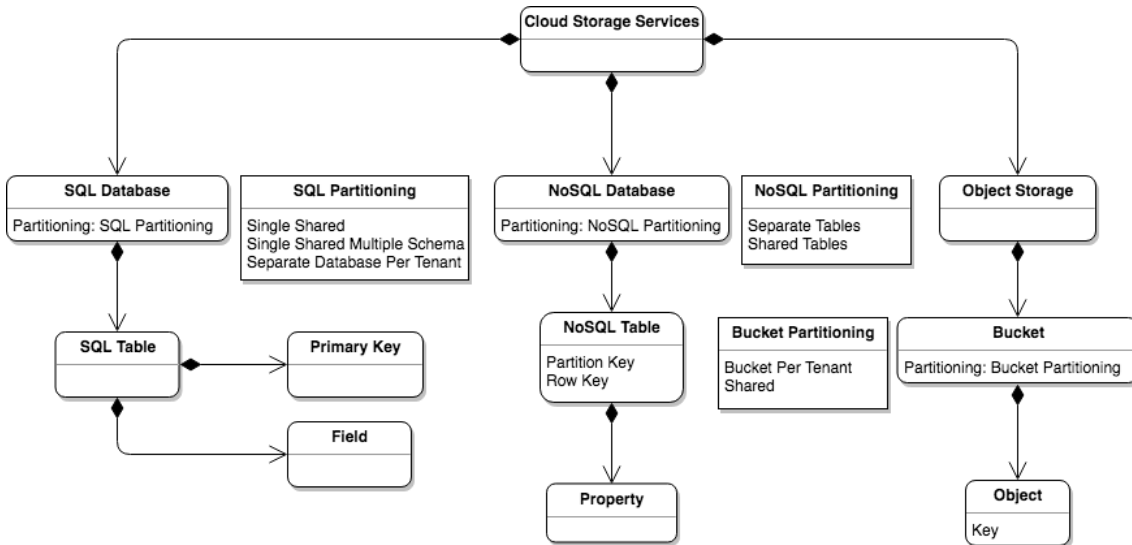


Figure 3.2: Selected concepts and terminology for CadaML.

Because all public cloud providers use a common vocabulary to describe a relational database, existing terminology is adopted to capture elements of a relational database in the modeling language. *SQL database* represents a relational database with its partitioning schemes (*i.e.*, *SQL partitioning*), and it consists of instances of *SQL table*. A *SQL table* is a collection of related data entries that contains *fields*, and it must have at least one *primary key*.

On the contrary, cloud providers exploit different concepts to define a non-relational database. Thus, concepts that are used only in the non-relational database

are selected. Specifically, *NoSQL table* is associated with a non-relational table, and it contains fundamental data elements called *properties*. *Partition key* is mapped to *primary key*, and *row key* is captured to produce a composite primary key. In the meantime, partitioning options for non-relational databases are grouped into *NoSQL partitioning*.

For blob storage, the most commonly used terminology in all four cloud blob storage are chosen. The concepts include *bucket* which is the basic container to store *objects*, *key* that identifies each *object* within a bucket, and *bucket partitioning* to express available partitioning alternatives.

It is worth mentioning that tenants have not been included in the domain model as a first-class entity. The rationale behind this design decision comes from a realization that tenants could be represented as an independent entity in a higher level of abstraction but not in the data architecture. Nevertheless, tenants implicitly exist in a form of partitioning options for each storage type. Moreover, the designed domain model provides modelers a flexibility to implement tenants in any form in any cloud data storage solution.

3.4 Design

Current cloud application modeling languages cannot be extended or reused to implement CadaML, as they do not support modeling a data architecture of cloud applications and particularly multi-tenancy therein. Thus, CadaML requires a novel *meta-model* which is derived from the domain model that captures concepts and terminology of different cloud storage types and their partitioning schemes. In order to design a *meta-model*, firstly, the relationships between CadaML and existing modeling languages are identified. Then, the design of CadaML is specified in accordance with the requirements imposed in Section 3.1.2. Finally, a *meta-model* and *semantics* of CadaML are produced.

3.4.1 Language Exploitation versus Language Invention

A modeling language can be designed by either exploiting an existing language or inventing a new one [91]. In the former, a modeling language is based on an existing language where *notations* and *semantic* concepts of the existing language are used, restricted or extended. In the latter, a modeling language is created with no commonality with existing modeling languages.

Both of these design patterns have advantages and drawbacks. In particular, when a language exploits an existing one, the *notations* and *concepts* are consistent with the host language with provided compiling and parsing. Though, the modeling

language is constrained by the host language. On the other hand, creating a new modeling language is more flexible in terms of deciding language concepts, terminology, and structure. However, a *meta-model* of the language must be defined, as well as a compiler to parse and process the *meta-model*, and to map the *meta-model* to the expected *semantics*.

Because none of the existing modeling languages that are described in Sections 2.3 and 2.4 support modeling a data architecture of cloud applications, there is no way to design CadaML based on concepts of these modeling languages. Therefore, it is decided to implement CadaML as an independent modeling language. This, in turn, requires a *meta-model* that covers the domain concepts and terminology that are deemed necessary based on the analysis in Section 3.3.

3.4.2 Design Specification

After the relationship to existing languages has been determined, the design of CadaML must be specified before implementation. The design specification can be distinguished between *informal* and *formal* designs [91]. In the *informal design*, the specification is typically written in natural language. Subsequently in the *formal design*, the specification is produced in a form of a *meta-model*.

The *informal design* is easier to perform compared to the *formal design*, though it can contain imprecisions that cause problems in the implementation phase. In contrast, formal specification of both *meta-model* and *semantics* can capture problems before implementation. Furthermore, *formal design* is commonly implemented by tools that significantly reduce implementation effort. As a result, *formal design* is applied to formulate *meta-model* and *semantics* for CadaML.

3.4.3 CadaML Meta-model

At the heart of a graphical DSL is the definition of a *meta-model* that captures concepts and relationships of the problem domain. Based on the *formal design* specification, a *meta-model* is commonly specified using a meta-modeling language. Most of the current meta-modeling languages are provided as a part of a framework or tool suite that supports development and deployment of modeling languages. There are a few widely exploited frameworks that are described and compared in Section 3.5.1.

As a result of thorough consideration and analysis of existing frameworks, and requirements regarding a meta-modeling language specified in Section 3.1.2, the *meta-model* of CadaML is defined in Ecore² model that is depicted in Figure 3.3. The *meta-model* is derived from the domain model presented in Section 3.3.4, and it

²The justification of this selection is given in Appendix C

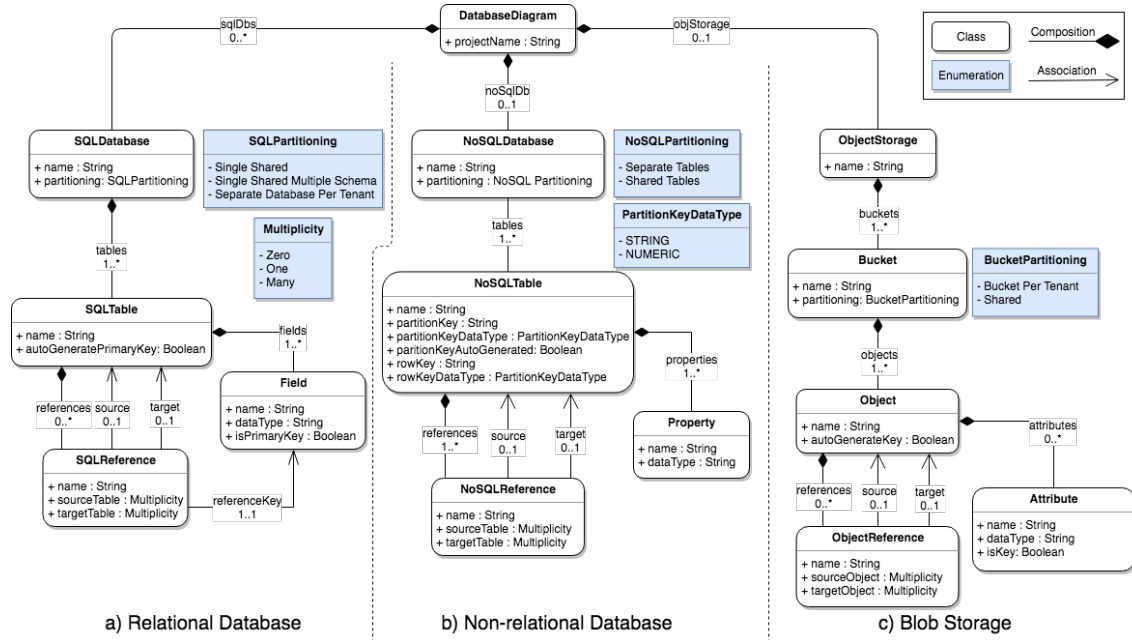


Figure 3.3: CadaML meta-model in UML class diagram.

is divided into three parts that cover domain concepts and the interrelations there in of a) relational databases; b) non-relational databases; and c) blob storage.

The main element of the meta-model is *DatabaseDiagram* that represents a diagram in a graphical editor where a cloud data layer architect or developer (hereafter modeler) designs a data architecture. A diagram may include *SQL Database*, *NoSQL Database* and *Object Storage*.

Relational databases are expressed by *SQL database*. *SQL Partition* of a relation database is classified according to partitioning schemes that were described in Section 3.3.2. A *SQL database* is composed of tables and their relationships that are represented by *SQL table* and *SQL reference*, respectively. A *SQL table* consists of *fields*, and each *field* has *name*, *data type* and *isPrimaryKey* parameters where the last parameter defines whether the field is a primary key. In addition, *autoGeneratePrimaryKey* parameter allows to automatically generate primary key values of a table by the application. The *source* and *target* parameters of *SQL reference* refer to tables in a relationship, and *reference key* indicates to a foreign key in a *target* table. Where multiplicity between tables are expressed by *source table* and *target table* parameters.

NoSQL Database represents non-relational databases with its partitioning schemes, and it consists of *tables* (i.e., instances of *NoSQL table*) and their interrelations. A *NoSQL table* is a collection of *properties*, where a *property* is a fundamental data element with *name* and *data type*. A *NoSQL table* must have a *partition key* and a *row key* with their data types (i.e., *STRING* or *NUMERIC*), where *partition key* values can be automatically generated by the application by setting

partitionKeyAutoGenerated parameter to *true*. In the meantime, the relationships among tables are represented by `NoSQL` reference, where *source table* and *target table* parameters refer to multiplicity (*i.e.*, *ZERO*, *ONE*, and *MANY*) between tables.

Object Storage is associated with Blob storage type. In blob storage, data is stored in *buckets*. A developer can specify *partition* of a *bucket* to one of the described in Section 3.3.2 partitioning schemes. *Object* represents a blob that is persisted in a *bucket*. An *object* is a set of *attributes*, where each *attribute* has *name*, *data type* and *isKey* parameters. The *isKey* parameter determines whether an *attribute* is a key that will be associated with the *object*. A *key* for a blob can be automatically generated by setting *autoGenerateKey* parameter of an *object* to *true*. An *object* can be in relationships with other *objects* which are expressed by *object reference*. The *source* and *target* parameters refer to blobs in a relationship, while multiplicity between blobs are expressed by *source object* and *target object* parameters.

It can be clearly seen from Figure 3.3 that reference elements (*i.e.*, *SQLReference*, *NoSQLReference*, and *ObjectReference*) in each storage type have the same attributes and relationships with storage elements. This formulates a recurring pattern that could be refactored using the concepts of inheritance. However, we decided to create a separate reference element for each storage type for the following two reasons: (i) mitigate the representation and comprehensibility of the meta-model; and (ii) minimize the complexity of implementing model validation and code generation capabilities.

3.5 Implementation

The aim of the *implementation phase* is to develop a modeling environment for CadaML that supports model validation and code generation. Model validation is important to ensure consistency of a model with the semantics of the *meta-model* and additional custom constraints, while code generation is necessary to produce data access layer code to (semi-)automate data layer implementation.

During this *implementation phase*, current frameworks and tools to implement `DSLs` are considered to select a most suitable one for CadaML (Section 3.5.1). Then, the frameworks and tools are analyzed and compared regarding the availability of the framework as an open source and the meta-modeling language requirements specified earlier (Section 3.5.2). Once a framework is selected, a *meta-model* of CadaML is designed, a graphical editor is generated from the *meta-model* (Section 3.5.3), and additional capabilities, such as model validation (Section 3.5.4) and code generator (Section 3.5.5), are implemented.

Hence, the full implementation process of CadaML consists of the following steps:

(i) the *meta-model* is created using Emfatic³ which offers textual notations for defining Ecore models; (ii) the *meta-model* is annotated using the EuGENia tool that provides annotations to automatically transform the notations from the *meta-model* into a concrete graphical modeling editor; (iii) the graphical editor is generated from the *meta-model*; (iv) custom constraints and validation rules are specified using Epsilon Validation Language (EVL); and (v) the code generator is defined using Epsilon Generation Language (EGL).

Overall 4684 Lines of Code (LoC) has been written to implement CadaML. This is broken down to 203 for the implementation of the meta-model, 106 for adjusting the graphical editor, 315 for validation, and 4,060 for code generation.

3.5.1 Graphical DSL Implementation Frameworks and Tools

DSL development is typically supported by frameworks that provide tools to create a *meta-model*, produce a modeling editor from the *meta-model*, specify model validation, and implement code generation. There are some widely used frameworks, such as Eclipse Modeling Framework (EMF), Eclipse Graphical Modeling Framework (GMF), Epsilon Framework, MetaEdit+ and Modeling SDK for Visual Studio (MSDK). These frameworks are briefly described individually before being compared against each other in Section 3.5.2. The aim of the comparison is to guide the selection of a suitable meta-modeling language to implement CadaML.

EMF and GMF

EMF⁴ is an Eclipse-based modeling framework and code generator facility to specify, construct and manage DSLs. EMF provides Ecore meta-modeling language to define a *meta-model* of a modeling language which can be described through various methods such as XML Metadata Interchange (XMI), Java annotations, UML and an XML scheme. From a *meta-model*, EMF generates a set of generic classes to construct a domain-specific modeling editor. In turn, the EMF code generator facility produces all necessary classes and a structured editor to build a complete modeling environment.

Nevertheless, EMF requires extensions or manual refactoring of the generated classes to create custom domain-specific visualizations. Moreover, neither Ecore nor EMF supports specification of additional domain constraints. To address these problems, Eclipse GMF⁵ is proposed to manually implement a custom editor, and Java-based Object Constraint Language (OCL) library [129] is introduced to specify constraints into the generated classes.

³<https://www.eclipse.org/emfatic/>

⁴<https://www.eclipse.org/modeling/emf/>

⁵<http://www.eclipse.org/modeling/gmf/>

GMF is a framework for developing graphical modeling editors using **EMF**. The main components of **GMF** are the *tooling* and the *run-time*. The *tooling* includes editors to specify and manage models that describe notations, semantics and tooling aspects of a graphical editor. It also includes a generator to produce the implementation of graphical editors, from which the *run-time* produces an extensible graphical editor.

Epsilon

Epsilon [72] offers a set of languages and tools for implementing **DSLs**. EuGENia⁶ is one of these tools that generates all necessary models to produce a **GMF** editor from an annotated *meta-model*. As in **EMF**, EuGENia uses Ecore meta-modeling language to define *meta-models*, and it provides high-level annotations to facilitate the complexity of implementing a **GMF** editor.

Epsilon also provides Epsilon Validation Language (**EVL**) [72] to specify constraints and validation rules. **EVL** is based on Eclipse Object Language (**EOL**) [72] which is an imperative programming language for constructing and managing **EMF** models. Despite the similarities of **EVL** constraints with **OCL** constraints, **EVL** allows defining dependencies between constraints, displaying custom error messages, and specifying fixes that can be invoked to repair inconsistencies in a model.

When using Epsilon, model-to-text transformation is implemented using Epsilon Generation Language (**EGL**) [72]. **EGL** is a template-based language for generating code, documentation and other textual artifacts from models. **EGL** uses a mixture of *static* and *dynamic sections* to manipulate the output of the transformation. *Static sections* include hand-written text or code, whereas *dynamic sections* can contain **EOL** statements. Furthermore, **EGL** provides several features, such as generating text to a variety of sources, formatting algorithms, and linking generated text with source models, to simplify and support the generation of texts from models.

MetaEdit+

MetaEdit+⁷ is a commercial domain-specific modeling environment that supports both development and exploitations of graphical **DSLs**. A modeling language is designed with MetaEdit+ Workbench, and the modeling language is used in MetaEdit+ Modeler.

In MetaEdit+ Workbench, a *meta-model* is described as a set of objects using the Graph, Object, Property, Port, Relationship and Role (**GOPRRR**) [92] meta-modeling framework. The meta-modeling process includes the following steps.

⁶<https://www.eclipse.org/epsilon/doc/eugenia/>

⁷<https://www.metacase.com/>

Firstly, language concepts and their composition rules are defined either graphically or using form-based meta-modeling tools. Then, the concepts are associated with visual notations. The notations can be drawn in Symbol Editor or imported from existing graphical elements. Model validation is based on the semantics of the specified *meta-model* which is automatically supported by the framework. Finally, model-to-text transformation is specified to produce required artifacts such as code, configuration, or testing data. Following the meta-modeling process, MetaEdit+Modeler provides a modeling environment to create a model using the graphical **DSL**, and generate corresponding artifacts from the model.

Modeling Software Development Kit (**SDK**) for Visual Studio

Modeling and Visualition SDK (**MSDK**) [56] provides tools and templates for building **DSL**s that can be integrated into Visual Studio. It allows the creation of a *meta-model* using built-in meta-modeling language, graphical representation of each component in the meta-model, validation of a model, and generation of code, documents, configuration files and other artifacts from the model.

A *meta-model* together with a graphical notation are defined in terms of domain classes and their relationships to represent concepts of the problem domain. From the *meta-model* a graphical editor and a tree-based model explorer are generated. Although, Modeling **SDK** ensures compliance of a model with the semantics of the *meta-model*, it includes a validation framework to specify additional constraints and validation rules. The ability to generate code and other artifacts from a model are supported using T4 Text Templates [93], which is a mixture of text blocks and control logic. Both model validation and code generator are specified in Visual C#. The implemented graphical **DSL** is integrated with Visual Studio Integrated Development Environment (**IDE**), and can be distributed as a plugin.

3.5.2 Comparing Graphical **DSL** Frameworks

In essence, a framework for implementing CadaML must provide a way to express the concepts and relationships of the modeled domain, and describe specific functionality of the modeling tool. Because all of the described frameworks use a common core set of meta-modeling constructs derived from **UML** class diagrams, they offer similar power to define the *meta-model*. However, each framework is supported by corresponding tool suites that exploit different mechanism to specify the *concrete syntax*. Moreover, some frameworks support special language features such as custom constraint specification, meta-model composition, model-to-model and model-to-text transformation.

Table 3.4 summarizes the modeling capabilities of the described frameworks

based on the requirements related to a meta-modeling language specified in Section 3.1.2 with an additional requirement regarding the availability of each framework.

Table 3.4: Comparing the DSL implementation frameworks

	EMF & GMF	Epsilon	MetaEdit+	MSDK
Support implementation of graphical DSLs (MR1)	+	+	+	+
Support generation of a model editor (MR2)	+	+	+	+
Provide a model validation tool (MR3)	-	+	-	+
Provide a model-to-text transformation tool (MR4)	+	+	+	+
Open source availability of the framework	+	+	-	-

All of the frameworks provide capabilities to implement graphical DSLs (MR1), create a model editor for DSLs (MR2), and to generate code from the model (MR3). On the other hand, only Epsilon and MSDK support defining custom constraints and validation rules in addition to default validation of a model with the *semantics* in the *meta-model* (MR4). Specifying custom constraints is crucial to guarantee the compliance of generated artifacts with the syntax and semantics of the target domain.

EMF & GMF and Epsilon are open source frameworks for Eclipse IDE, while, MetaEdit+ and MSDK are commercial products. MetaEdit+ requires purchasing MetaEdit+ Workbench and Modeler, and MSDK requires Visual Studio Community or other paid versions of Visual Studio. Note that Visual Studio Community has substantial limitations compare to other versions⁸. Furthermore, MetaEdit+ does not provide an integrated development environment to support cloud application implementation and deployment.

As a result of comparing the current frameworks to develop and deploy graphical DSLs, CadaML is designed using the Epsilon framework. The choice of the framework is also supported by the design and implementation preferences that are identified in the design phase (*i.e.*, Section 3.4). In particular, Epsilon is preferred as it provides the following capabilities: (i) defining a *meta-model* in Ecore⁹ meta-modeling language, (ii) producing a model editor from the *meta-model* using the EuGENia¹⁰ tool, (iii) specifying constraints and validation rules in EVL, and (iv) implementation of the model-to-text transformation in EGL.

⁸<https://visualstudio.microsoft.com/vs/compare/>

⁹<https://www.eclipse.org/modeling/emf/>

¹⁰<http://www.eclipse.org/epsilon/doc/eugenia/>

3.5.3 Graphical Editor for CadaML

A graphical editor for CadaML is crucial to provide a modeling environment. Using the editor, cloud data architects and developers should be able to create a model, validate the model and generate data access layer code from the model.

```

@gmf
package databaseDSML;

@gmf.diagram
class DatabaseDiagram{
    attr String projectName;
    val SQLiteDatabase[0..*] sqlDbs;
    val NoSqlDatabase[0..1] noSqlDb;
    val ObjectStorage[0..1] objStorage; }

@gmf.node(label="name, partitioning", label.pattern="{0} ({1})",
tool.name="SQL Database", color="252,252,252")
class SQLiteDatabase{
    attr String name = "SQL Database";
    attr Partitioning partitioning;
    @gmf.compartment
    val SqlTable[1..*] tables;}

enum Partitioning{
    SingleShared;
    SingleSharedMutlipleSchema;
    SeparateDatabasePerTenant;}

@gmf.node(label="name", tool.name="SQL Table", color="242,248,255")
class SqlTable{
    attr String name;
    attr Boolean isPublic = false;
    attr Boolean autoGeneratePrimaryKey = false;
    @gmf.compartment(layout="list")
    val Field[1..*] fields;
    val SqlReference[*] references;}

@gmf.node(label="name, dataType", label.pattern="{0}:{1}")
class Field{
    attr String name = "Name";
    attr String dataType = "Data Type";
    attr Boolean isPrimaryKey = false;}

```

Listing 1: Defining a database diagram and concepts for relational databases in Ecore

An editor for CadaML is generated from the Ecore *meta-model* which is annotated using the EuGENia tool. An excerpt of the *meta-model* that describes a database diagram is presented in Listing 1. The `@gmf` annotation is applied to a package, and it indicates that `GEMF`-related annotations are expected in the package. In the meantime, the database diagram is denoted with the `@gmf.diagram` annotation which represents a root element of the *meta-model*. The database diagram is an

environment in which a modeler constructs a model of a data architecture, and it can contain instances of relational databases, non-relational database and object storage.

Listing 1 also shows that *SqlDatabase*, *SqlTable*, and *Field* are expressed using `@gmf.node` annotation since they represent elements of the model within the database diagram. Thus, these model elements appear on the diagram as nodes. In turn, *SqlDatabase* contains a collection of *SqlTable* instances, where *SqlTable* comprises a list of *Field* instances as a compartment. Lastly, different partitioning schemes for relational databases are composed into *Partitioning* enumeration.

A node can include several parameters to describe graphical styling (*e.g.*, color, shape, label, and size) of the node, and to specify properties for an associated tool (*e.g.*, name, description, and icon) with the node. A label for *SqlDatabase* contains its *name* and *partitioning* attributes that are shown in the editor following the defined label pattern (*i.e.*, *name (partitioning)*). For *SqlTable*, only the *name* attribute is displayed as a label, where for *Field*, both *name* and *dataType* attributes are included in a label that are presented according to the label pattern. Each of the model elements has the corresponding tool name, and distinctive color for intuitive visual differentiation. Note that for *Field* the associated tool will have the same name as the model element because no custom tool name is defined for it.

```
...
@gmf.link(label="name, sourceTable, targetTable", label.pattern="{0} [{1}..{2}]",
source="source", target="target", target.decoration="filledclosedarrow",
style="dot", width="2", tool.name="SQL Relationship", tool.color="0,0,255")
class SqlReference{
    attr String name="Name";
    attr Multiplicity sourceTable;
    attr Multiplicity targetTable;
    ref SqlTable source;
    ref SqlTable target;
    ref Field referenceKey; }

enum Multiplicity{
    Zero;
    One;
    Many; }
...
```

Listing 2: Defining relationships between tables for relational databases in Ecore

SqlReference is described using the `@gmf.link` annotation as shown in Listing 2. Thus, it appears on the diagram as a link that connects two tables (*i.e.*, *source* and *target*). The name of the link and its multiplicity are included in the link label which are displayed as specified in the label pattern. A value for multiplicity is selected from an enumeration which can be either *Zero*, *One* or *Many*. Moreover, the link is represented as a dotted line with filled closed arrow at the end, and it is associated

with `SQL Relationship` tool.

The concepts of the rest storage types and their interrelations are annotated with their configuration parameters following the same principles. When the *meta-model* is complete, the EuGENia tool is used to produce necessary models from it, and to generate a graphical editor for CadaML. The editor consists of three parts as illustrated in Figure 3.4: ① a *canvas* represents *DatabaseDiagram* from the *meta-model* in which a modeler creates model elements, and the relationships that define links between model elements; ② the *Palette* comprises tools associated with the model elements specified in the *meta-model*; and ③ the *Properties* tab that shows properties of each selected model element in the canvas.

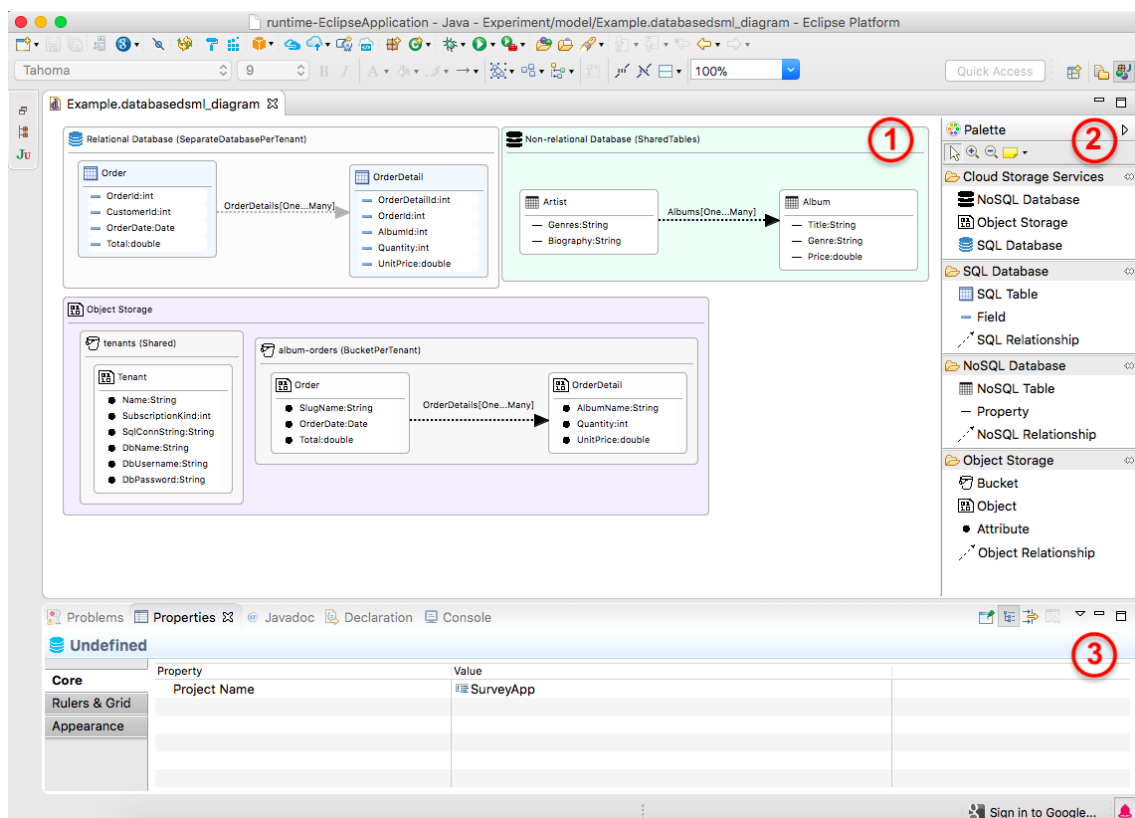


Figure 3.4: The concrete syntax of CadaML implemented as the graphical editor with three parts: ① canvas ② palette, and ③ properties tab.

An illustrative example of a multi-tenant data architecture is depicted in the *canvas*. The data architecture is divided into three separate diagrams that represent different cloud data storage types, and data storage type specific model elements with their relationships are defined in each diagram. Moreover, each data storage type is highlighted with a different color, and its model elements are represented with distinctive graphical notations.

When a model element is selected in the *canvas*, information about it is displayed in the *Properties* tab. Information regarding model elements varies based on the *meta-model*. Specifically, the *Properties* tab includes attributes of a model

element specified in the *meta-model*. For example, *projectName* attribute of the *DatabaseDiagram* is shown in Figure 3.4 as a property.

Tools in the *Palette* are grouped into *Cloud Storage Services*, *SQL Database*, *NoSQL Database*, and *Object Storage* categories. *Cloud Storage Services* category includes three different cloud data storage solutions. While, the rest categories comprise model elements that represent concepts of each data storage solution. In addition, different icons are used to illustrate model elements in each category.

3.5.4 Validation Rules and Constraints

EuGENia allows to specify the *meta-model*, and to generate a graphical editor for CadaML. Nevertheless, there are some subtle constraints need to be specified to ensure consistency of a model that is created using CadaML. For example, no table should have an empty name, and a table name must be a valid identifier. These are examples of custom constraints. Hence, using the graphical editor a modeler can temporarily create a data architecture with constraints violations. However, these violations must be captured and fixed before the modeler saves the model, or produces other artifacts from it. To help with this, Epsilon provides *EVL*.

In CadaML, there are common and data storage type specific validation rules and constraints. The common constraints and validation rules are applicable to all data storage types. The data storage type specific ones are required by each data storage solution. The validation rules and constraints are based on principles of Java programming language as the target representation is the data access layer code in Java, and peculiarities of different cloud data storage types.

As every model element has the *name* attribute, constraints related to this attribute are common for most of the model elements. As an example, Listing 3 illustrates a specification of the common constraints for *SqlTable*. Primarily, the *name* attribute must be defined in a model, and it should be a valid identifier. Thus, a name must be composed of letters, digits and underscore, and it may only begin with a letter. Moreover, the *name* of a model element cannot be repeated within a compartment. Therefore, *table names* must be unique in a relational database, and *fields* must have different names in a relational table.

Apart from the common constraints, each data storage type imposes additional constraints. In the relational and non-relational databases, a table cannot contain a field/property that matches with the name of the table. Likewise, in the object storage, an object name and names of attributes within the object must be different. Furthermore, a relational table must have at least one primary key, whereas both partition key and row key must be defined for a non-relational table.

Once the constraints are specified, they are bound to the graphical editor as an

```

context SqlTable{
  constraint HasName {
    check : self.name.isDefined()
    message: self.name + ' must have a name.'
  }

  constraint NameValidIdentifier{
    guard: self.satisfies('HasName')
    check : self.name.matches("^[_0-9a-zA-Z]+")
    message : self.name + ' name "' + self.name
              + '" may contain letters, digits, and underscore.'
  }

  constraint NameMustStartWithLetter{
    guard: self.satisfies('HasName')
    check : self.name.characterAt(0).matches("[a-zA-Z]+")
    message: self.name + ' name "' + self.name + '" must start with a letter.'
  }

  constraint HasUniqueIdentifier {
    check: SqlTable.allInstances().forall(i|i.name = self.name implies i=self)
    message: 'The ' + self.name + ' must have a unique name.'
  }

  constraint HasUniqueFieldNamesInTable {
    check: self.checkFields()
    message: 'The ' + self.name + ' must have unique field names.'
  }
}

```

Listing 3: An excerpt of the constraints for relational tables in [EVL](#)

extension. When a modeler attempts to save the model in the graphical editor, the constraints are automatically checked for violations. In case of any violation, model elements that does not comply with the validation rules are highlighted with an error mark, and the cause of the violation is displayed in the *Problems* tab underneath the canvas.

3.5.5 Code Generation

The main objectives of CadaML are to increase developer productivity and improve code reliability by (semi-)automating data architecture implementation. To achieve these objectives, CadaML includes a code generator that transforms a multi-tenant data architecture designed by a modeler to executable Java code for Alibaba Cloud, [AWS](#), and Azure. We had to exclude Google from supported cloud service providers due to the lack of implementation commonalities with other cloud service providers. The code generator is implemented using [EGL](#) which is a template-based language that allows to generate any kind of textual artifacts from models. For CadaML, the code generator produces different set of Java interfaces and classes for each data

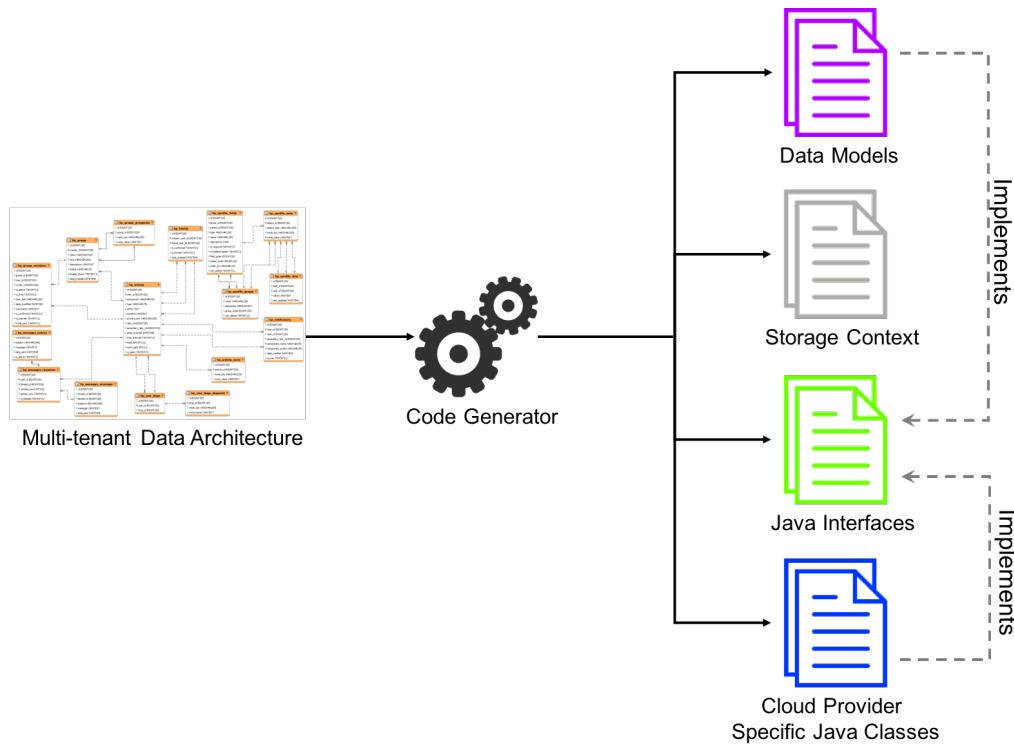


Figure 3.5: CadaML code generation process.

storage solution.

The high level overview of the code generation process is illustrated in Figure 3.5. The code generator tool produces *data models*, *storage context classes*, *Java interfaces* and *cloud provider specific classes*. A *data model* is a Java class that contains fields with corresponding getters and setters for each field. A *storage context class* contains storage related fields, such as provider name, storage credentials, region, and replication to initialize a storage connection. Finally, a *Java interface* contains generic method signatures that are further implemented by *cloud provider specific classes*.

All code that is specific to each storage type are located in different packages. In addition, the generated code decouples the data access logic from other layers of the application. This separation, crucially, provides ease of code maintenance, and allows to independently scale the data layer.

Data Model Generation

Objects and relational tables are directly transformed into data models that are shared among all cloud providers. In contrast, non-relational tables are transformed into Java interfaces, where for each interface cloud provider specific data models are generated. The reason for this is the design of non-relational data models varies depending on a cloud provider, and interfaces provide an abstract representation of

```

@DynamoDBTable(tableName="Artist")
public class Artist implements ArtistInterface {
    private String artistId;
    private String artistName;
    private String genres;
    private String biography;
    private List<AlbumInterface> albums;

    @DynamoDBHashKey(attributeName="ArtistId")
    @DynamoDBAutoGeneratedKey
    public String getArtistId(){ return artistId; }
    public void setArtistId(String artistId){ this.artistId = artistId; }

    @DynamoDBRangeKey(attributeName="ArtistName")
    public String getArtistName(){ return artistName; }
    public void setArtistName(String artistName){ this.artistName = artistName; }

    @DynamoDBAttribute(attributeName = "Genres")
    public String getGenres() { return genres; }
    public void setGenres(String genres) { this.genres = genres; }

    @DynamoDBAttribute(attributeName = "Biography")
    public String getBiography() { return biography; }
    public void setBiography(String biography) { this.biography = biography; }

    @DynamoDBIgnore
    public List<AlbumInterface> getAlbums() { return albums; }
    public void setAlbums(List<AlbumInterface> albums) { this.albums = albums; }
}

```

Listing 4: ‘Artist’ data model for DynamoDB generated by CadaML

different data models.

Data models for object storage are generated as plain Java data models. Whereas, relational and non-relational data models require annotations in order to map fields of a data model to actual attribute names in database tables. Specifically, relational data models are annotated using [JPA](#), where cloud provider specific non-relational data models for Alibaba Cloud and [AWS](#) are denoted using [JPA](#) and DynamoDB Java Annotations, respectively. In the meantime, the generated non-relational data models for Azure extends the base object type provided by Azure Storage Services.

Listing [4](#) presents ‘Artist’ data model that is generated by CadaML for [AWS](#). The data model is annotated using DynamoDB annotations, and it is mapped to ‘Artist’ table in DynamoDB database. The class properties are associated to their corresponding columns in the table, where ‘artistId’ and ‘artistName’ refer to the partition key and row key of the table, respectively. However, the ‘albums’ property is ignored as it references data in another table. Generated data models for Alibaba Clouds^{[11](#)} are annotated in the same manner with explicit mappings, and specification of a composite key.

¹¹The data model for Alibaba Table Store is presented in Appendix [D](#)

```

public class Artist extends TableServiceEntity implements ArtistInterface {
    private String genres;
    private String biography;
    private List<AlbumInterface> albums;

    public String getArtistId() { return this.partitionKey; }
    public void setArtistId(String artistId) { this.partitionKey = artistId; }

    public String getArtistName() { return this.rowKey; }
    public void setArtistName(String artistName) { this.rowKey = artistName; }

    public String getGenres() { return genres; }
    public void setGenres(String genres) { this.genres = genres; }

    public String getBiography() { return biography; }
    public void setBiography(String biography) { this.biography = biography; }

    @Ignore
    public List<AlbumInterface> getAlbums() { return albums; }
    public void setAlbums(List<AlbumInterface> albums) { this.albums = albums; }
}

```

Listing 5: ‘Artist’ data model for Table Storage generated by CadaML

The generated data model for Azure from the same table is shown in Listing 5. Compared to AWS, Azure automatically maps class properties to table columns that require persistence in a table, whilst referenced data are denoted with *@Ignore* annotation. Moreover, there is no need to declare partition key and row key in the data model as they are inherited from the base object type (*i.e.*, ‘TableServiceEntity’).

Storage Context Generation

A storage context class is important to specify configuration information in order to establish a storage connection. In CadaML, a storage context is produced for each data storage solution. Typically, configuration information required by each data storage type differs. In particular, object storage and non-relational databases require cloud provider, region, storage credentials, and replication mode for the storage. On the contrary, a relational database needs database engine, database name, database credentials, hostname and port to initialize a connection to a database instance. In both cases, the storage context class needs data storage type specific implementation to initialize a connection with the provided configuration information.

Java Interfaces and Implementation Classes Generation

For every cloud data storage type, a Java interface and cloud provider specific implementation classes are generated. The former contains generic method signatures to initialize a storage and to perform database related operations. Where, the latter implements these methods for each cloud provider.

For relational databases, an interface comprises abstract methods for storage initialization and *Create, Read, Update, and Delete* (CRUD) operations. In addition to these methods, interfaces for object storage and non-relational databases include creation of a bucket/table and specification of a region where buckets/table will be created.

```
public interface BlobStorage {
    void initializeStorage();
    void createBucket(String bucketName);
    void setRegion(String region, Boolean replication);
    <T> void uploadBlob(T blob, String bucket, String key);
    <T> T getBlob(String bucket, String key, Class clazz);
    <T> List<T> getBlobList(String bucket, Class clazz);
    void deleteBlob(String bucket, String key);
}
```

Listing 6: Object Storage Interface generated by CadaML

Listing 6 presents an interface for object storage that contains all necessary abstract methods, namely, storage initialization, bucket creation, region specification, object upload, retrieval of a single object, retrieval of a list of objects, and deletion of an object. The implementation of storage initialization and object upload for Amazon S3 are shown in Listing 7. The storage initialization method establishes a new Amazon S3 client using AWS credentials to access Amazon S3 in a specified region. In turn, the blob upload method serializes a Java object to a JavaScript Object Notation (JSON), and stores it in a bucket. It is worth noting that cloud provider specific classes implement methods in a generic way that work on different data models.

3.6 Reflection on Requirements

The concepts and notations included in CadaML are obtained through analyzing different cloud data storage solutions offered by widely used cloud providers, investigating current cloud modeling languages that allow developing and deploying cloud applications, and exploring available data partitioning patterns.

Since analyzed cloud providers and modeling languages share a common vocabulary to describe relational databases, the existing terminology is reused in CadaML to represent a relational database and its components. In contrast, each cloud provider uses different concepts to describe non-relational databases and blob storage solutions. Moreover, some cloud providers exploit the same concepts that are used in relational databases to define particular elements of non-relational databases. For CadaML, in order to provide explicitness and eliminate ambiguity, concepts that are only specific to non-relational databases are chosen to depict elements of a non-

```

public class BlobStorageAmazonImpl implements BlobStorage {
    private AmazonS3 client;
    private String accessKey;
    private String secretKey;
    private Regions region;
    ...
    public void initializeStorage() {
        BasicAWSCredentials awsCreds =
            new BasicAWSCredentials(accessKey, secretKey);
        client = AmazonS3ClientBuilder
            .standard()
            .withCredentials(new AWSStaticCredentialsProvider(awsCreds))
            .withRegion(region)
            .build();
    }

    public <T> void uploadBlob(T blob, String bucket,
                               String key) {
        ObjectMapper mapper = new ObjectMapper();
        String jsonInString = mapper.writeValueAsString(blob);
        byte[] content = jsonInString.getBytes();
        ByteArrayInputStream contentsAsStream =
            new ByteArrayInputStream(content);
        ObjectMetadata md = new ObjectMetadata();
        md.setContentLength(content.length);
        client.putObject(new PutObjectRequest(bucket, key, content, md));
    }
    ...
}

```

Listing 7: The code generate by CadaML that implements storage initialization and object upload methods for the Amazon [S3](#) object storage service.

relational database. Whilst, most commonly used terminology among different cloud providers is captured to describe a blob storage. Hence, CR1 and CR2 requirements are supported by reusing existing terminology and providing invariance of concepts to represent different cloud data storage types and their components.

Once the concepts are defined, the implementation of CadaML is achieved by leveraging tools and languages offered by the Epsilon framework. Specifically, the EuGENia tool is used to design the CadaML meta-model, and to generate an editor from the meta-model. Thus, MR1 and MR2 requirements are satisfied as Epsilon effectively supports implementation of CadaML as a graphical modeling language, and it reduces the development effort to produce a modeling environment. Epsilon also provides [EVL](#) to define and evaluate custom constraints on models. This, in turn, certainly fulfills MR3 requirement by enabling validation of custom constraints in addition to ensuring compliance of a model with the semantics of the meta-model. In the meantime, MR4 requirement is achieved by generating the data access layer code from a model using [EGL](#). This also supports CR3 requirement as a model contains all the necessary information to produce the sufficient data access layer

code.

3.7 Summary

This chapter introduced **CadaML**, a modeling language that enables designing a multi-tenant data architecture of cloud applications as a combination of different cloud data storage solutions in a cloud provider agnostic manner. **CadaML** also provides the model validation to keep a data architecture consistent with the semantics of the **CadaML** meta-model and additional custom constraints. Moreover, it produces data access layer code from a data architecture that are executable on three major cloud service providers.

In order to design and implement **CadaML**, first of all, requirements related to **CadaML** concepts, and a meta-modeling language to develop **CadaML** are specified. These requirements are crucial to support design and evaluation of **CadaML**, and to alleviate the complexity and risk to the **CadaML** development.

Then, the **CadaML** development methodology is described which includes domain analysis, design and implementation phases. During the domain analysis phase, different cloud data storage types, offered by four widely used cloud service providers, with their partitioning schemes, and characteristics of existing cloud application modeling languages are explored to formulate a domain model. The domain model presents unified concepts and terminology to represent different data services in an abstract way. While in the design phase, the relationship of **CadaML** to current cloud application modeling languages is identified, and the design specification for **CadaML** is defined. As an outcome of the design phase, the **CadaML** meta-model is derived from the domain model. Following the design phase, frameworks to develop and deploy **DSLs** are analyzed and compared in the implementation phase to select a suitable one for **CadaML**. With the chosen framework, a modeling environment is produced from the meta-model, validation rules and constraints are specified, and code generation is implemented.

Finally, the implementation decisions are validated against the requirements regarding the concepts of **CadaML**, and the meta-modeling language exploited to deliver **CadaML**.

Chapter 4

Application & Qualitative Evaluation of CadaML

This chapter describes the exploitation of CadaML and presents our qualitative evaluation of the modeling language through a case study. The exploitation explains the process of evolution from single- to multi-tenancy using the modeling environment of CadaML. It also demonstrates the capability of the modeling language to design different multi-tenancy patterns at the abstract level, and generate the corresponding data access layer code. As a case study, an industrial business process analyzing web application is evolved from single-tenant on-premises to a multi-tenant service deployed to a public cloud. During the case study, we evaluate the feasibility of the language and the adequacy of multi-tenancy implementation by the generated code. The feasibility is examined through qualitative evaluation methods, while the adequacy of multi-tenancy implementation is assessed by combining manual code reviewing and automated unit testing approaches.

The chapter starts with a comparison of the evolution process from single- to multi-tenancy using the manual approach and the modeling environment of CadaML in Section [4.1](#). Then, Section [4.2](#) presents the experimental use case, and describes its evolution motivation along with challenges. Section [4.3](#) follows these with the comparison of the existing data partitioning schemes regarding the requirements of the use case application. This section also presents the implementation of the selected data partitioning pattern using CadaML and re-architecting the application structure. The evaluation methodology and results of applying CadaML are interpreted in Section [4.4](#). While Section [4.5](#) discusses reflection on evolution challenges, and comments on the limitations of the performed case study. Finally, Section [4.6](#) provides conclusion of the chapter.

4.1 Evolving from Single- to Multi-tenancy

Multi-tenancy is an attractive pattern for efficiently utilizing cloud resources by sharing them across multiple tenants. Hence, applications built using this pattern can be offered at a lower price, and reduce maintenance effort as less application instances and supporting cloud resources must be maintained (*e.g.*, [17, 28, 32]). However, evolving a single-tenant application to a multi-tenant cloud service needs re-engineering all layers of an application, and the data layer is no exception. In this context, evolution refers to the modification of the application to reflect multi-tenancy requirements. Currently, introducing multi-tenancy to existing single-tenant applications is predominantly achieved using manual approaches (*e.g.*, [17, 26, 32, 76]). Nevertheless, manual implementation is usually time-consuming and error prone (*e.g.*, [23, 45]). In this section, we describe the manual implementation of multi-tenancy at the data layer, and compare it against implementing with CadaML.

4.1.1 Manual Evolution

A traditional manual implementation process based on the software development life-cycle [120] covers the following steps as illustrated in Figure 4.1a: (i) data layer requirements are gathered and captured in a requirement specification document; (ii) the requirements are analyzed into models, schemes and business rules; (iii) a data architecture is typically designed in a form of entity relationship diagram; (iv) developers implement a data access layer from the data architecture model; and (v) developers systematically discover and debug errors in the code.

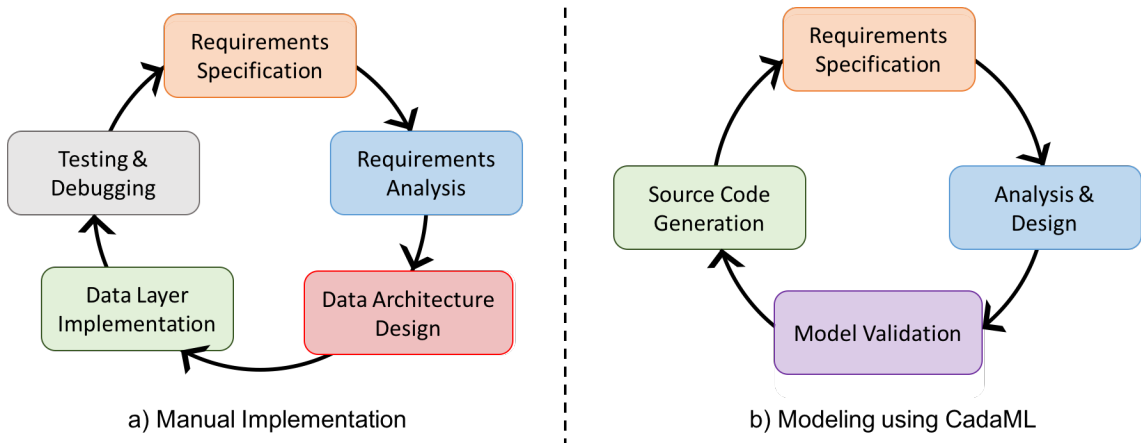


Figure 4.1: Comparing the implementation processes of the manual approach and that of CadaML.

During the data layer implementation, developers implement a collection of classes and interfaces with their methods and properties. The implementation should provide features to connect to the database, establish and terminate connections, and

perform **CRUD** operations. Moreover, the implementation must be cloud provider independent to offer the flexibility of exploiting multiple cloud providers as data storage services, or to enable quick and easy switching from services of one cloud provider to another when needed.

In this manual approach, whenever the data layer requirements change, developers have to go through all the subsequent steps and, eventually, modify the existing code. For example, a multi-tenant data architecture can be implemented by either sharing a single database by all tenants, deploying a separate scheme for each tenant in a shared database, or deploying a separate database instance per tenant. Initially, developers might implement a single shared database for all tenants. Later, security requirements of tenants may require a more isolated approach that cannot be provided in a single database instance. Thus, developers need to consider these requirements, change the data architecture model, modify the data access layer code, and verify the changes.

Although the data architecture model is validated at the design level against requirement specifications, transformation from data architecture models to implementations is generally performed in ad-hoc manner without any formal guidelines or process. As such, some important implementation actions may be neglected since application implementation process usually changes as the implementation progresses. This will negatively affect the quality of the application. Moreover, this type of transformation neither ensures the correctness of the implementation nor guarantees the reflection of requirements in the implemented code [4]. Therefore, manual implementation tends to be time-consuming and error prone (*e.g.*, [23,45]).

4.1.2 Modeling using CadaML

In order to mitigate such design and implementation processes, model-driven approaches have been successfully applied both in academia and industry for building service-oriented applications, developing autonomic enterprise applications, and automating industrial management processes (*e.g.*, [10,39,53,70,133]), but not for implementing multi-tenant cloud applications. These approaches have resulted in reduced effort on development, increased productivity of developers, improved quality and maintainability of the application. Inspired by this, we propose CadaML to enable describing a data architecture in an abstract level by hiding the implementation details of the underlying storage type. As shown in Figure 4.1b, a data layer implementation workflow using CadaML involves four steps: (i) first, as in the manual approach, data layer requirements are gathered; (ii) the requirements are analyzed and a data architecture model is designed using the graphical editor of CadaML; (iii) the model is validated for constraints and validation rules imposed by

CadaML; and (iv) the data access layer source code is produced from the model.

```

...
public SQLiteDatabaseImpl(String className, String jdbcURL, String schema) {
    this.className = className;
    this.URL = jdbcURL;
    this.schema = schema;
}

public void initializeDatabase() {
    try {
        Class.forName(className);
        System.out.println("Initializing SQL connection...");
        conn = DriverManager.getConnection(URL);
        conn.setSchema(schema);
    }
    catch (ClassNotFoundException e) { ... }
    catch (SQLException e) { ... }
}
...

```

Listing 8: The generated Java code by CadaML for establishing a database connection for a shared database with separate schemas

When using the modeling environment of CadaML, developers design a multi-tenant data architecture in terms of tables and their interrelations. Since CadaML captures different data partitioning patterns as configuration options for each available cloud data storage type, developers can specify a suitable data partitioning scheme at the abstract level. In this scenario, changes in the requirements can be directly reflected in the model by selecting an appropriate data partitioning option. Compared to the manual approach, CadaML automates the data layer implementation by producing a corresponding data access layer code from the data architecture model. Specifically, CadaML produces different code for each data partitioning pattern. For both separate databases and a shared database with separate schemas approaches, the queries do not require filtering as the data access layer enables isolation of tenant data by connecting to a tenant-specific database and a tenant-specific schema, respectively. Listing 8 presents a code excerpt produced by CadaML that implements establishing a connection to a SQL database and specifying a tenant-specific schema. The constructor accepts three parameters to initialize a class name for a database engine, a database connection Uniform Resource Locator (URL), and a schema name, where the last is used to set a tenant-specific schema when creating a database connection. This information is typically stored as part of the tenant configuration.

In contrast, when tenants share a schema in a shared database, filtering queries to perform CRUD operations is mandatory to provide logical isolation of tenant data. CadaML implements filtering in a generic manner. For example, Listing 9 presents a code fragment that filters retrieval queries with the WHERE clause. This method can

```

...
public <T, F> T selectList(String table, String fkName, F fk, Class clazz) {
    List<T> list = new ArrayList<>();
    String selectQuery = null;

    if (fk instanceof String)
        selectQuery = String.format("SELECT * FROM %s WHERE %s = '%s'",
            table, fkName, fk);
    else
        selectQuery = String.format("SELECT * FROM %s WHERE %s = %d",
            table, fkName, fk);

    try {
        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery(selectQuery);
        ...
    } catch (SQLException e) {...}

    return (T) list;
}
...

```

Listing 9: A generic method generated by CadaML for retrieving data from a shared database with a shared schema

be used to retrieve data from any table filtered by any field. This implementation is based on industrial and academic work [17, 32, 58] that proved to be efficient in enabling tenant isolation.

In addition to the capability of producing corresponding data access layer code for different data partitioning patterns, the generated code is cloud provider independent. More concretely, a connection to a relational database is implemented using a `JDBC` `API` driver which is supported by almost all cloud service providers. As a result, the generated code only requires a database `URL` that consists of the server name hosting the database, port number, database name, database user, and password. For non-relational databases and blob storage, the generated code is implemented by exploiting the concept of polymorphism to dynamically interact with an appropriate data storage service of the three major cloud service providers (*i.e.*, Alibaba Cloud, `AWS`, and Azure) based on the configuration information.

Listing 10 demonstrates the creation of a connection to a blob storage service hosted by a cloud provider. In the code fragment, `blobStorage` is an instance of an interface that is initialized to an implementation of a particular cloud provider at run-time. Hence, this instance is used to perform data operations over any blob storage. In the same manner, the generated code implements methods to interact with non-relational databases.


```

...
public BlobStorage getBlobStorage() {
    if (cloudProvider.equalsIgnoreCase("aws")
        || cloudProvider.equalsIgnoreCase("amazon")) {
        blobStorage = new BlobStorageAmazonImpl(identity, credential);
        blobStorage.setRegion(region, replication);
    } else if (cloudProvider.equalsIgnoreCase("azure")) {
        blobStorage = new BlobStorageAzureImpl(identity, credential);
    } else if (cloudProvider.equalsIgnoreCase("alibaba")) {
        blobStorage = new BlobStorageAlibabaImpl(identity, credential);
        blobStorage.setRegion(region, replication);
    }
    return blobStorage;
}
...

```

Listing 10: A method generated by CadaML for initializing a connection to an appropriate blob storage service based on a cloud provider

4.2 Industrial Case Study: Background

To investigate the practical feasibility and evaluate the utility of applying CadaML, a case study has been conducted. As an experimental use case, an industrial web application is evolved to introduce multi-tenancy, and to deploy the application in a cloud environment. The application is owned by a research center of a major international telecommunication provider operating in 150+ countries (name of the research center is redacted). The aims of the research center are to re-architect the application as a multi-tenant cloud service, centralize the management of the application, and reduce the associated development and maintenance effort.

The application is distributed to many subsidiaries (hereafter, tenants) of a holding company, and the purpose of the application is to ensure compliance of business processes of each subsidiary with the policies imposed by the holding company.

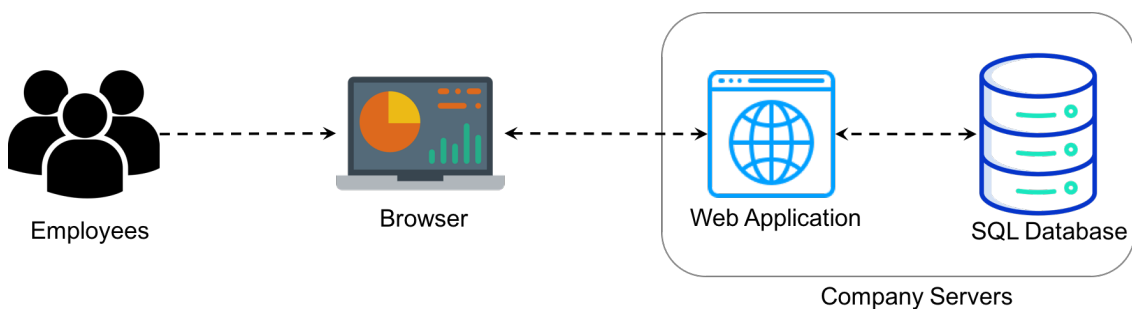


Figure 4.2: The business process application architecture.

4.2.1 Overview

A high-level view of the original application architecture is illustrated in Figure 4.2. The architecture is straightforward, and is one that many other applications use: the application is a three-tier Java web service with an SQL database (Oracle in this case) for data storage. The application with its components are deployed to the servers of each tenant, allowing tenant employees to interact with it over the Intranet using a browser.

The application itself is developed using Google Web Toolkit (GWT)¹, and it consists of the presentation layer, business logic layer, and data layer. This separation helps to manage complexity during development and enable loose coupling between the application layers.

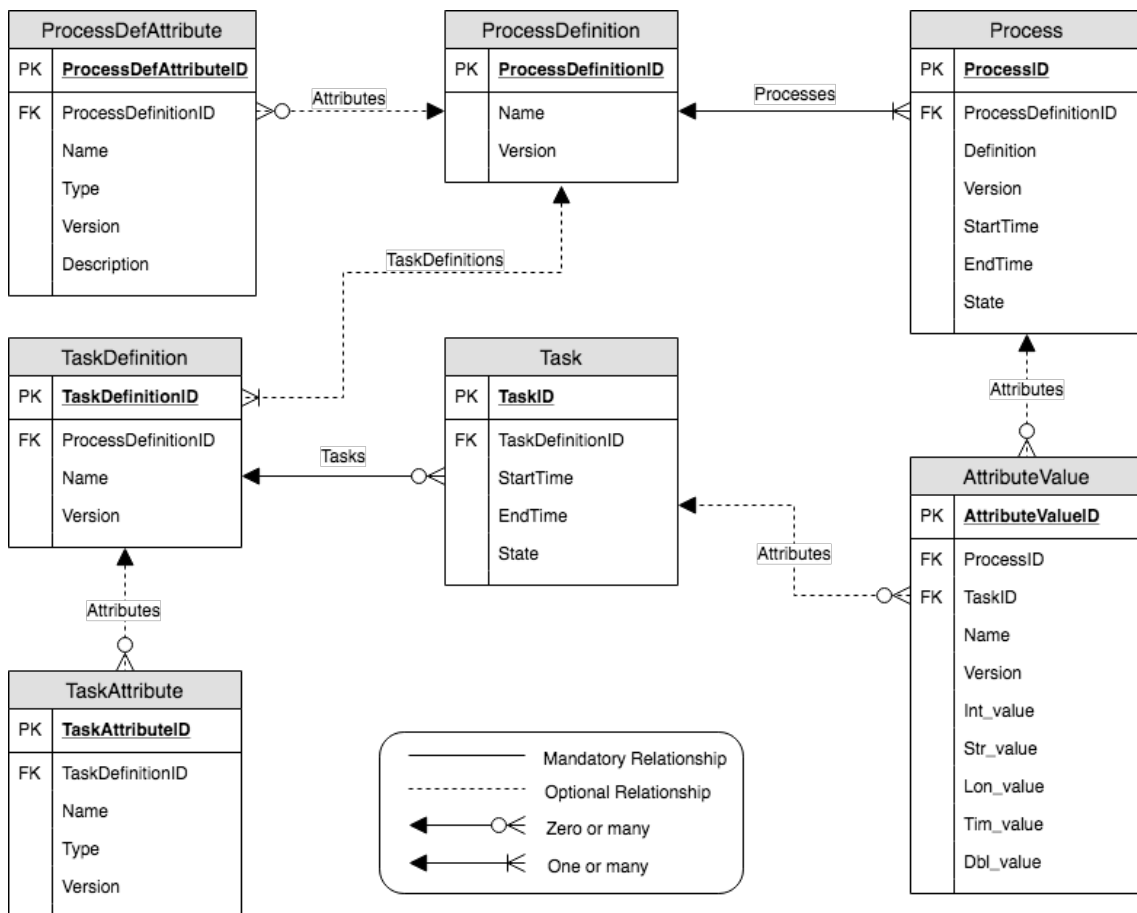


Figure 4.3: The ER diagram of the business process application.

The actual data architecture of the application is designed for a relational database that contains 18 entities with their interrelations. For the experiment, only a set of core entities is used to demonstrate the feasibility of CadaML. An ER diagram of the experimental data architecture is presented in Figure 4.3.

¹<http://www.gwtproject.org/>

The *process definition* entity defines a business process, and it comprises *process* and *task definition* entities. The *process* entity describes a job, order, or process execution, such as service fulfillment or fault repair process, where the *task definition* defines a description for a *task* in a business process. The remaining entities, namely, *process attribute*, *task attribute*, and *attribute value*, hold additional attributes to provide extensibility of the data architecture.

Currently, for each tenant an application and a database instance are deployed on tenant premises. Tenants regularly upload log files of business processes to the application, the application generates reports from the uploaded files, and at the end of each month tenants send reports to the holding company. The reports are analyzed by the holding company for conformance to its business regulations.

4.2.2 Evolution Motivation & Challenges

For the holding company, provisioning and deploying a new tenant requires preparation of the deployment infrastructure, configuration of networks, installation of all necessary software, and ensuring proper functioning of the infrastructure. These processes are time consuming and labor-intensive. Moreover, maintenance of multiple applications with their supporting software and hardware infrastructure require additional effort, and most of the provisioned resources are underutilized. All these factors also incur additional costs.

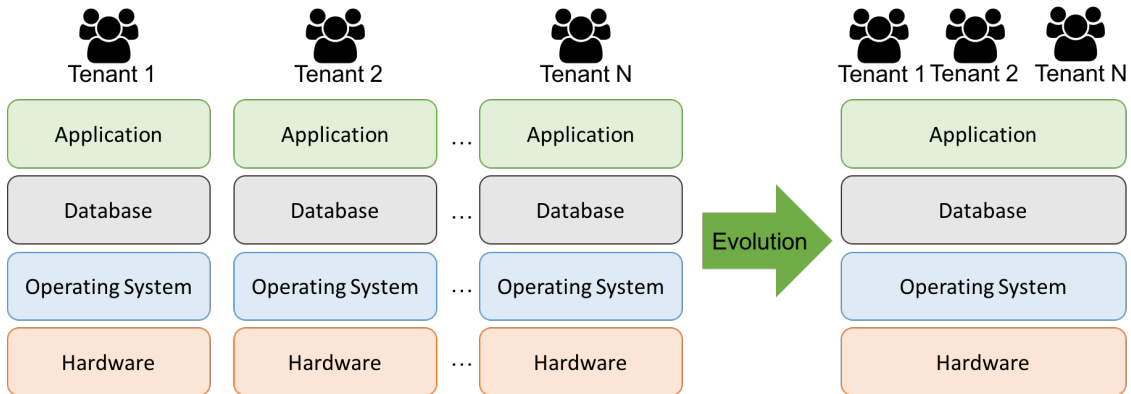


Figure 4.4: The evolution of the use case application from single- to multi-tenancy.

The holding company wants to change this allocation and focus on the application rather than on the infrastructure, hence, it decided to evolve the application into a multi-tenant cloud service as illustrated in Figure [4.4](#). In multi-tenancy, tenants can subscribe to the application, upload log files, generate reports, and arrange the analysis with the holding company without the need to send their reports. Adopting multi-tenancy and deploying the application and database to a public cloud can also bring advantages of economies of scale, promote the centralized control of

its application and database, and automate processes for managing the application resources.

Nevertheless, there are several challenges that the holding company needs to address when introducing multi-tenancy. These challenges affect all layers of the application structure, and, especially the data layer as other layers are typically stateless in cloud applications [11, 38]. In particular, the following multi-tenancy concerns along with design factors influence the application design:

CH1: *Configurability and extensibility:* Before introducing multi-tenancy, every tenant has its own, possibly customized, database instance. In multi-tenancy, the holding company may consider either to deploy each tenant to a dedicated database instance, or to deploy a single database instance for all tenants. For the single shared approach, tenants must be able to configure and extend the database to cater for their specific needs.

CH2: *Tenant isolation:* When sharing a database, one of the highest priorities is to ensure that tenants only view and edit their own data. This requirement must also guarantee that tenant-specific configurations and extensions do not directly affect the data layer for other tenants.

CH3: *Scalability:* The database workload varies when employees of multiple tenants interact with a multi-tenant database instance. Thus, the data layer should be able to horizontally scale as the workload changes. During horizontal scaling, database resources are created or released to match database performance requirements [59].

CH4: *Ease of development and maintenance:* The holding company is concerned that introducing multi-tenancy may increase the complexity in development and maintenance processes. This, in turn, may lead to increase in the application cost.

4.3 Industrial Case Study: Implementation

During the adoption of multi-tenancy, several modifications are required to address the challenges described in the previous section. In essence, the evolution process requires re-architecting the database schema and re-designing the application structure. Primarily, we compare benefits and drawbacks of the current data partitioning options for relational databases. Then, we evolve the data architecture of the use case based on the most suitable data partitioning scheme. Subsequently, we re-architect the application structure to make the solution scalable.

4.3.1 Comparing the Data Partitioning Schemes

Providing tenant isolation, extensibility, and scalability of the data layer requires a partition scheme. The following three approaches have been proposed for data segmentation in relational databases (see Section 3.3.2): (i) separate databases where each tenant is deployed to a dedicated database instance; (ii) a shared database with separate schemas for each tenant; and (iii) a shared database with a shared schema. For the use case application, pros and cons of these three approaches are considered regarding tenant isolation, customizability, development and maintenance effort, and are now discussed.

A separate database per tenant is the simplest approach to ensure isolation and customizability at the data layer. Although this approach does not incur changes to the existing database schema, an additional database is needed to store tenant configuration information. This can be solved by applying an external configuration store pattern [59], where the configuration information are stored in a separate storage. The application reads configuration settings from the external storage, and associates each tenant with its database instance. However, this approach does not solve the concerns of the holding company, as it leads to higher maintenance effort to manage and support multiple database instances for each tenant.

On the contrary, in the shared database with multiple schemas pattern, each tenant is hosted with its own separate set of tables in a single database. A tenant-specific schema is created when a tenant is first deployed to the application. This ensures isolation and customizability at the schema level. Similarly, in the separate databases approach, additional tables are needed to store tenant configuration information to map each tenant with a correct schema. This pattern is relatively easy to implement, but it also requires additional effort to manage and maintain multiple schemas in the database.

Finally, tenants share a database and a set of tables to store their data in the last approach. An identifier, commonly *TenantID*, is used to associate rows in the tables with a specific tenant. Among three patterns, the shared database approach has the lowest costs as it allows to deploy more tenants per database server [29]. A significant drawback of this approach is that additional development effort is required to ensure tenant isolation at the application level.

Through analyzing and discussing the three approaches described above, the research center decided to model the data architecture of the use case application following the single shared database with a shared schema approach for two main reasons. First, this partitioning approach can serve more tenants with a small number of servers. Secondly, it offers low costs and more ease of management as opposed to other partitioning schemes. As a result, data of tenants are combined

into a single database instance.

4.3.2 Evolving the Data Architecture

In this section, we describe how to enable multi-tenancy in a single-tenant data architecture using CadaML. During the evolution, the existing data model is reused with minor modifications to address the evolution challenges.

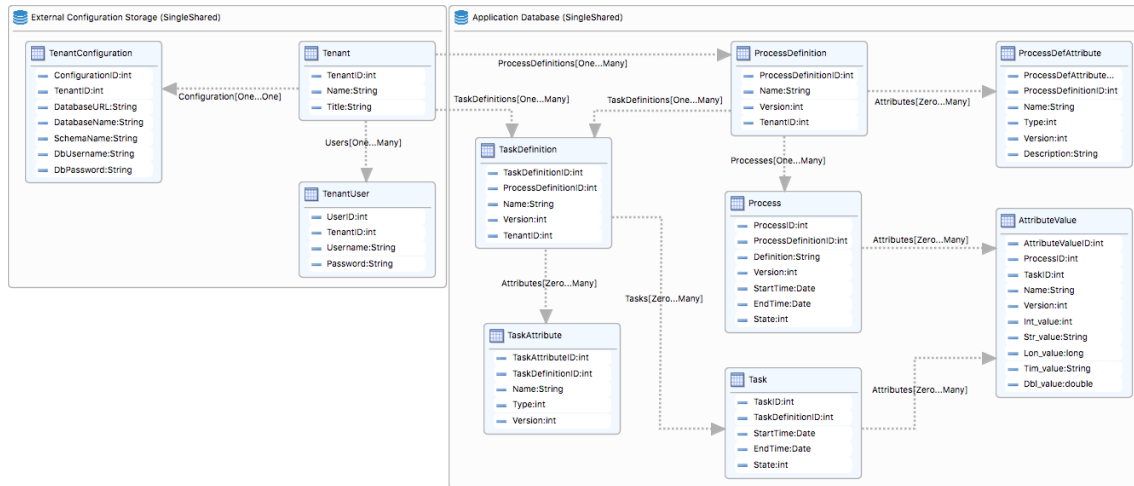


Figure 4.5: The evolved multi-tenant data architecture modeled in CadaML.

The evolved data architecture of the use case is depicted in Figure 4.5. Firstly, a relational database model for configuration data is designed. In this model, *tenant*, *tenant configuration*, and *tenant user* entities are created using *SQL* tables. The *tenant* and *tenant configuration* entities describe a tenant who has subscribed to the application with its configuration information. The *tenant user* entity is introduced to represent an employee of a tenant. Both *tenant configuration* and *tenant user* tables contain the tenant identifier (*i.e.*, ‘*TenantID*’) as the foreign key in order to map a tenant with its configuration data and employees. In addition, the partitioning scheme for this storage is specified as ‘*Single Shared*’ since the external storage is shared across all tenants.

The application data presented in the *ER* diagram (see Figure 4.3) is designed in another relational database model with ‘*Single Shared*’ data partitioning scheme. All entities are modeled as *SQL* tables with collections of *fields*. In a shared database, each tenant has its own set of processes and tasks with their definitions. Thus, the *tenant* entity has relationships with *process definition* and *task definition* entities. Because all tenants share one database, a straightforward solution to ensure logical isolation will be including the tenant identifier to all entities. However, to prevent unnecessary duplication of data, the tenant identifier is only added to *process definition* and *task definition* tables as data from other tables are retrieved based on the primary keys of these two tables.

After this, the model has been checked for violation of constraints by means of the validation tool. Then, the data access layer code in Java is produced from the model. Although, CadaML generates source code for Alibaba, [AWS](#), and Azure, the application code for [AWS](#) is exploited to deploy the given use case. Specifically, CadaML produced data models for each tables and implementation of [CRUD](#) operations for all tables.

4.3.3 Evolving the Application Architecture

In all three data partitioning schemes for relational databases, additional tables are required to store tenant-specific configuration information. Therefore, the application architecture is evolved based on the external configuration store pattern [\[59\]](#). This pattern provides easier management and control of configuration data, and it enables sharing configuration data across other applications and application instances.

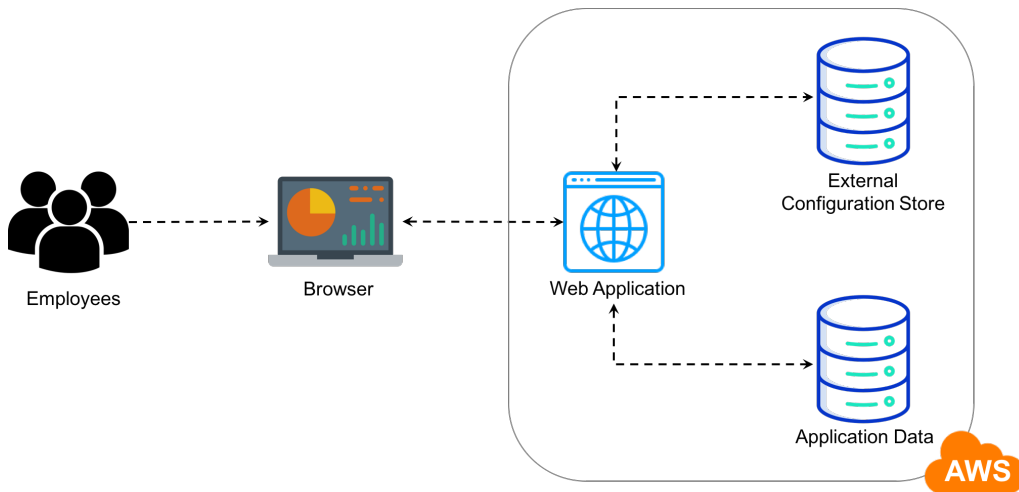


Figure 4.6: The first deployment scenario: the evolved architecture of the business process application that is deployed to services of [AWS](#).

We come up with two deployment scenarios for the holding company to host its application and databases. The first scenario is depicted in Figure [4.6](#) where both application and database instances are hosted in services of [AWS](#). More concretely, an instance of Elastic Compute Cloud ([EC2](#)) is created to deploy an application, while two instances of Amazon [RDS](#) for Oracle are launched for data store (*i.e.*, first instance for configuration data and second instance for application data). This type of deployment is preferred as both [EC2](#) and Amazon [RDS](#) provide dynamic scalability to maintain with changing demands of the application and database.

As shown in Figure [4.6](#), the configuration information of tenants are deployed to a centralized external storage following the external configuration store pattern.

When a tenant employee interacts with the application, configuration data of the corresponding tenant are retrieved from the external storage. According to the configuration data, the application loads tenant-specific user interface, and maps tenant employees to their corresponding fields in the shared database.

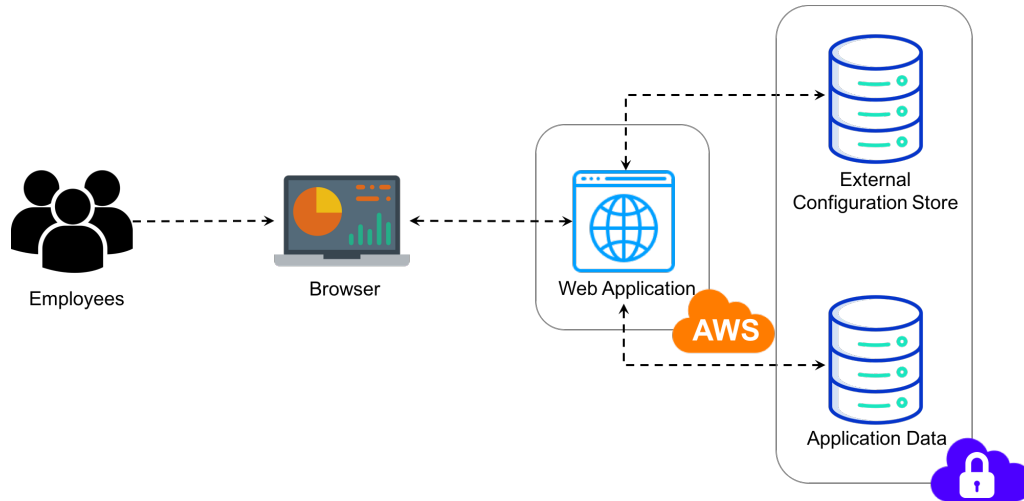


Figure 4.7: The second deployment scenario: the evolved architecture of the business process application where the application is deployed to services of [AWS](#) and the databases are deployed to a private cloud of the holding company.

In the second scenario, as is illustrated in Figure [4.7](#), the application instance is deployed to [AWS](#), while the database instances are hosted in the private servers of the holding company due to the concerns of the holding company to surrender the control of its tenant data to a public cloud service provider. The application works as in the previous scenario except it needs to connect to database instances that are running on-premises of the holding company. Although this scenario meets the security requirements of the holding company, it requires implementation of the scalability of the data layer.

For this use case, **CadaML** provides a significant advantage as it produces the data access layer code that supports both deployment scenarios regardless of the service provider. Compared to the manual approach, **CadaML** removes the need to change the code in order to cater for different deployment specifications or to correctly interact with services of different providers as long as the configuration data is provided correctly.

The running prototypes of both evolved application architectures have been demonstrated to the research center for consideration. The main concerns of the research center was that the holding company was reluctant to deploy its databases to the services of [AWS](#) due to their data privacy and security regulations. They were also worried about the deployment region of their application and databases as [AWS](#) does not support the region where the holding company resides. After few

meetings and discussions, the research center approved the proposed prototypes in November 2017 in order to evaluate the potential benefits of moving the application and databases to services of a public cloud provider.

4.4 Qualitative Evaluation

The case study involved modeling the data architecture of the use case application, validation of the model for constraints and validation rules imposed by the language, and generation of the data access layer code from the validated model. After evolving the application, we analyze the utility of applying CAdAML. Specifically, through this case study we aim to characterize and interpret the application of CAdAML from the following perspectives since there is no [DSI](#) to compare against that also allows modeling multi-tenant data architectures.

- *Evaluation of the feasibility:* When exploiting CAdAML in real-world contexts and settings, we expect that the modeling language improves the productivity of developers, reduces the development effort and provides an appropriate level of usability. However, the effort of applying the modeling language should be reasonable in terms of time to familiarize with the language, convenience of using the modeling environment, and suitability to implement multi-tenancy at the data layer.
- *Evaluation of the adequacy of multi-tenancy implementation:* Although CAdAML (semi-)automates multi-tenancy implementation at the data layer by means of the code generation, we must demonstrate that the generated code implements tenant-isolation and extensibility correctly.

The evaluation was conducted in collaboration with a representative of the research center who has experience in enabling configurability of multi-tenant applications using model-driven engineering approaches, in October 2017.

4.4.1 Evaluation Methodology

The case study was organized based on the guidelines for conducting and reporting case research in software engineering [\[108\]](#), and it involved (i) planning and executing the case study; (ii) collecting and analyzing data; and (iii) reporting the results.

In the planning and executing phase, we gathered goals and concerns of the company, and analyzed the data layer of the application that are presented in previous sections. Then, we compared partitioning schemes for relational databases to identify the most appropriate one that supports the evolution motivation of the

holding company. Based on the selected partitioning scheme, the data architecture was designed using CADA_{ML} with tenant-isolation, configurability and extensibility requirements captured in the model. Finally, the data architecture model was validated, and the data access layer code was produced from the model.

Following the planning and conducting phase, we collected data to evaluate and discuss the application of CADA_{ML}, and adequacy of multi-tenancy implementation by the generated code. The feasibility was assessed through a qualitative evaluation where we analyzed CADA_{ML}, its graphical editor and code generation capability. The questions that are considered to evaluate the feasibility are:

FQ1: Is CADA_{ML} easily usable by the intended domain experts?

- (a) Is the user interface of CADA_{ML} intuitive to exploit?
- (b) Are the tools provided by CADA_{ML} descriptive?

FQ2: Does CADA_{ML} provide an appropriate level of abstraction and notations for building the data layer of multi-tenant cloud applications?

- (a) Does the model created using CADA_{ML} hide implementation details of the underlying storage type?
- (b) Are the notations used in the model expressive and self-explanatory enough?

FQ3: Does CADA_{ML} serve the purpose of the development such as generating relevant artifacts?

- (a) Does CADA_{ML} generate appropriate code that represents business entities of the data architecture?
- (b) Does CADA_{ML} generate appropriate code to connect to the database?
- (c) Does CADA_{ML} generate appropriate code to perform **CRUD** operations in the application?
- (d) Is the generated code provider independent?

FQ4: Does CADA_{ML} appropriately capture and implement multi-tenancy at the data layer?

- (a) Does CADA_{ML} enable modeling tenant isolation?
- (b) Does CADA_{ML} generate code that ensures tenant isolation?
- (c) Does CADA_{ML} enable modeling extensibility?
- (d) Does CADA_{ML} generate code that ensures management of extensions?

In the meantime, the adequacy of multi-tenancy implementation was investigated based on code reviewing [2,3] and unit testing techniques. In code reviewing, we manually inspected the generated code to reduce application defects and improve the quality of the code. While in unit testing, we run several test cases on the method level using JUnit framework². We inspected whether data operations are correctly performed when employees interact with the application. To ensure this, we added three different tenants with several employees, generated business processes and tasks with definitions for each tenant, and stored them in the database.

Once we set up the database, we specified the following test cases to examine methods of each entity in the data architecture. The test cases are grouped based on **CRUD** operations and marked with a related concern under testing. We identified three testing concerns which are operations that require isolation of tenants (I), operations to manipulate extensions (E), and operations to manipulate referencing data (R).

TC1: Viewing tenant-specific data.

- (a) Retrieve an item from a shared table (I).
- (b) Retrieve an item with extensions from shared tables (I, E).
- (c) Retrieve an item with referencing items from shared tables (I, R).
- (d) Retrieve a list of items from a shared table (I).

TC2: Storing tenant-specific data.

- (a) Store a single item in a shared table (I).
- (b) Store a single item with extensions in shared tables (I, E).
- (c) Store a single item with referencing items in shared tables (I, R).
- (d) Store a list of items in a shared table (I).

TC3: Updating tenant-specific data.

- (a) Update a single item in a shared table (I).
- (b) Update a single item and extensions in shared tables (I, E).
- (c) Update a single item and referencing items in shared tables (I, R).
- (d) Update a list of items in a shared table (I).

TC4: Deleting tenant-specific data.

²<https://junit.org/junit5/>

- (a) Delete a single item from a shared table (I).
- (b) Delete a single item with extensions from shared tables (I, E).
- (c) Delete a single item with referencing items from shared tables (I, R).
- (d) Delete a list of items from a shared table (I).

4.4.2 Evaluation of the Feasibility

CadaML has a graphical user interface and provides click and create feature to model a data architecture which makes it simple and convenient to use (FQ1a). In addition, it offers descriptive tools and notations since the current concepts of available cloud storage solutions are reused to represent them (FQ1b). Meanwhile, the expressiveness of the model is validated by mapping the notations from the model to the source code that interacts with the actual storage type (FQ2b). CadaML also allows modeling the data architecture in an abstract way, thus, it only represents the essential elements of the data layer in terms of tables and their interrelations (FQ2a). This enabled us to focus on the data layer rather than on implementation details. These factors conclude that the user interface of the modeling language is simple and convenient to exploit, the tools and notations are descriptive, and the modeled data architecture is self-explanatory and expressive. Therefore, we can claim that CadaML is easily usable by intended domain experts (FQ1) and provides an appropriate level of abstraction and notations (FQ2).

When modeling the data architecture, we were able to specify the data partitioning option through the user interface of CadaML. As we decided to deploy all tenants in a shared database with a shared schema, we included the tenant-identifier to logically isolate rows of each tenant in shared tables in both databases (FQ4a). In order to enable extensibility of the solution, we used additional tables to hold custom data (FQ4c). After we modeled the data architecture, we validated it using the model validation tool provided by CadaML. As the final step, we used the code generation capability of the language to produce the data access layer code.

The code generator, firstly, transformed all tables into data models. Then, it produced a class to establish and manage a connection to the database. The class is implemented in a generic manner, hence, it can interact with any database engine running on any server. Furthermore, the code generator created an interface with its implementations for each data model to perform **CRUD** operations. It is worth mentioning that tenant isolation and extensions modeled in the data architecture were appropriately reflected in the generated code. Specifically, tenant isolation is implemented by adjusting queries to filter by the tenant identifier (FQ4b), and extensibility is implemented in forms of aggregation and composition (FQ4d). This

demonstrates that CadaML enables addressing and designing the multi-tenancy concerns at the abstract level (FQ4), gives modelers a freedom to implement extensibility strategies that best suit their application requirements, and it serves the purpose of the development by producing the corresponding data access layer code from the model (FQ3). The involved representative of the research center highlighted these facts, and confirmed that CadaML would be feasible to apply in a practical setting.

Nevertheless, there are a few factors that turned out to be crucial to evolve the existing use case application into a multi-tenant web service using such modeling approach. Most importantly, the participation of the representative of the research center was beneficial as he was consulted to gather knowledge about the data layer, and to select the desired data partitioning scheme. In addition, we haven't spent much time and effort on enabling extensibility of the data layer since the existing database diagram already supported extensions of each table by providing additional tables to store custom data. Besides, the proficiency in modeling and code generation was also important to expedite learning and exploitation of the modeling language. Because not all cloud application and data layer developers have such expertise level on a regular basis, developers may require time and effort to familiarize with the modeling environment of CadaML. This is something we will investigate in Chapter 5.

4.4.3 Evaluation of the Adequacy of Multi-tenancy Implementation

To evaluate the quality of the generated code and adequacy of multi-tenancy implementation we combined code reviewing and unit testing techniques. As such, during code reviewing we selected snippets of the generated code, and investigated its readability, maintainability, correctness and vulnerabilities. Firstly, we validated that tables were appropriately encapsulated into the data model. Second, we ensured that relationships between tables were transformed in a form of aggregation or composition in data models. Third, a code excerpt that is responsible for establishing connection to a database was reviewed for logical errors, and queries were inspected to ensure that data operations isolate tenant-specific data appropriately. Finally, we reviewed the implementation of storing and retrieving custom data to/from additional tables.

The test cases listed in the previous section were expanded for individual methods of *process definition*, *task definition*, *process* and *task* entities. We come up with 56 test cases in total which are presented in Appendix E. All of these test cases are geared to support tenant isolation as shown in Table 4.1. Among them, 16 and 8 test cases also cover management of custom extensions and referencing data, respectively.

The expanded test cases for *process definition* and *process* entities are presented

Table 4.1: The distribution of test cases based on concerns under investigation per entity

	Process Definition	Task Definition	Process	Task	TOTAL
Tenant Isolation	16	16	12	12	56
Extensibility	4	4	4	4	16
Reference Data	4	4	0	0	8

in Table 4.2. Specifically, firstly, methods to manage a single *process definition* and a list of *process definitions* without any referencing data and custom attributes were tested. Then, methods to manipulate custom attributes of a single *process definition* were inspected. Finally, methods that operate over referencing data were investigated. For *process* entity, all methods except those that manipulate referencing data were examined. Test cases to manage referencing data were not extended for *process* and *task* entities since these entities do not refer to other entities in the data architecture. In the same manner, we tested methods to manage *task definition* and *task* entities.

All methods successfully passed all test cases without violating any of the concerns under the investigation. This might be due to the model validation tool that captures major errors in the model before generating the code. Based on this combined code reviewing and unit testing results, we can confirm that CadaML produces a valid data access layer code that meets multi-tenancy requirements in terms of tenant isolation and extensibility. Although these evaluation methods were sufficient for this particular case study due to the support from the research center, we acknowledge that automated methodology for testing multi-tenant applications would provide more quantitative validation. This is something we plan to do in the future.

4.5 Discussion

The findings of the case study certainly demonstrates feasibility of CadaML to implement and evolve the data layer of the given use case. We now discuss if the evolution challenges that are set in Section 4.2.2 have been addressed, and comment on the limitations of the performed case study.

4.5.1 Reflection on Challenges

When introducing multi-tenancy it is important to provide configurability or customizability of the data layer to support tenant-specific extensions. This require-

Table 4.2: The expanded test cases for *ProcessDefinition* and *Process* entities

	Process Definition	Process
TC1 (a)	Retrieve a process definition	Retrieve a process
TC1 (b)	Retrieve a process definition with custom attributes	Retrieve a process with custom attributes
TC1 (c)	Retrieve a process definition with corresponding task definitions and processes	-
TC1 (d)	Retrieve a list of process definitions	Retrieve a list of processes
TC2 (a)	Store a process definition	Store a process
TC2 (b)	Store a process definition with custom attributes	Store a process with custom attributes
TC2 (c)	Store a process definition with corresponding task definitions and processes	-
TC2 (d)	Store a list of process definitions	Store a list of processes
TC3 (a)	Update a process definition	Update a process
TC3 (b)	Update a process definition and its custom attributes	Update a process and its custom attributes
TC3 (c)	Update a process definition and corresponding task definitions and processes	-
TC3 (d)	Update a list of process definitions	Update a list of processes
TC4 (a)	Delete a process definition	Delete a process
TC4 (b)	Delete a process definition with corresponding custom attributes	Delete a process with corresponding custom attributes
TC4 (c)	Delete a process definition and corresponding task definitions and processes	-
TC4 (d)	Delete a list of process definitions	Delete a list of processes

ment is already supported by the actual data architecture of the use case where additional tables hold custom attributes. As a result, tenants can include additional information regarding their processes, tasks and their definitions. During the evolution, we adopted the same approach to address CH1 concern, although it provides limited flexibility of the data architecture.

Another main concern in multi-tenancy is isolation of tenants in the database. For the use case, all tenants are deployed to a shared database with a shared schema after analyzing the existing partitioning patterns for relational databases. To associate tenants with their data in a shared database, the tenant identifier is included in tables. Hence, CH2 challenge is solved as using the tenant-specific identifier provides logical isolation of tenant data. Furthermore, the evaluation confirms that CH1 and CH2 challenges are correctly addressed by testing methods of the generated code for tenant isolation and extensibility of the data architecture.

It is also important to make database access scalable since a single database instance is exploited across multiple tenants. A requirement for scalability is that new database servers should be added to the database pool when needed. Fortunately, current cloud providers eliminates the necessity to manually implement such scalability requirement by offering auto scaling capability of their services at the [PaaS](#) level. Subsequently, we deployed the data layer of the use case to an instance of Amazon [RDS](#). This, in turn, supports CH3 concern by provisioning on-demand resources to meet the database workload.

Finally, we confirm that CadaML as a proposed solution addresses CH4 challenge by mitigating the evolution process and maintenance effort of the data layer. The evolution process is alleviated as CadaML removed most of the manual implementations due to the following capabilities. Firstly, it allows implementation of database partitioning in a model. Secondly, errors are captured by the validation tool in the model. Lastly, the model is transformed to the corresponding data access layer code using the code generator tool. Meanwhile, maintenance effort is reduced by deploying data of all tenants to a single database. This also leads to higher utilization of database resources and lower overall costs.

4.5.2 Limitations

The case study describes the advantages of applying CadaML through evolving a data architecture of an industrial web application. However, we could not fully demonstrate applicability of the modeling language. The reason for this is the data layer of the use case is designed for relational databases, although CadaML supports different cloud data storage types offered at the [PaaS](#) service model. This limitation requires another study where we need to model a multi-tenant data architecture as a combination of multiple cloud storage solutions.

Moreover, we did not compare CadaML against other [DSLs](#) or modeling techniques due to the lack of other works that tackle multi-tenancy at the data layer. This is discussed and highlighted in more detail in Chapter [2](#). We also did not compare CadaML against manual code re-factoring, as a baseline due to its common use for implementation of multi-tenancy. Therefore, we need a more in-depth study to quantify the benefits of the modeling language. More concretely, we need to identify how CadaML affects productivity of developers compared to the manual method, assess quality of the generated code, and evaluate usability of the language. Such evaluation requires conducting an experimental user study with real developers who are familiar with Java programming language, cloud application and data layer implementation. In order to address these limitations, we performed a controlled experiment with task analysis technique that is described in Chapter [5](#).

4.6 Summary

This chapter described the application of CADA_{ML} and our qualitative evaluation of the modeling language. The application demonstrates how the modeling environment of the language can facilitate designing different data partitioning patterns, and generating source code that corresponds to each data partitioning option. Meanwhile, the evaluation shows the feasibility of CADA_{ML} in practical settings and the correctness of multi-tenancy implementation by the generated code through a case study. The feasibility is investigated by qualitatively assessing CADA_{ML}, its modeling environment and code generation tool. While, the adequacy of multi-tenancy implementation is assessed by applying double approach of code reviewing and unit testing. Although the evaluation outcome validates the applicability of CADA_{ML} in an industrial setting, and emphasizes the sufficiency of multi-tenancy implementation, we conducted a more in-depth study with real developers to further evaluate the modeling language. The study is presented in the next chapter.

Chapter 5

Experimental Evaluation

In order to quantify benefits of CadaML, the experimental use case described in Section 4.2 is exploited. The purpose of the evaluation is to assess the productivity of developers, reliability of the generated code, and usability of CadaML. The productivity is measured in terms of time required to design the data layer and completion rate of the experiment tasks. The reliability is calculated by debugging the generated code against several test cases. Meanwhile, the usability is evaluated through an interview regarding the concepts and graphical editor of CadaML. Moreover, CadaML is compared against manual code re-factoring where developers implement the data layer of the given use case for AWS.

This chapter starts with the description of the application’s data layer evolution in Section 5.1. Then, Section 5.2 presents the experimental design and evaluation methods that are adopted to assess the modeling language. The proficiency levels of participants in programming languages, cloud application/data layer implementation, and modeling tools are interpreted in Section 5.3, while their allocation is presented in Section 5.4. Section 5.5 explains the modeling process of the data architecture using CadaML. The evaluation results of productivity, reliability, and exit interviews are illustrated in Sections 5.6, 5.7, and 5.8, respectively. Section 5.9 discusses the findings and limitations of the experiment, and Section 5.10 analyzes threats to validity. Finally, Section 5.11 summarizes this chapter.

5.1 Evolving the Application

During the evolution process, the data architecture of the use case is re-designed to use a combination of different cloud storage solutions. This, in turn, provides scalability, customizability and extensibility of the data layer, and reduces the costs for data storage. Figure 5.1 shows, at a high level, which data is stored in the different types of storage.

The application collects most of the tenant information with configuration data

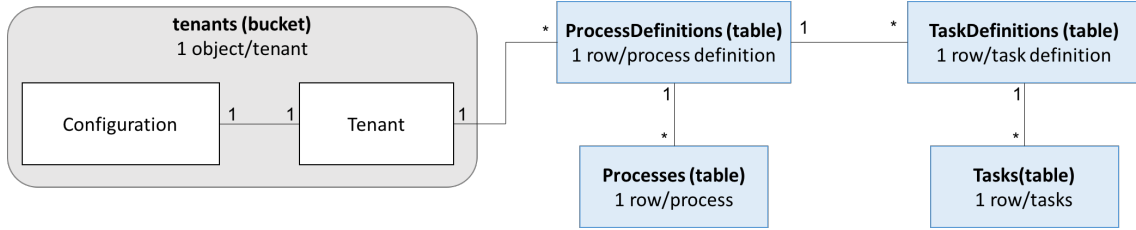


Figure 5.1: Data storage in the business process application.

during on-boarding process, and stores them as a single object in a public bucket named ‘*tenants*’. A *tenant* and its *configuration* are modeled as separate entities to enable customization and management of each entity independently. These entities will be deployed to Amazon [S3](#).

In the meantime, the application stores *process definitions*, *processes*, *task definitions* and *tasks* entities in separate non-relational tables in Amazon DynamoDB. Storing these entities in non-relational tables simplifies the implementation of customizability and extensibility of the application. To extend these tables in a relational database, additional tables are used to hold custom attributes (see Figure [4.3](#)). Fortunately, non-relational tables allow to use multiple schemes in the same table, thus, each tenant can have its own custom attributes.

Nevertheless, non-relational databases support limited operations which restricts execution of complex queries. Therefore, for tenants who need complex analysis and management of their own custom reporting requirement, the application will provision a new relational database instance of Amazon [RDS](#) during the on-boarding process. For tenants with such requirements, the provisioning process will create necessary tables in the database. Ideally, the actual database scheme should be remained unchanged. For the experiment, the same set of entities that are used for non-relational databases but with different organizational structure are constructed.

5.2 Experimental Design

The experiment strategy is to employ real developers and quantify their experience in using *CadaML*. In order to systematically conduct the evaluation, it is important to carefully plan the experiment procedure. Therefore, the evaluation of *CadaML* is carried out in adherence to the controlled experiment design with the task analysis technique. Such analysis allows us to observe how participants interact with *CadaML* in order to identify advantages *CadaML* could bring, understand difficulties participants might face when using the modeling language, and determine improvements that might be needed.

5.2.1 Experiment Procedure

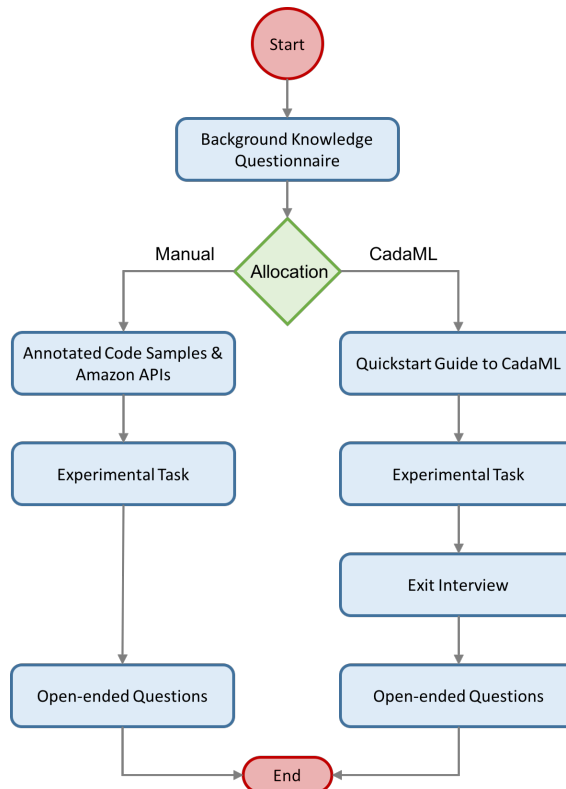


Figure 5.2: The flowchart of the experiment procedure.

The experiment procedure lasts for a maximum of an hour, and it takes place according to the flowchart illustrated in Figure 5.2. (i) Firstly, a participant fills a questionnaire about his/her experience in programming languages, cloud application/data layer development and modeling tools. (ii) Then, the participant is assigned to implement the data architecture either through manual coding or using CadaML. The allocation is performed in an alternating order. This type of allocation is based on the between-group design to avoid interaction effects, and to ensure equal number of participants for both approaches. (iii) For participants who are assigned to manually implement the experiment tasks, code samples annotated with comments and Amazon APIs documentations are provided. Similarly, participants using CadaML are given a very brief quickstart guide. Moreover, additional guidance is offered upon request for participants who have trouble interacting with CadaML or Amazon APIs. (iv) The participant is given an experiment task which differs based on participant's allocation. More detailed description of the experiment tasks are presented in Section 5.2.3. It is worth noting that the participants work independently and are unaware of each other's work. (v) After completing the experiment task, participants who are allocated to use CadaML are interviewed regarding reliability of the code, usability of the modeling language, and how CadaML affected

their productivity. The interview questions are listed in Section 5.2.4. (vi) Finally, open-ended questions are included towards the end of the experiment to solicit feedback on struggles participants had while manually implementing the experiment tasks, and things to improve in CadaML. Answering these questions is optional.

5.2.2 Participant Recruitment

Based on the experiment design and goals of the evaluation, we recruited real developers to participate in the experiment. The recruitment involved a series of activities including identifying eligible participants, explaining the experiment to the potential participants, acquiring informed consent, ensuring ethical standards, and supporting participants throughout the experiment.

Participants for the experiment were employed from the body of Computer Science researchers and graduate students (*i.e.*, Masters and PhD students) at the School of Computing and Communications, Lancaster University. The recruitment process took place between May and July 2018. The main requirements for participants were knowledge in Java programming language, and, preferably, experience in cloud application or data layer implementation. As an incentive for participation, an Amazon online shopping voucher was offered (£10 in value) that was given upon completion of the experiment task.

In total 24 developers showed interest in participating in the experiment. Unfortunately, one developer who was allocated for the manual method withdrew after reading the experiment task due to the lack of experience in cloud applications and data layer implementation.

5.2.3 Experimental Task

Once participants are recruited, they are allocated to evolve the data layer of the use case using either manual approach or CadaML. Depending on the allocation, participants are given an experiment task. In general, an experiment task is divided into separate sections for each storage type where each section contains a list of implementation tasks. These implementation tasks are then used as a checkpoint system to gauge the level of completion of each participant.

For the manual approach, sections of each storage type consist of two groups of implementation tasks. The first group covers creation of data models, and the second group comprises of interaction with the storage solution. Specifically, there are 20 tasks in the relational database section, 18 in the non-relational database section, and 8 the in blob storage section.

The implementation tasks for CadaML include modeling elements of each storage type, defining relationships between storage elements, validation of the modeled

elements and their relationships, and generation of the data access layer code from the model. The modeling specification contains 34 tasks for the relational database, 26 tasks for the non-relational database, and 18 tasks related to the blob storage.

5.2.4 Exit Interview Questions

The participants who use CadaML for modeling the data architecture of the use case are interviewed after finishing the experiment tasks. They are asked a set of interview questions on three different themes, and asked to respond using a 5-point Likert scale ('Strongly disagree', . . . , 'Strongly agree'). Furthermore, additional follow up questions may be asked to clarify the reason behind certain responses. For example, if a participant agreed that CadaML makes the data layer implementation easier, a rationale that supports the participant's response is requested. The interview questions with additional follow up questions in brackets are as follows.

- Productivity
 - a) I spent less time to come up with source code. (How much time would it take to manually write the code?)
 - b) CadaML makes the data layer implementation easier. (What made you disagree/agree/be neutral about this statement?)
 - c) Extra manual coding [other than custom code for business logic] is required to implement the data layer. (What extra code did you add?)
- Quality of Generated Code
 - a) The generated code is easy to read. (What makes the generated code easy/not easy to read?)
 - b) Fewer errors occur compared to manual coding.(If disagreed, what types of errors did you encounter in the generated code?)
 - c) It is harder to find errors in the generated code compared to manually written code. (If agreed, what makes it harder to identify errors?)
- Usability
 - a) CadaML is difficult to use. (What makes CadaML difficult/easy to use?)
 - b) CadaML restricts my freedom as a programmer. (What did you find restrictive?)
 - c) The concepts and notations are intuitive to use. (What makes it intuitive/non-intuitive?)

The described experimental design and experimental tasks have been reviewed and approved by the Faculty of Science and Technology Research Ethics Committee ([ESTREC](#)) which is an Institutional Review Board ([IRB](#)) at Lancaster University.

5.3 Participant Expertise

Overall 23 developers participated in the experiment with varying expertise levels in Java, cloud application/data layer development, and modeling tools.

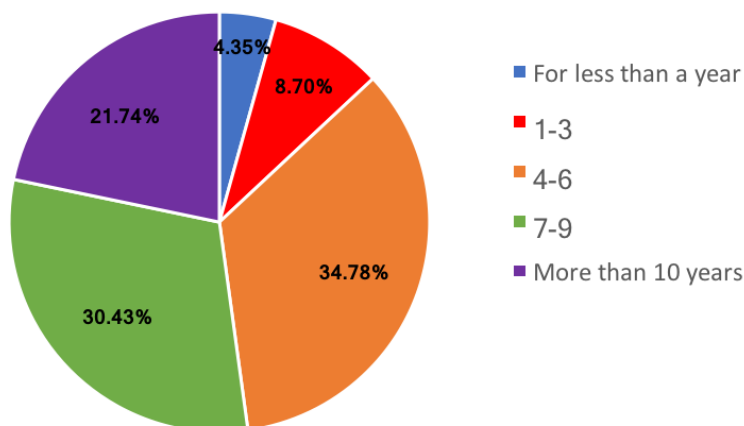


Figure 5.3: The participants' self-reported programming experience in years.

As illustrated in Figure [5.3](#), the majority of participants are proficient programmers who have been using various languages for years. Specifically, around 85% of participants have coding experience for at least 4 years. While, the remaining participants (4.35%(1) and 8.70% (2)) have been using programming languages for up to 3 years.

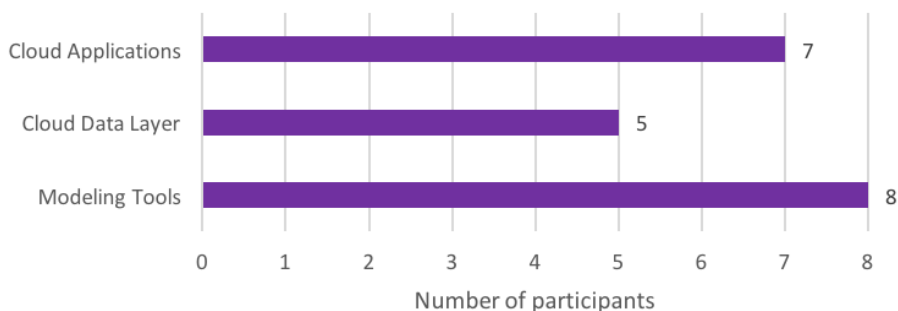


Figure 5.4: The number of participants based on self-reported experience in cloud application development and modeling.

Of all 23 developers who took part in the experiment, 7 participants have cloud applications implementation background as shown in Figure [5.4](#). Only 5 among them have experience in data layer implementation of cloud applications. Nearly

same number of developers (8) have used modeling tools such as [UML](#), Microsoft Visio, MySQL Workbench and feature modeling tools.

Table 5.1: The participants' self-reported expertise level in cloud [APIs](#) and storage services.

	Low (1-2)	Medium (3-5)	High (6-7)	TOTAL
APIs	-	1	6	7
SQL Databases	-	-	5	5
NoSQL Storage	1	1	3	5
Blob Storage	1	2	2	5

Meanwhile, Table [5.1](#) presents that nearly all of the participants who have implemented cloud applications are highly competent in applying [APIs](#) provided by cloud service providers. Similarly, the majority of the experienced participants in cloud data layer implementation have medium and high expertise level in using different cloud storage services. Only one participant reported low proficiency in implementing a data layer using [NoSQL](#) and blob storage.

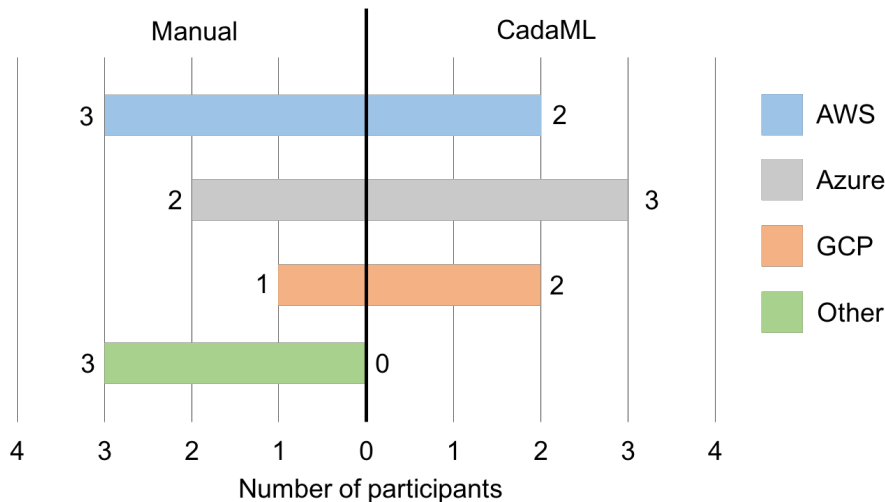


Figure 5.5: The number of participants based on self-reported experience in exploiting cloud service providers, and their allocation.

Figure [5.5](#) demonstrates that [AWS](#) and Azure are mostly exploited cloud service providers by the participants. More precisely, 5 participants have developed and deployed cloud applications using services of these cloud providers. The number of participants who have used [GAE](#) (3) is slightly less. The reason for this might be that [AWS](#) and Azure are current market leaders, while [GAE](#) is a relatively new provider. Interestingly, 3 participants mentioned that they have also implemented cloud applications using other cloud service providers, namely International Business

Machines ([IBM](#)) SoftLayer ^[1], eApps ^[2], and Vultr ^[3].

5.4 Participation Allocation

The allocation of the participants with their self-reported expertise level in Java is presented in Table 5.2. Among the participants, 11 developers manually implemented the data layer, while 12 developers used CadaML. In general, most of the participants have medium (9) and high (12) level expertise in Java, and one participant in each approach at a beginner’s level.

Table 5.2: The participants’ self-reported expertise level in programming with Java.

	Low (1-3)	Medium (4-6)	High (7-9)	TOTAL
Manual	1	5	5	11
CadaML	1	4	7	12
Total	2	9	12	23

Figure 5.6 shows that among the participants who are allocated to the manual approach, 4 participants have cloud applications development background. Most of these participants (3) also have experience in data layer implementation of cloud applications and exploiting modeling tools. Moreover, these participants have medium and high level proficiency in cloud application and data layer implementation.

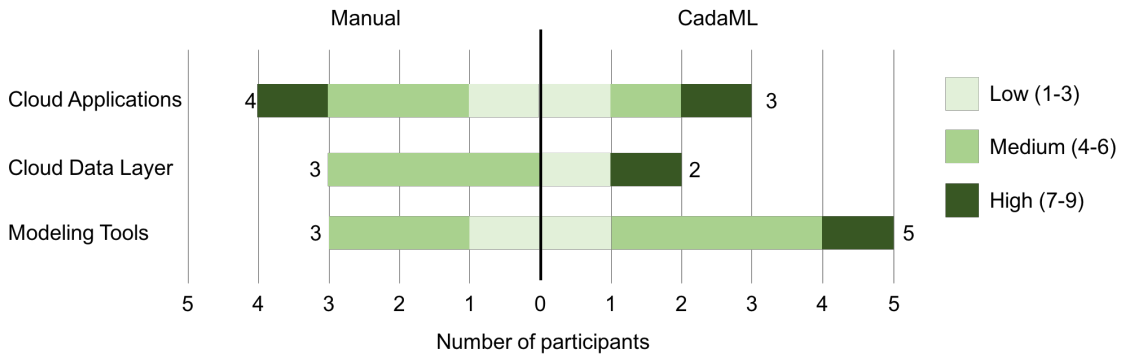


Figure 5.6: The number of participants based on self-reported expertise level in years in cloud application development and modeling tools, and their allocation.

In the meantime, of all the participants who are assigned to use CadaML, 3 participants have experience in cloud applications implementation, where 2 of them have also developed the data layer. Finally, 5 participants have used different modeling tools for more than 4 years.

¹<http://www.softlayer.com/>

²<https://www.eapps.com/>

³<https://www.vultr.com/>

Both the table and the figure emphasize that developers are as fairly allocated for both approaches as possible, without taking skills into consideration.

5.5 Modeling in CadaML

CadaML is exploited by participants who are allocated to model the data architecture of the experimental use case. Modeling the architecture starts with a creation of a database diagram using the graphical editor. Then, instances of the corresponding cloud data storage types, specifically, *Object Storage*, *NoSQL Database*, and *SQL Database*, are created in the diagram. It is worth mentioning that all participants came up with relatively the same model but with slight differences in the layout, and the provided figures represent a summary of the typical data architecture models by most participants.

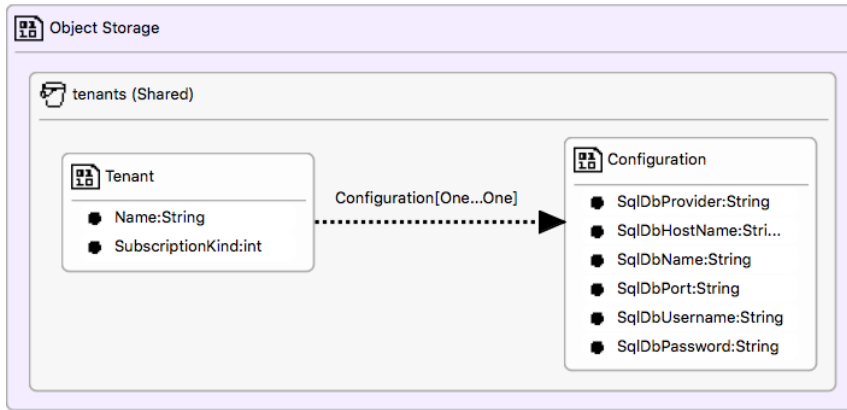


Figure 5.7: Object storage data architecture modeled in CadaML.

As shown in Figure 5.7, a single *bucket* is created in the *object storage*. The ‘*Shared*’ partitioning scheme is specified for the *bucket* since the *bucket* is used as a central storage for tenant-specific configuration data, and it is shared across all tenants. Within the *bucket*, *Tenant* and *Configuration* entities are modeled as *objects* with *attributes*, and the relationship between these *objects* are defined by the *object reference*.

The *Tenant* object has ‘*Name*’ and ‘*SubscriptionKind*’ attributes where the first attribute is set as the *key* that will be associated with an instance of the *object* when storing it in the *bucket*. Meanwhile, ‘*Configuration*’ object holds configuration information that is used to provision a new relational database instance, and it is bound to the *Tenant* object.

A *NoSQL database* instance is modeled with ‘*Shared Tables*’ partitioning scheme as illustrated in Figure 5.8. In the database, non-relational entities are modeled as *NoSQL tables* with their partition keys and row keys. However, the partition keys and row keys are not shown in the diagram because they are specified as attributes

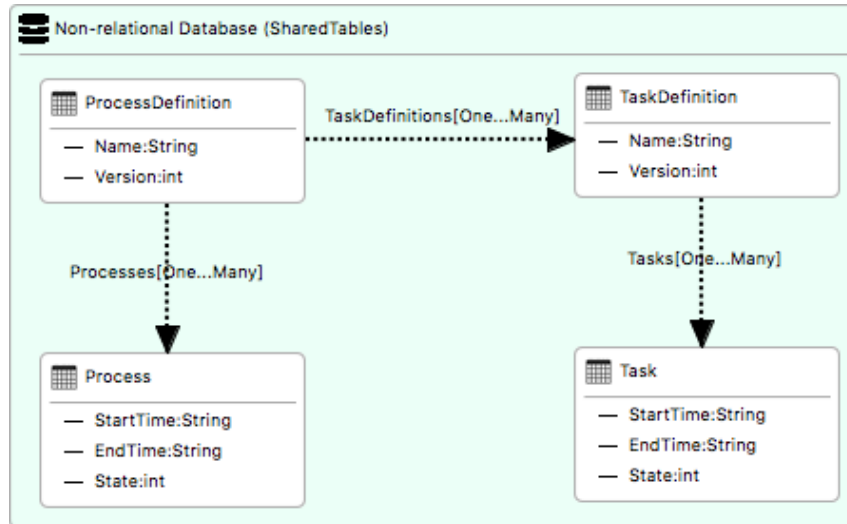


Figure 5.8: Non-relational data architecture modeled in CadaML.

in the CadaML meta-model, thus, they can be seen in the *Properties* tab in the graphical editor.

The partition key of the *ProcessDefinition* table contains the tenant identifier. This value allows filtering by tenant identifier, and ensuring the isolation of process definitions by tenant. While the row key comprises the process definition identifier to make sure that tenants cannot create two process definitions with the same identifier. The partition key for the *Process* and *TaskDefinition* tables contain the row key from the *ProcessDefinition* table, which is the process definition identifier. This enables the application to insert all processes and task definitions for a process definition in a single transaction, and to retrieve them from a single partition. In the meantime, the *Process* and *TaskDefinition* tables hold the process identifier and task definition identifier, respectively, in their row keys.

Similarly, the row key of the *TaskDefinition* table (*i.e.*, task definition identifier) is set as the partition key for the *Task* table, and the task identifier is included in the row key. Other elements of entities are added as *properties*, and the relationships between entities are captured by *NoSQL reference*.

As illustrated in Figure 5.9, a *SQL database* instance is created with ‘*Separate Database Per Tenant*’ partitioning scheme, as the application will provide an instance of *SQL database* for tenants who require additional reporting capabilities. Relational entities are created as *SQL tables* with their *fields*. For primary key fields, *isPrimaryKey* property is set to *true*. The relationships are specified using *SQL reference*, where a foreign key serves as a link between entities.

When the design of the data architecture is complete, CadaML checks the model for errors, and validates the model before generating other artifacts from it. If there is no violation of constraints and validation rules, a modeler can transform the model

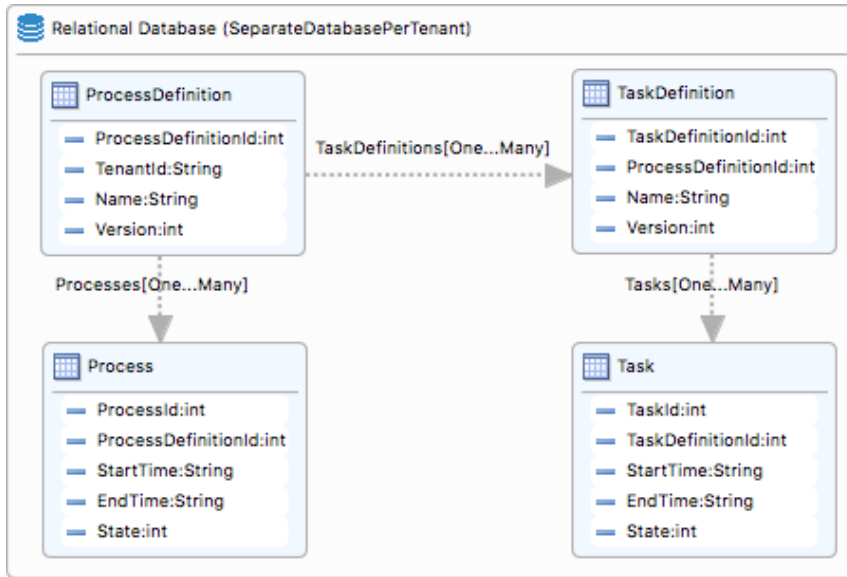


Figure 5.9: Relational data architecture modeled in CadaML.

to the data access layer code in Java for Alibaba Cloud, **AWS**, and Azure.

5.6 Evaluation of Productivity

Defining right metrics is important to measure productivity of developers. In general, developer productivity can be determined based on measurement of work completed, and quality of the completed work. Therefore, the productivity of participants is evaluated by the completion rate of tasks through testing and debugging the generated/written code, and recording the implementation time per storage type.

Table 5.3: Time spent (in h:min:s) and completion rate (CR) by participants for each storage type through manual implementation








































	Blob Storage		NoSQL		SQL		
	CR	Time	CR	Time	CR	Time	
P1	50.0%	31:39	25.0%	29:07	0.0%	-	
P2	62.5%	25:48	40.0%	35:02	0.0%	-	
P3	62.5%	36:26	20.0%	23:41	0.0%	-	
P4	87.5%	39:12	20.0%	21:02	0.0%	-	
P5	0.0%	-	25.0%	25:22	30.0%	34:44	
P6	0.0%	-	52.0%	01:00:09	0.0%	-	
P7	0.0%	-	60.0%	01:00:23	0.0%	-	
P8	100.0%	38:54	20.0%	21:06	0.0%	-	
P9	62.5%	41:23	20.0%	18:38	0.0%	-	
P10	100.0%	36:24	25.0%	23:36	0.0%	-	
P11	75.0%	24:48	25.0%	20:55	0.0%	-	
Median	62.5%	36:25	25.0%	23:41	0.0%	34:44	

Table **5.3** shows the completion rate of the implementation tasks for each storage

type, as well as the associated time required to develop the data architecture through manual coding. Before starting, participants are shown tutorials including brief code samples, the time of which is not included in the demonstrated figure. On average, participants spent 30-35 minutes for each storage type, with a median completion rate of 62.5% for blob storage, 25.0% for non-relational database, and 0.0% for relational database. Using median gives a much more representative value for our results, as some participants could not accomplish tasks for particular storage types in an hour.

Within the hour of time given for the experiment, none of the participants could fully accomplish all experiment tasks using manual methods. Two participants (*i.e.*, **P8** and **P10**) showed 100% completion rate for blob storage, with the best completion rate for non-relational data architecture being 60% (*i.e.*, **P7**). Nevertheless, the latter participants spent the given time for implementing only the non-relational data architecture. Meanwhile, only one participant could manually implement 30% of the data layer for relational database. This clearly demonstrates the complexity of successfully completing the required task using manual methods in under an hour.

Table 5.4: Time spent (in h:min:s) and completion rate (CR) by participants for each storage type using CadaML

	Blob Storage		NoSQL		SQL	
	CR	Time	CR	Time	CR	Time
P12	100.0% 	15:28	100.0% 	14:55	100.0% 	20:05
P13	100.0% 	13:40	100.0% 	11:48	100.0% 	15:57
P14	100.0% 	13:26	100.0% 	08:24	100.0% 	11:03
P15	100.0% 	08:41	100.0% 	06:07	100.0% 	10:19
P16	100.0% 	13:08	100.0% 	12:11	100.0% 	16:36
P17	100.0% 	16:46	100.0% 	15:54	100.0% 	19:40
P18	100.0% 	16:09	100.0% 	19:07	100.0% 	12:41
P19	100.0% 	10:18	100.0% 	11:32	100.0% 	09:08
P20	100.0% 	16:18	100.0% 	18:06	100.0% 	17:12
P21	100.0% 	11:27	100.0% 	11:36	100.0% 	12:43
P22	100.0% 	22:17	100.0% 	19:53	100.0% 	16:23
P23	100.0% 	09:54	100.0% 	12:16	100.0% 	08:07
Median	100.0% 	13:33	100.0% 	12:14	100.0% 	14:20

In stark contrast to the manual method, it can clearly be seen that using CadaML significantly improves the completion rate and development time as demonstrated in Table 5.4. Similarly to the manual approach, the time spent to familiarize the participants with the user interface of CadaML (around 2-3 minutes) is not included in the figure.

The data here provides interesting results. Participants spent around 14 minutes on average to model the data architecture of each storage option. The minimum time

required for blob storage, non-relational and relational databases are about 9, 6, and 8 minutes, respectively. Meanwhile, the maximum times were 22 minutes for blob storage, and 20 minutes for non-relational and relational databases. Moreover, all participants fully completed all three experiment tasks within an hour, and produced code that successfully passed all test cases.

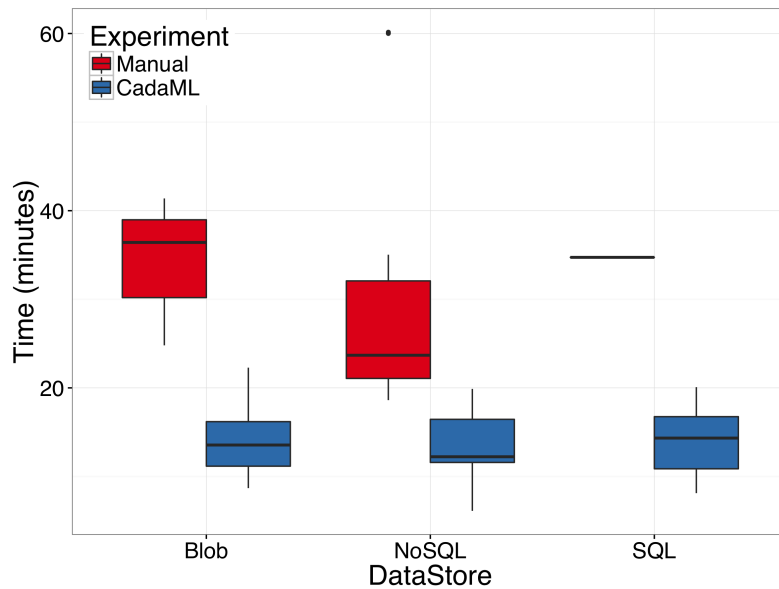


Figure 5.10: The distribution (median and interquartile range) of time taken by participants to finish the data layer implementation tasks using the 3 different data-store types. Using CadaML significantly reduces the development time. Note that only one participant attempted to accomplish any progress on `SQL` using manual implementation, hence the very narrow box on the far right.

To further expand on this, the general distribution of time taken by the participants in both experiments is depicted using boxplots in Figure 5.10. In general, 30-40 minutes were required to manually implement the blob storage architecture, and 20-35 minutes for the non-relational database structure. The majority of the participants started the implementation with the blob storage, and spent any remaining time developing other storage types. Therefore, the time for the implementation of non-relational data access layer is less, but with lower completion rates (as discussed before, and in Table 5.3). Unfortunately, implementation time for relational databases could not be generalized, as only one participant attempted to accomplish the experiment task for this storage type. On the other hand, most participants using CadaML were able to finish the data layer implementation in 10-17 minutes.

5.7 Error Analysis

It is critical to ensure that the application code consistently performs according to its specification. For this purpose, the written/generated data access layer code by each participant is evaluated using JUnit⁴ testing framework. The data access layer code is tested and debugged on a method level against several test cases that are similar to those that are described in Section 4.4.3, Chapter 4. The test cases are designed to demonstrate the implementation correctness of the experiment tasks, and the results of the evaluation are presented in this section.

Table 5.5: The distribution of test cases planned for each storage type per participant.

	Blob Storage	NoSQL Database	SQL Database	TOTAL
Test Cases	5	17	17	39

As shown in Table 5.5, in total 39 test cases per participant were formulated based on the experiment tasks to demonstrate the correctness of the written methods. The test cases are designed to verify storage initialization and data manipulation operations. For example, tenant and its configuration are considered as a single entity, hence, the test cases are written to check establishing a connection to **AWS S3**, creating a bucket in a storage, uploading, retrieving, and deleting a tenant with its configuration. Similarly, the test cases for **NoSQL** and **SQL** databases are designed in the same manner except **CRUD** operations were tested for each entity in the data architecture.

Table 5.6: The distribution of test cases that were planned and execute with their success and failure status.

	Planned	Executed	Passed	Failed
Blob Storage	55	40	26	14
NoSQL Database	187	187	56	131
SQL Database	187	17	6	11
TOTAL	429	244	88	156

The overall number of the planned test cases was 429 as demonstrated in Table 5.6. However, we only executed 244 as not all of the participants were able to

⁴<https://junit.org/junit5/>

fully implement the experiment tasks in an hour of given time. Although the success rate of test cases for blob storage showed 65%, the remaining test cases failed with the approximate success rates for `NoSQL` and `SQL` databases 30% and 35%, respectively. In contrast, the generated code by `CadaML` passed all the test cases without any major errors in the code.

Noticeably more errors were encountered in the application code by participants who manually implemented the data access layer. Specifically, errors were discovered in the code of 9 (out of 11) participants. On the contrary, `CadaML` users fared much better: only 5 (out of 12) participants made errors in the data architecture model, most of which were captured (as discussed below) by the validation tool of `CadaML`.

During the manual implementation experiment, the most common errors were incorrect implementation of: (i) object serialization and de-serialization to upload and retrieve a blob; (ii) non-relational table creation; and (iii) storing referenced entities in a non-relational database. The reason for these errors seem to stem from some participants perceiving the provided code samples as prescriptive rather than illustrative. For example, in the Amazon `S3` tutorial, an example is given of uploading a file as a blob, not as a Java object. Some participants simply ignored this fact, and blindly followed the tutorial when instead they needed to upload as a Java object, which caused errors. Another possible reason is the time constraint. Some participants may have felt the need to fully finish the experiment tasks in the allocated time of an hour without ensuring the validity of their code.

Conversely, there were no fundamental errors in the code generated by `CadaML`. Moreover, most of these errors were captured and fixed by the validation tool. Examples of such errors include: (i) missing primary keys for relational tables; (ii) incorrect multiplicity specification for a relationship between non-relational tables; and (iii) creation of relationships between wrong tables. The participants who encountered such errors admitted that they were made because of lack of attention while following the experiment tasks. This might be suggest too much reliance on `CadaML`, although it is difficult to tell if this is indicative without conducting a much wider study.

5.8 Exit Interview Results

After completing the experiment tasks, participants who are assigned to model using `CadaML` are interviewed about their experience in exploiting the modeling language. The first three interview questions are aimed to find out how `CadaML` affects the productivity. The next three questions are related to the reliability of the generated code, and the remaining questions focus on the usability of `CadaML` (see Section `5.2.4`).

5.8.1 Productivity

The productivity-related questions are geared to solicit comments on time spent to implement the data layer, whether CadaML eases or complicates the development process, and adequacy of the generated code to accomplish the implementation of the data layer. The participants' feedback regarding these questions are depicted in Figure 5.11.

All participants admitted that using CadaML reduces the time required to come up with source code for the use case. The participants who are experienced in Java and cloud application/data layer implementation indicated that it would take them from 2 to 4 hours to manually develop the given experiment task, while the other participants stated it would take at least a day and a maximum of 3 days.

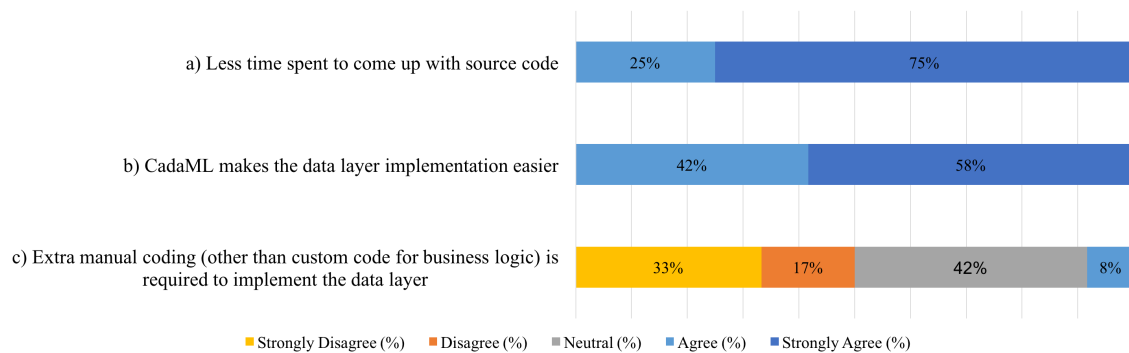


Figure 5.11: Participant feedback regarding productivity: all agreed that CadaML helps reduce implementation time and difficulty, but not all agreed that it was sufficient on its own.

The participants also agreed that the modeling language made the implementation process easier. According to the answers to the follow up question, the participants emphasized several benefits of CadaML to support this statement. Firstly, concepts and notations provided by CadaML hide implementation details of different cloud data storage solutions. Second, being able to design a data architecture as a combination of three cloud storage types in a single model gives a general overview of the application's data layer. Third, visual representation of a data architecture is more convenient to understand and manipulate. Finally, making changes in the model is easier than in the application code and it shortens the development time.

Nonetheless, the participant responses differ about extra manual coding required to implement the data layer. The half of the participants agreed that CadaML generates sufficient code to implement the data layer. Less than the half of the participants (42%) expressed neither agreement nor disagreement with this statement claiming that some extra manual coding may be needed depending on the application requirements. Only one participants (8%) stated that the generated code required few changes to fully implement the data layer.

5.8.2 Quality of the Generated Code

The quality of the generated code is measured by interviewing the participants regarding the readability of the generated code, occurrence of errors, and identifying errors in the code. The outcome of the interview responses is shown in Figure 5.12

Analyzing the responses, we find that the majority of participants (92%) acknowledged well readability of the generated code. The participants highlighted that the produced code is clear, well formatted, and it follows coding conventions and guidelines for Java. On the contrary, one participant (8%) found the generated code neither easy nor hard to read. The response of the participant is affected by participant's personal preferences.

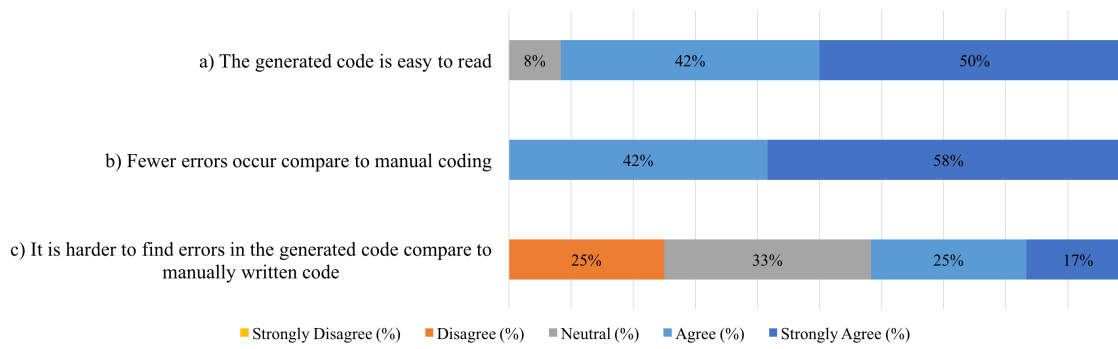


Figure 5.12: Participant feedback on reliability: generated code is of high readability and low frequency of errors; but with mixed perceptions about the ability of finding errors in the generated code.

All participants claimed that fewer errors occurred in the code with **CadaML** in contrast to manual implementation. Most commonly, the participants pointed out the visual interface and model validation capability of **CadaML** as the main features to reduce errors. In particular, visual representation of a data architecture prevents making mistakes in defining model elements and links between them. While the model validation helps to identify and capture errors at the model level without digging in into the code.

However, for 42% of participants found it harder to locate errors in the produced code compared to the manually written alternative. The participants gave different reasons to support their opinions. Some participants expressed that it is difficult to identify cause of errors in the code since **CadaML** generates a large number of Java classes and interfaces. Other participants referred to the fact that managing own code is less challenging as opposed to the machine generated one. In the meantime, one third of the participants stated that finding errors in the code requires same effort regardless the code is written manually or produced using the modeling language.

5.8.3 Usability of CadaML

Difficulty in exploitation, flexibility and intuitiveness of concepts and notations provided by the modeling language are the main criteria to assess the usability of CadaML. The participants responses regarding these aspects are illustrated in Figure 5.13.

The most of the participants (83%) argued that CadaML is relatively easy to use due to its simple user interface and convenient tools to model a data architecture. Nevertheless, a few participants (17%) struggled with applying CadaML when first started using the modeling editor. The reason for struggling was unfamiliarity with concepts of some tools. Therefore, these participants found the modeling language neither difficult nor easy to use.

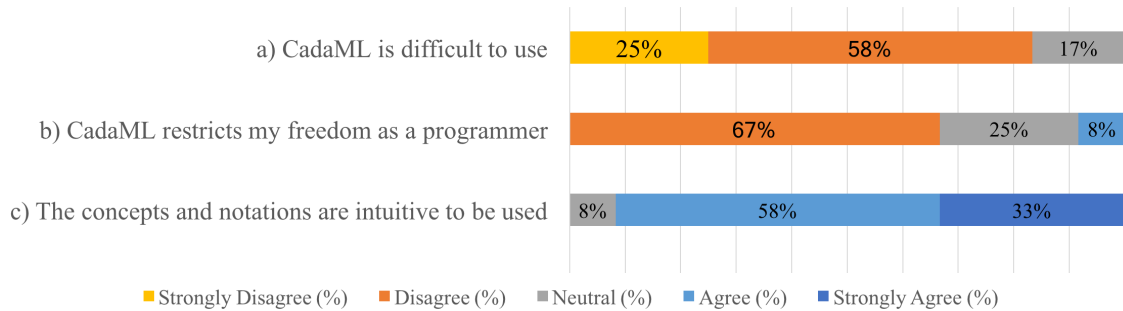


Figure 5.13: Participant feedback on usability: CadaML is generally perceived to be intuitive and easy to use without restricting the developers' freedom of choice.

More than half of the participants (67%) agreed with flexibility of CadaML to implement the data layer. On the contrary, one participant (8%) stated that CadaML restricts his freedom as a programmer. The participant expected to manually implement some parts of the data layer and model other parts using the modeling language. The remaining participants (25%) evaluated CadaML somehow flexible and restrictive at the same time. The reason for this is that the participants can make changes in the generated code. However, changes in the code are not reflected in the model.

Almost all participants (92%) found the concepts and notations provided by CadaML intuitive enough to be applied. The participants appreciated that CadaML reuses the existing terminology to define elements of different cloud data storage solutions. The remaining participants (8%) expressed neutral point of view regarding the intuitiveness of concepts and notations because of lack of knowledge in cloud data storage types.

5.9 Discussion

This section discusses the findings of the experiment and the exit interview. It reflects on the strengths and weaknesses of **CadaML** in terms of ease of exploitation, productivity of developers and code generator.

5.9.1 Ease of Exploitation

A major strength of **CadaML** is that it is generally simple and easy to exploit. Based on the participants' feedback, the graphical interface is one of the prime characteristics that makes the modeling language convenient to use. Moreover, the graphical representation of a data architecture facilitates the modeling process, and eases understanding the data architecture. This makes **CadaML** suitable for modelers with little to strong technical knowledge in different cloud data storage solutions to create and manipulate multi-tenant data architectures.

Currently, **CadaML** allows designing a data architecture in a single model as a combination of three cloud data storage. This enables to get an overview of an application's data layer, though, representation and management of a graphical model with many elements can be cumbersome. Creating a separate data architecture model for each cloud data storage could alleviate representation of the model, but would not solve the manageability issue.

In addition, a minority of participants found the graphical interface of **CadaML** restricting and suggested a few improvements. Particularly, the participants recommended including copy and paste capability to automate repetitive tasks such as creating a model element, replicating attributes of a model element, or duplicating relationships between model elements. The participants also proposed attributes of model elements to be typed with Java primitive data types since **CadaML** generates data access layer code in Java. Finally, the participants suggested to emphasize primary and foreign keys of relational tables, and keys of objects in a model.

5.9.2 Productivity of Developers

The results of the experiment certainly show that using **CadaML** reduces the time to implement the data layer of the given use case. More concretely, manual implementation would take around 3–5 hours in average to fully accomplish the experiment tasks. This data is extrapolated from the experiment outcome presented in Table 5.3. In contrast, with **CadaML** the participants spent 42 minutes on average to complete the data layer implementation. Therefore, we can claim that the participants spent about 4–7 times less time exploiting **CadaML** than using the manual approach.

Furthermore, improvements in the productivity of developers are echoed in the exit interview results. According to the participants, the main factors that facilitate increase in productivity are the visualization of the data architecture, validation tool, and code generator. The discussed benefits and drawbacks of the visualization in the previous section also apply to the productivity of developers. Meanwhile, the validation tool ensures consistency of the model by capturing errors at the model level. This eliminates the need to manually find and fix errors in the application code. Finally, the code generator shortens the time to come up with data access layer code by producing it from a model.

In terms of a limitation, an additional time is required to learn the modeling language, its concepts and graphical editor when a modeler starts using CadaML. Unfortunately, we cannot provide the exact amount of time that would be needed to familiarize the participants with the modeling language for two main reasons. First, the participants were given 2–5 minutes introduction to the modeling environment of CadaML before starting the experiment. Secondly, the participants were offered additional guidance when they struggled exploiting the modeling language during the experiment. Nevertheless, the experiment results demonstrate significant reduction in time and increase in the completion rate when applying CadaML. Thus, we can argue that the time to learn the language could be effectively traded for long term improvements in the productivity of developers.

5.9.3 Code Generator

Another advantage of CadaML is that it (semi-)automates the implementation of the data layer by producing all necessary code from a model. This removes a lot of time doing repetitive coding tasks such as creating data models, implementing **CRUD** operations for each data model, and establishing connection to different types of cloud data storage. However, some participants mentioned a few limitations which made the generated code inconvenient to maintain and restrictive to modify.

The participants mainly referred that they would need more time to understand the generated code compared to their own written alternative. The participants also concerned that testing and debugging the generated code may be complicated as CadaML produces a number of Java classes and interfaces. In contrast, some participants disagreed with these limitations by arguing that understanding the generated code and finding errors in it depends on a level of a programmer and the application complexity. Furthermore, synchronization issues emerge between the model and the generated code, because changes in the latter are not reflected in the former. This issue requires a capability to handle such two-way synchronization and automatically propagate changes between the model and the code.

5.10 Threats to Validity

There are some threats that may hinder validity of the inference of our experiment. These threats are divided into four types where each type addresses a specific methodological question.

5.10.1 Construct Validity

Before starting the experiment, some design decisions have been made to systematically conduct the experiment and properly perform data collection. In order to avoid the threat of an interaction effect, the between-group design is applied to allocate participants. Following this design, the participants are divided into two groups where each group uses different approaches to evolve the data layer of the given use case.

For data collection, the participants are interviewed regarding their experience in using CadaML once completing the experiment tasks. The interview questions are formulated using a Likert type scale. The chosen method enables quantifying the participants' responses, though it does not allow gathering qualitative data on benefits and drawbacks of CadaML. Thus, additional questions are asked based on participant's responses to the interview questions, and open-ended questions are included towards the end of the interview.

5.10.2 Internal Validity

A number of developers have been recruited for the experiment with various capabilities in programming and cloud data layer implementation. This, in turn, increases the threat that these external factors influenced the experiment results. Nevertheless, with CadaML the participants performed better in terms of completion rate and implementation time than using the manual approach. The participants also gave positive comments and feedback about CadaML and its features in their qualitative evaluation. Moreover, there were considerably less errors in the generated code. Therefore, we can assure that CadaML results improvements in productivity of developers and quality of the application code.

5.10.3 External Validity

Among the experiment participants, more than 65% reported lack of knowledge in cloud application and data layer implementation. This leads to the concerns in the generalizability of the experiment results to cloud experts. Despite of this, we identified significant improvement in productivity across all the developer-base.

However, further evaluation with cloud experts would provide more comprehensive insights about CadaML.

Furthermore, the representativeness threat may be incurred due to the selected use case. To mitigate this threat, a data layer of a business process analyzing application has been chosen. The application is a real-world web service with a complex data model owned by a major telecommunication company, and its architecture is similar to many enterprise applications. During the experiment, the data architecture is evolved to store the application data in different cloud storage solutions, and the evolved application prototype is approved by the research center.

5.10.4 Conclusion Validity

Although a necessary amount of data has been collected to validate the benefits of CadaML, a limited amount of time (*i.e.*, 1 hour) was given to the participants to develop the experiment tasks. As a result, the participants who were assigned to the manual approach could not accomplish the experiment tasks in an hour of time. This may lead to the threat that the participants assigned to the manual approach may thought the necessity to complete the experiment tasks in the allocated time without guaranteeing the validity of their code. Regardless the time constraint, the experienced participants stated that it would take them 2–4 hours to manually implement the tasks.

Another threat is that external factors such as participants' knowledge and experience in cloud applications and data storage implementation might also influence the experiment outcome. This threat is alleviated by the fact that the participants were as fairly allocated as possible for both approaches without considering their expertise level. Therefore, we can conclude that with CadaML less time is spent to implement the data layer of the given use case, and with less errors appearing in the generated application code.

5.11 Summary

This chapter evaluated CadaML against manual code re-factoring, and quantified the benefits of the modeling language. As a use case for the evaluation, an industrial business process analyzing application is evolved by deploying the application data to a combination of different cloud storage services. The evaluation is conducted following the controlled experiment design with the task analysis technique.

For the experiment, a number of developers with varying familiarity with Java, cloud application and data layer implementation has been recruited. The developers are allocated to implement the experiment task either exploiting CadaML or

manually. During the evaluation, the productivity of developers, reliability of the generated code, and usability of the modeling language are assessed. The productivity is estimated by calculating the completion rate of the evaluation tasks, and time required to model/implement the data architecture. The reliability is measured by debugging and testing the generated/written code against several test cases. The usability is analyzed through an interview and open-ended questions at the end of the evaluation.

Through the evaluation, we have demonstrated that exploiting CadaML can significantly reduce the time and effort – by a factor of 4–7 – to implement the data architecture, and decrease number of errors in the application code. We also discussed limitations that are gathered through the exit interview, and considered threats to our study.

Chapter 6

Discussions & Conclusions

In this chapter, we summarize the thesis, describe the significance of our findings, discuss limitations of CadaML, and present new insights that emerged as a result of this research.

6.1 Thesis Summary

The aim of this thesis is to *facilitate implementation of multi-tenancy in the data layer of cloud applications using MDE techniques*. The research, to achieve this aim, is developed following the three research phases: 1) *literature review* of the past and current approaches that are geared towards addressing multi-tenancy concerns at the data layer; 2) *development* of CadaML, its graphical editor and modeling environment; and 3) *evaluation* of CadaML in order to identify its application benefits.

The purpose of the *literature review* phase is to analyze information gathered by identifying gaps and limitations in the related research, and formulating requirements for CadaML. Chapter 2 examines the existing literature and identifies a set of (i) challenges of developing multi-tenancy at the data layer; (ii) current cloud modeling languages and their important features; and (iii) their weaknesses and limitations.

The outcome of the *literature review* is expanded in *development* phase to converge on requirements for CadaML and to build these requirements into an actual modeling language. Chapter 3, firstly, specifies the requirements along with associated rationale for concepts that need to be included in CadaML, and for a meta-modeling language to implement the language. The chapter, then, identifies a methodology to develop CadaML through analyzing the existing literature on DSL development. Following the methodology, the specified requirements are transformed into a modeling language with a graphical editor. This was accomplished by analyzing different cloud storage solutions offered at the PaaS provisioning level by four major cloud service providers, designing a meta-model based on the do-

main knowledge gathered through the analysis, and implementing CadaML and its modeling environment.

The *evaluation phase* is reflected in Chapters 4 and 5, where the modeling language is investigated through (1) a qualitative evaluation based on a case study; and (2) an experimental evaluation which involved a controlled user study and an in-depth interview with developers who have exploited CadaML.

The evaluation is performed on evolution of an industrial web application to adopt multi-tenancy, and further evolution of the data layer to store application data in different cloud data storage. The qualitative evaluation explores the application of CadaML to a real-world industrial application, and demonstrates adequacy of multi-tenancy implementation. While the experimental evaluation provides quantitative assessment of the benefits of CadaML in terms of well usability of the language, increased productivity of developers, and improved reliability of the data access layer code. Hence, these chapters present evidence that CadaML effectively facilitates implementation of multi-tenancy at the data layer by engaging developers with diverse programming and cloud application development background, and offering concepts and notations that enable developers to build a multi-tenant data architecture in a cloud provider agnostic way.

6.2 Contributions

This thesis makes several contributions that are categorized around the three research phases:

Literature review of manual approaches and modeling techniques to implement multi-tenancy at the data layer in order to identify challenges of multi-tenancy implementation and gather an appropriate feature set for CadaML.

- C1. A literature search that captures the previous and current manual and modeling approaches along with patterns for multi-tenant data architecture development (Chapter 2).
- C2. The overlap among manual and modeling approaches, and gap in them are identified through analyzing published literature on multi-tenancy implementation (Chapter 2, Section 2.6).

Development of CadaML to mitigate and expedite the implementation of multi-tenant data architectures for cloud applications.

- C3. A set of concepts and terminology requirements for CadaML are formulated based on the design methodology on graphical DSL development 44 (Chapter 3, Section 3.1.1).

- C4. A set of meta-modeling language requirements to implement CadaML are specified (Chapter 3, Section 3.1.2).
- C5. A domain model is formulated that captures common vocabulary of available cloud data storage solutions with their partitioning alternatives at the PaaS service level offered by four major public cloud service providers (Chapter 3, Section 3.3.4).
- C6. A meta-model of the language is designed and implemented that converge on the requirements specified in C2 and C3 (Chapter 3, Section 3.4.3), and these requirements are integrated into a graphical modeling language (Chapter 3, Section 3.5.3).
- C7. A set of deterministic validation rules are specified to keep a model created using CadaML consistent, and ensure reliability of the model-to-code transformation (Chapter 3, Section 3.5.4).
- C8. A code generation tool is implemented to produce a data layer implementation, from a validated model, with multi-tenancy management logic that corresponds to the specific data storage types and policies selected by the developer (Chapter 3, Section 3.5.5).

Evaluation of CadaML in terms of its applicability, usability, productivity of developers, and reliability of the generated code

- C9. A case study is conducted to qualitatively evaluate the applicability of CadaML to design and implement a multi-tenant data architecture of an industrial web application (Chapter 4).
- C10. An experimental user study is performed where real developers with varying expertise level further evolved the data layer of the industrial web application to deploy it on different cloud data storage solutions. Through the user study, we observe how CadaML affects the productivity of developers, evaluate whether the language improves reliability of the generated code, and assess the usability of the modeling language (Chapter 5).

These contributions certainly support the research aim to “*facilitate the implementation of multi-tenancy in the data layer of cloud applications using MDE techniques*”. Furthermore, the followed language design methodology built up in the implementation of CadaML, and the evaluation methodologies resulted in the constructive assessment of the benefits and limitations of the modeling language.

6.3 Reflection on Research Objectives

As a result of considering multi-tenant applications, their challenges and the existing approaches that tackle these challenges, Section 1.4 identified a list of research objectives to facilitate the implementation of multi-tenant data architectures. These research objectives are now revisited with reflection on how they are addressed throughout the thesis.

- R01:** *Analyze existing approaches that consider multi-tenancy challenges at the data layer of cloud applications.* In Chapter 2, we critically reviewed and analyzed the previous and current approaches that include both manual implementations and modeling techniques related to enabling multi-tenancy in the data layer. Through reviewing and analyzing these approaches, we gave an overview of the significant literature published, described the relationship among approaches, identified their limitations, revealed the gaps that exist in the literature, and presented our perspective on the research in enabling multi-tenancy.
- R02:** *Provide a way to describe a multi-tenant cloud data architecture at an abstract level.* In Chapter 3, we specified a set of requirements for concepts and notations that need to be included in CadaML in order to provide a unified representation of varying cloud data storage types offered by different cloud service providers. These requirements then were transformed into a modeling language that enables building multi-tenant data architectures in a cloud provider independent way by hiding cloud-specific implementation details. In Chapter 4, we qualitatively evaluated this in a collaboration with an experienced researcher in cloud application implementation. The outcome of the evaluation emphasizes that CadaML provides developers an appropriate level of abstraction by only representing the essential elements of the data layer and multi-tenancy patterns which also enabled developers to focus on the data architecture rather than on the implementation variance of different cloud data storage types.
- R03:** *Reduce the development effort during the implementation of a multi-tenant cloud data architecture.* We compared CadaML against manual coding technique in Chapter 5 where two separate groups of developers introduced multi-tenancy to the data architecture of an industrial application. The outcome of the comparison showed that developers spent 4–7 times less time exploiting CadaML than using the manual approach.
- R04:** *Improve reliability of the application code (specifically at the data layer).* To decrease the number of errors in the application code, CadaML provides model

validation and code generation tools. To evaluate whether this is achieved, the generated code is assessed by combining manual code reviewing and automated unit testing in Chapters 4 and 5. Code reviewing showed that a data architecture modeled using CadaML was appropriately transformed into the corresponding data access layer code. Unit testing confirmed that the generated code ensures isolation of tenants and management of custom data. Furthermore, unit testing demonstrated that the number of errors in the application code was reduced by almost half when using CadaML.

R05: *Offer developers with a reasonable level of usability.* The usability of CadaML was evaluated by interviewing developers who used the modeling language during the controlled experiment in Chapter 5. All participants emphasized that the graphical editor makes the modeling a data architecture more convenient. They also highlighted that the visual representation eases understanding the model, and saves effort when applying changes in the model. Moreover, the majority of the participants were satisfied with the concepts and notations provided by the language.

6.4 Limitations

Beside the benefits of applying CadaML that are mentioned above, there are a few limitations identified during the evaluation of the language and its modeling environment.

- Using CadaML developers can design a data architecture that consists of different cloud data storage solutions in a single model. This, in turn, enables to get an overview of an application's data layer, eases the understandability of the data architecture, and mitigates the exploitation of the modeling language. However, representation and managing a graphical model with a large number of elements can be challenging. Although CadaML provides a hierarchical tree view of model elements, and allows designing each cloud data storage type in a separate model to alleviate the representation issue, this would not properly address the manageability challenge. This is something that could be addressed using *model slicing* techniques where a large and complex model is broken down into relevant model parts or elements.
- CadaML produces the data access layer code from a model in order to automate the data layer implementation of cloud applications. However, developers might need to extend or customize the generated code to meet additional data layer requirements. Currently, changes in the generated code are not reflected in the model which leads to synchronization issues between the code and the model.

Moreover, changes in the generated code are overwritten whenever developers re-produce the code from the model. This limitation requires two-way synchronization where relevant modification in one artifact need to be propagated to another. Model synchronization is outside the scope of this thesis but a very relevant area.

6.5 Discussion

- **Efficient cloud application implementation and evolution**

Through implementing and applying CadaML, we realized that modeling approaches could be suitable for prototyping cloud applications as they enable describing a purpose of an application more easily than with coding. Thus, modeling can produce a high-level overview of an application and give the ability to generate an application structure with basic functionality already implemented. This in turn can provide a good starting point for developers to further extend, customize or enrich the generated prototype to meet the application requirements.

Moreover, modeling approaches can actually facilitate efficient implementation and evolution of cloud applications. As in prototyping, developers can design a cloud application in a high-level model based on requirement specifications. An application model can be then transformed into an application code using code generation tools. When application requirements evolve over time, changes can be directly captured in the model and, subsequently, reflected in the application code. This is useful to keep the requirement specifications, application model, and application code up to date, and it expedites the evolution process.

Finally, model validation and code generation capabilities can also bring a number of benefits. Code generation helps to automatically parse a model into executable application code. Meanwhile, model validation helps to keep a model consistent, handle errors within the model before producing source code, and ensure semantic correspondence of the generated code with the target environment. In our case, CadaML significantly improved productivity of developers in terms of time and development effort by a factor of 4–7 as opposed to the manual implementation, increased quality of the application code, and minimized number of errors in it.

- **Cloud portability and generalizability of CadaML**

Currently, cloud providers offer specific tools and libraries to support developing applications that can interact with their own services and platforms. Therefore, the implemented applications are locked to a particular cloud platform. In the

future, customers may consider moving to another cloud environment due to changes in their requirements or policies [7, 83, 106]. However, moving from services of one cloud platform to another requires tremendous re-engineering effort [16, 30]. To address this issue and enable portability between cloud service providers, customers could benefit from abstraction offered by modeling techniques. Thus, a cloud application model can be designed in a platform-independent manner, and platform-specific artifacts can be transformed from the model anytime. As a supportive to this, we showed that CadaML supports Alibaba, Amazon, and Azure, though it would still be applicable to other cloud service providers that offer similar data storage solutions such as Backblaze B2 Cloud Storage, Rackspace Cloud Files, MongoDB, Oracle **NoSQL** Database, and Cassandra. To support more cloud platforms, we would need to extend the code generation tool, or use abstraction libraries that can operate across multiple clouds. However, we avoided using such abstraction libraries in this thesis as they currently support only blob storage of a few cloud service providers while most real world cloud applications store their data in different storage solutions including relational and non-relational databases.

- **Lowering barrier to adopt cloud environments**

There is a number of different cloud service providers offering heterogeneous cloud services and solutions. Moreover, each cloud service provide has its own tools and libraries to develop cloud applications as described in previous section. This diversity is often seen as an obstacle by new cloud customers and cloud users [7, 13, 89]. Modeling approaches can help to overcome this obstacle through an appropriate level of abstraction to exploit cloud services, and build cloud applications using these services. As a result, cloud customers could be provided with an ability to develop and host cloud applications on different cloud platforms without going into cloud-specific implementation details. The prominent example is Force.com [1], a cloud application development platform that offers cloud customers a meta-data driven modeling environment to model and run custom applications on its cloud environment. Another example is the **TOSCA**-based cloud-agnostic modeling framework Cloudify [2] that can be used to automate application orchestration, maintenance and management.

On the other hand, it would also be a mistake to assume that modeling approaches can anticipate and address all the challenges associated with cloud application development. Some development tasks such as complex business logic, security related concerns, and fault handling are arguably easier and more

¹<http://www.force.com/>

²<https://cloudify.co/>

efficient to implement manually. Therefore, modeling languages should be integrated with manual implementations to express different concerns of cloud application development. Some modeling approaches can be domain specific while others more related to technical concerns so they can be more generic.

- **Modeling languages for cloud applications**

In Chapter 2, we considered past and current modeling languages so that we could possibly reuse their concepts or extend them to realize CadaML. This could significantly reduce language development time, and provide an initial level of quality. However, most of the modeling approaches propose diverse modeling concepts even though they are geared towards describing the data architecture of cloud applications. Although TOSCA has been proposed as a standardized software modeling language, we could not extend it as it is intended to describe a topology instead of structure of cloud applications in terms of components and their relationships.

Moreover, we considered exploiting CadaML in a combination with other modeling tools to fully automate cloud application development since there is also a high variety of modeling languages that capture cloud applications from different implementation perspectives. However, most of these languages are not mature enough, abandoned, or unavailable to public. Even with available tools such as multi-view cloud variability framework [88], CloudDSL [119], and CloudML-SINTEF [14], we failed to make them work properly due to lack of tool support and documentation. As a consequence, we couldn't extend existing modeling languages or integrate CadaML with them because of unclear overlap, semantic mismatches and interoperability issues.

Another interesting finding is that current modeling approaches are mostly applicable at design-time to generate cloud deployment configurations or part of an application implementation [13]. Though, considering run-time characteristics and evolution requirements within the model would be beneficial due to the following reasons. The former will allow to capture quality aspects of provisioned computing services which can be used for refining or optimizing cloud services. The latter will enable to efficiently evolve cloud application by defining evolution requirements in the model and transforming them into the application code.

- **DSL development methodologies and meta-modeling languages**

This thesis has shown that modeling approaches can help in managing the complexity of developing cloud applications, but efficient implementation of modeling languages requires a set of appropriate methodologies along with lightweight, and easy to use techniques and tools. In Chapter 3, we reviewed

published literature on [DSL](#) development and examined meta-modeling languages to implement CadaML. The reviewed literature differs from each other based on tools and frameworks they use to implement a modeling language. Some works embed detailed instructions, while others provide explicit design guidelines. The examined meta-modeling languages are also divergent regarding tool support, features and capabilities to design a [DSL](#). Some of the meta-modeling languages are not powerful enough to implement a complete language and require additional effort to complement their weaknesses. Based on this, we can argue that there are no commonly accepted methodologies and meta-modeling languages to develop graphical [DSL](#)s. Hence, there is a need for a standardized meta-modeling language, and formal methodologies and guidelines to support development and exploitation of graphical [DSL](#)s.

- **Other cloud data storage services**

In order to support the demands of emerging technologies and provide efficient solutions to open problems, cloud providers offer new data storage and analysis services such as graph databases and data streams. Graph databases are another type of non-relational databases to store and navigate relationships, and they are commonly used for fraud detection, social networking, and knowledge graphs. Meanwhile, data streams enable to implement applications to analyze and process streaming, for example, for collecting log and event data, real-time analysis, and social media feeds. These data storage and analysis services are optimized for certain use case scenarios and provide benefits over existing data storage and analyzing solutions. Supporting these services by CadaML would further expand its application and generalizability. This would require analyzing similar services of cloud providers, capturing conceptual and implementation commonalities and differences, extending the domain model and meta-model of CadaML, and subsequent modification of the graphical editor, model validation, and code generation tools.

- **Runtime model evaluation and optimization**

Evaluation and validation of non-functional requirements would provide valuable insights to optimize and adapt provisioned cloud services. This was also highlighted in the systematic review of cloud modeling languages [\[13\]](#). The runtime aspects of provisioned cloud resources can be typically monitored and managed using cloud integrated services such as Amazon CloudWatch [\[3\]](#), Azure Monitor [\[4\]](#), and Google Cloud Monitoring [\[5\]](#). CadaML could be integrated with

³<https://aws.amazon.com/cloudwatch/>

⁴<https://azure.microsoft.com/en-gb/services/monitor/>

⁵<https://cloud.google.com/monitoring/>

these services for responding to performance changes, optimizing resource utilization, and retrieving an overview of operational health based on the monitored information. This could be achieved by implementing an additional module that compares the status and workload of provisioned cloud services provided by the cloud monitoring services against non-functional customer requirements and adapts to changes if required..

6.6 Future Work

In addition to the future work to address the limitations outlined before, there are a number of avenues of work that proceed from this thesis, including the following:

- For the experimental study, developers of varying familiarity with Java and cloud data layer architectures were recruited. A benefit of this was to identify baseline improvement across the wide developer-base. However, a more in-depth study with experienced cloud data layer developers would provide further insights, particularly in usability of CadaML, appropriateness of concepts used in the language, and effects on development effort. This is a priority in the future research.
- With the growth of cloud service providers and evolution of cloud applications, more and more applications use a combination of different cloud data storage solutions from multiple cloud service providers to optimally exploit data storage services regarding pricing, performance, flexibility, geographical coverage, and other quality related characteristics. Consequently, implementing/enhancing abstraction libraries and multi-cloud management platforms is something for future work. Moreover, analyzing these quality requirements will help to adapt and optimize deployment configurations based on customer requirements. This needs monitoring and evaluating behavior of cloud services at run-time, and refining deployment configurations based on the evaluation results. This set of added value activities constitute part of the responsibilities of a trusted third party, i.e. a cloud broker [36].
- Since cloud services have been improving and new services have been continuously introduced over recent years, moving towards Functions as a Service (FaaS) could be a promising solution to address multi-tenancy concerns. Thus, investigating benefits and drawbacks of building multi-tenant cloud applications following this model would be a valuable contribution.

Bibliography

- [1] Mohammad Abu-Matar and Jon Whittle. MDE opportunities in multi-tenant cloud applications. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, CloudMDE@MoDELS 2014*, pages 1–5, Valencia, Spain, 2014.
- [2] Frank Ackerman, Lynne Buchwald, and Frank Lewski. Software inspections: An effective verification process. *IEEE software*, 6(3):31–36, 1989.
- [3] Frank Ackerman, Priscilla Fowler, and Robert Ebenau. Software inspections and the industrial production of software. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 13–40. Elsevier North-Holland, Inc., 1984.
- [4] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 187–197. IEEE, 2002.
- [5] Kena Alexander, Choonhwa Lee, Eunsam Kim, and Sumi Helal. Enabling end-to-end orchestration of multi-cloud applications. *IEEE Access*, 5:18862–18875, 2017.
- [6] Mikio Aoyama and Nozomi Kurono. An extended orthogonal variability model for metadata-driven multitenant cloud services. In *Software Engineering Conference (APSEC), 2013 20th Asia-Pacific*, volume 1, pages 339–346. IEEE, 2013.
- [7] AppDirect. AppDirect SMB cloud service adoption report, 2017.
- [8] Ankita Atrey, Hendrik Moens, Gregory Van Seghbroeck, Bruno Volckaert, and Filip De Turck. An overview of the OASIS TOSCA standard: Topology and orchestration specification for cloud applications. Technical report, IBCN-iMinds, Department of Information Technology, Gaston Crommenlaan, 2015.

- [9] Stefan Aulbach, Torsten Grust, Dean Jacobs, Alfons Kemper, and Jan Rittinger. Multi-tenant databases for software as a service: Schema-mapping techniques. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 1195–1206. ACM, 2008.
- [10] Paul Baker, Shiou Loh, and Frank Weil. Model-driven engineering in a large industrial context—Motorola case study. In *International Conference on Model Driven Engineering Languages and Systems*, pages 476–491. Springer, 2005.
- [11] Luiz Andre Barroso and Urs Hoelzle. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*, volume 8. Morgan and Claypool Publishers, 1st edition, 2013.
- [12] Keith H. Bennett and Václav Rajlich. Software maintenance and evolution: A roadmap. In *22nd International Conference on Software Engineering, Future of Software Engineering Track, ICSE*, pages 73–87, Limerick Ireland, Jun 2000.
- [13] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. A systematic review of cloud modeling languages. *ACM Comput. Surv.*, 51(1):22:1–22:38, February 2018.
- [14] Alexander Bergmayr, Alessandro Rossini, Nicolas Ferry, Geir Horn, Leire Orue-Echevarria, Arnor Solberg, and Manuel Wimmer. The evolution of CloudML and its applications. In *Proceedings of the 3rd International Workshop on Model-Driven Engineering on and for the Cloud 18th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2015), Ottawa, Canada, September 29, 2015.*, pages 13–18, 2015.
- [15] Alexander Bergmayr, Javier Troya, Patrick Neubauer, Manuel Wimmer, and Gerti Kappel. UML-based cloud application modeling with libraries, profiles, and templates. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, CloudMDE@MoDELS*, pages 56–65, Valencia, Spain, September 2014.
- [16] Alexandre Beslic, Reda Bendraou, Julien Sopenal, and Jean-Yves Rigolet. Towards a solution avoiding vendor lock-in to enable migration between cloud platforms. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering for High Performance and Cloud computing co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 5–14, Miami, Florida, USA, sep 2013.

- [17] Dominic Betts, Alex Homer, Alejandro Jezierski, Masashi Narumoto, and Hanzhong Zhang. *Developing Multi-tenant Applications for the Cloud on Windows Azure*. Microsoft patterns & practices, 2013.
- [18] Dominic Betts, Alex Homer, Alejandro Jezierski, Masashi Narumoto, and Hanzhong Zhang. *Moving Applications to the Cloud on Windows Azure*. Microsoft patterns & practices, 2013.
- [19] Cor-Paul Bezemer and Andy Zaidman. Challenges of reengineering into multi-tenant SaaS applications. *Technical Report Series TUD-SERG-2010-012*, 2010.
- [20] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. TOSCA: portable automated deployment and management of cloud applications. In *Advanced Web Services*, pages 527–549. Springer, 2014.
- [21] Raouf Boutaba, Qi Zhang, and Mohamed Faten Zhani. Virtual machine migration in cloud computing environments: Benefits, challenges, and approaches. In *Communication Infrastructures for Cloud Computing*, pages 383–408. IGI Global, 2014.
- [22] Hugo Bruneliere, Jordi Cabot, and Frédéric Jouault. Combining model-driven engineering and cloud computing. In *Modeling, Design, and Analysis for the Service Cloud-MDA4ServiceCloud’10: Workshop’s 4th edition (co-located with the 6th European Conference on Modelling Foundations and Applications-ECMFA 2010)*, 2010.
- [23] Sven Burmester, Holger Giese, Martin Hirsch, Daniela Schilling, and Matthias Tichy. The Fujaba real-time tool suite: model-driven development of safety-critical, real-time systems. In *Proceedings of the 27th international conference on Software engineering*, pages 670–671. ACM, 2005.
- [24] Hong Cai, Ning Wang, and Ming Jun Zhou. A transparent approach of enabling saas multi-tenancy in the cloud. In *6th World Congress on Services, SERVICES 2010*, pages 40–47, Miami, Florida, USA, 2010.
- [25] Everton Cavalcante, André Almeida, Thais Batista, Nélio Cacho, Frederico Lopes, Flavia C. Delicato, Thiago Sena, and Paulo F. Pires. Exploiting software product lines to develop cloud computing applications. In *Proceedings of the 16th International Software Product Line Conference - Volume 2, SPLC ’12*, pages 179–187, New York, NY, USA, 2012. ACM.

- [26] Guo Chag Jie, Sun Wei, Huang Ying, Wang Zhi Hu, and Gao Bo. A framework for native multi-tenancy application development and management. In *The 9th IEEE International Conference on E-Commerce Technology and The 4th IEEE International Conference on Enterprise Computing, E-Commerce and E-Services (CEC-EEE 2007)*, pages 551–558. IEEE Computer Society, 2007.
- [27] Clovis Chapman, Wolfgang Emmerich, Fermín Galán Márquez, Stuart Clayman, and Alex Galis. Software architecture definition for on-demand cloud provisioning. *Cluster Computing*, 15(2):79–100, 2012.
- [28] Frederick Chong and Gianpaolo Carraro. Architecture strategies for catching the long tail. *MSDN Library, Microsoft Corporation*, pages 9–10, 2006.
- [29] Frederick Chong, Gianpaolo Carraro, and Roger Wolter. Multi-tenant data architecture. *MSDN Library, Microsoft Corporation*, pages 14–30, 2006.
- [30] Cloud Standards Coordination (Phase 2). Cloud computing users needs - Analysis, conclusions and recommendations from a public survey. Special Report 003 381 V2.1.1, The European Telecommunications Standards Institute, February 2016.
- [31] Charles Consel and Renaud Marlet. Architecture software using: A methodology for language development. In Catuscia Palamidessi, Hugh Glaser, and Karl Meinke, editors, *Principles of Declarative Programming*, pages 170–194, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.
- [32] Bezemer Cor-Paul, Zaidman Andy, B. Platzbeecker, T. Hurkmans, and A. ’. Hart. Enabling multi-tenancy: An industrial experience report. In *2010 IEEE International Conference on Software Maintenance*, pages 1–8, 2010.
- [33] Katie Costello and Sarah Hippold. Gartner says nearly 50 percent of paas offerings are now cloud-only, 2018.
- [34] Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE ’14*, pages 211–222, New York, NY, USA, 2014. ACM.
- [35] Tharam S. Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: Issues and challenges. In *24th IEEE International Conference on Advanced Information Networking and Applications, AINA*, pages 27–33, Perth, Australia, Apr 2010.

- [36] Abdessalam Elhabbash, Faiza Samreen, James Hadley, and Yehia Elkhatib. Cloud brokerage: A systematic survey. *ACM Comput. Surv.*, 51(6):119:1–119:28, January 2019.
- [37] Yehia Elkhatib. Mapping cross-cloud systems: Challenges and opportunities. In *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016*, Denver, CO, USA, 2016. USENIX Association.
- [38] Yehia Elkhatib, Gordon S. Blair, and Bholanathsingh Surajbali. Experiences of using a hybrid cloud to construct an environmental virtual observatory. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms, CloudDP '13*, pages 13–18, New York, NY, USA, 2013. ACM.
- [39] Miao Fang, Georg Leyh, Joerg Doerr, and Christoph Elsner. Multi-variability modeling and realization for software derivation in industrial automation management. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 2–12. ACM, 2016.
- [40] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud computing patterns: Fundamentals to design, build, and manage cloud applications*. Springer, 2014.
- [41] Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin, and Arnor Solberg. Managing multi-cloud systems with CloudMF. In *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, pages 38–45. ACM, 2013.
- [42] Martin Fleck, Javier Troya, Philip Langer, and Manuel Wimmer. Towards pattern-based optimization of cloud applications. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, CloudMDE@MoDELS*, pages 16–25, Valencia, Spain, Sep 2014.
- [43] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Professional, 1st edition, 2010.
- [44] Ulrich Frank, Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom Cohen, and Jorn Bettin. *Domain-Specific Modeling Languages: Requirements Analysis and Design Guidelines*, pages 133–157. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

- [45] Yujian Fu, Zhijiang Dong, and Xudong He. An approach to validation of software architecture model. In *Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific*, pages 8–pp. IEEE, 2005.
- [46] Fatih Gey, Dimitri Van Landuyt, and Wouter Joosen. Middleware for customizable multi-staged dynamic upgrades of multi-tenant SaaS applications. In *8th IEEE/ACM International Conference on Utility and Cloud Computing, UCC*, pages 102–111, Limassol, Cyprus, Dec 2015.
- [47] Glauco Estacio Gonçalves, Patricia Takako Endo, Marcelo Anderson Santos, Djamel Fawzi Hadj Sadok, Judith Kelner, Bob Melander, and Jan-Erik Mångs. CloudML: An integrated language for resource, service and request description for D-clouds. *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 399–406, 2011.
- [48] Jack Greenfield and Keith Short. Software factories: Assembling applications with patterns, models, frameworks and tools. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '03*, pages 16–27, New York, NY, USA, 2003. ACM.
- [49] Joaquín Guillén, Javier Miranda, Juan Manuel Murillo, and Carlos Canal. A UML profile for modeling multicloud applications. In Kung-Kiu Lau, Winfried Lamersdorf, and Ernesto Pimentel, editors, *Service-Oriented and Cloud Computing*, pages 180–187, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
- [50] James Hadley, Yehia El-khatib, Gordon S. Blair, and Utz Roedig. MultiBox: Lightweight containers for vendor-independent multi-cloud deployments. In *Embracing Global Computing in Emerging Economies - First Workshop, EGC*, pages 79–90, Almaty, Kazakhstan, Feb 2015.
- [51] Mohammad Hamdaqa and Ladan Tahvildari. Stratus ML: A layered cloud modeling framework. In *2015 IEEE International Conference on Cloud Engineering, IC2E 2015*, pages 96–105, Tempe, AZ, USA, 2015.
- [52] James Hamilton. On designing and deploying internet-scale services. In *Proceedings of the 21st Conference on Large Installation System Administration Conference, LISA'07*, pages 18:1–18:12, Berkeley, CA, USA, 2007. USENIX Association.
- [53] Felienne Hermans, Martin Pinzger, and Arie Van Deursen. Domain-specific languages in practice: A user study on the success factors. In *International*

- Conference on Model Driven Engineering Languages and Systems*, pages 423–437. Springer, 2009.
- [54] Mike Hinchey, Sooyong Park, and Klaus Schmid. Building dynamic software product lines. *Computer*, pages 22–26, 2012.
- [55] Gordon Hogenson, Cai Saisang, Warren Genevieve, Alex Homer, Mike Jones, and Mike B. *Modeling SDK for Visual Studio - Domain-Specific Languages*. Microsoft, 2016.
- [56] Gordon Hogenson, Cai Saisang, Warren Genevieve, Alex Homer, Mike Jones, and Mike B. Modeling SDK for Visual Studio - domain-specific languages, 2016.
- [57] Ta'id Holmes. Automated provisioning of customized cloud service stacks using domain-specific languages. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, CloudMDE@MoDELS 2014*, pages 46–55, Valencia, Spain, September 2014.
- [58] Alex Homer, John Sharp, Larry Brader, Masashi Narumoto, and Trent Swanson. *Cloud design patterns: Prescriptive architecture guidance for cloud applications*. Microsoft patterns & practices, 2014.
- [59] Alex Homer, John Sharp, Larry Brader, Masashi Narumoto, and Trent Swanson. *Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications*. Microsoft patterns & practices, 2014.
- [60] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of the 5th International Conference on Software Reuse, ICSR '98*, pages 134–142, Washington, DC, USA, 1998. IEEE Computer Society.
- [61] Cloud Industry. Cloud computing models demystified, 2017.
- [62] Pooyan Jamshidi and Claus Pahl. Orthogonal variability modeling to support multi-cloud application configuration. In *European Conference on Service-Oriented and Cloud Computing*, pages 249–261. Springer, 2014.
- [63] Enrique Jiménez-Domingo, Javier Torres Niño, Angel Lagares Lemos, Miguel Lagares-Lemos, Ricardo Colomo Palacios, and Juan Miguel Gómez-Berbís. CLOUDIO: A cloud computing-oriented multi-tenant architecture for business information systems. In *IEEE International Conference on Cloud Computing, CLOUD 2010*, pages 532–533, Miami, FL, USA, July 2010.

- [64] Assylbek Jumagaliyev and Yehia Elkhatib. CadaML: A modeling language for multi-tenant cloud application data architectures. In *12th IEEE International Conference on Cloud Computing, CLOUD*, 2019.
- [65] Assylbek Jumagaliyev and Jon Whittle. Model-driven engineering for multi-tenant saas application development. In *Proceedings of the 3rd Workshop on CrossCloud Infrastructures & Platforms, CrossCloud '16*, pages 8:1–8:2, New York, NY, USA, 2016. ACM.
- [66] Assylbek Jumagaliyev, Jon Whittle, and Yehia Elkhatib. Evolving multi-tenant saas cloud applications using model-driven engineering. In *10th International Workshop on Models and Evolution, CEUR Workshop Proceedings*, pages 60–64. CEUR-WS.org, 10 2016.
- [67] Assylbek Jumagaliyev, Jon Whittle, and Yehia Elkhatib. Using DSML for handling multi-tenant evolution in cloud applications. In *IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2017*, pages 272–279, Hong Kong, December 2017.
- [68] Jaap Kabbedijk, Cor-Paul Bezemer, Slinger Jansen, and Andy Zaidman. Defining multi-tenancy: A systematic mapping study on the academic and the industrial perspective. *Journal of Systems and Software*, 100:139–148, 2015.
- [69] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1990.
- [70] Richard B Kieburtz, Laura McKinney, Jeffrey M Bell, James Hook, Alex Kottov, Jeffrey Lewis, Dino P Oliva, Tim Sheard, Ira Smith, and Lisa Walton. A software engineering experiment in software component generation. In *Proceedings of the 18th international conference on Software engineering*, pages 542–552. IEEE Computer Society, 1996.
- [71] Dongmin Kim, Hanif Muhammad, Eunsam Kim, Sumi Helal, and Choonhwa Lee. TOSCA-based and federation-aware cloud orchestration for Kubernetes container platform. *Applied Sciences*, 9(1):191, 2019.
- [72] Dimitrios Kolovos, Louis Rose, Richard Paige, and A Garcia-Dominguez. The Epsilon book. *Structure*, 178:1–10, 2010.

- [73] Derrick Kondo, Bahman Javadi, Paul Malecot, Franck Cappello, and David P Anderson. Cost-benefit analysis of cloud computing versus desktop grids. In *IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009.
- [74] Rouven Krebs, Christof Momm, and Samuel Kounev. Architectural concerns in multi-tenant SaaS applications. *Closer*, 12:426–431, 2012.
- [75] Indika Kumara, Jun Han, Alan Colman, Tuan Nguyen, and Malinda Kapuruge. Sharing with a difference: Realizing service-based SaaS applications with runtime sharing and variation in dynamic software product lines. In *2013 IEEE International Conference on Services Computing*, pages 567–574. IEEE, 2013.
- [76] Thomas Kwok, Thao Nguyen, and Linh Lam. A software as a service with multi-tenancy support for an electronic contract management application. In *2008 IEEE International Conference on Services Computing (SCC 2008)*, pages 179–186, Honolulu, Hawaii, USA, 2008.
- [77] Fabien Latry, Julien Mercadal, and Charles Consel. Processing domain-specific modeling languages: A case study in telephony services. In *Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems*, Portland, United States, October 2006.
- [78] Kwanwoo Lee, Kyo C Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In *International Conference on Software Reuse*, pages 62–77. Springer, 2002.
- [79] Frank Leymann, Christoph Fehling, Ralp Mietzner, Alexander Nowak, and Schahram Dustdar. Moving applications to the cloud: An approach based on application model enrichment. *International Journal of Cooperative Information Systems*, 20(03):307–356, 2011.
- [80] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. CloudCmp: comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Internet Measurement Conference, IMC*, pages 1–14, Melbourne, Australia, Nov 2010.
- [81] Hongbo Li, Yuliang Shi, and Qingzhong Li. A multi-granularity customization relationship model for SaaS. In *Web Information Systems and Mining, 2009. WISM 2009. International Conference on*, pages 611–615. IEEE, 2009.

- [82] Cui Lizhen, Wang Haiyang, Jinjiao Lin, and Haitao Pu. Customization modeling based on metagraph for multi-tenant applications. In *5th International Conference on Pervasive Computing and Applications*, pages 255–260, Dec 2010.
- [83] Logicworks. Roadblocks to cloud success, 2016.
- [84] Shuai Luan, Yuliang Shi, and Haiyang Wang. A mechanism of modeling and verification for SaaS customization based on TLA. In *International Conference on Web Information Systems and Mining*, pages 337–344. Springer, 2009.
- [85] Janne Luoma, Steven Kelly, and Juha-Pekka Tolvanen. Defining domain-specific modeling languages: Collected experiences. In *4th Workshop on Domain-Specific Modeling*, 2004.
- [86] Kun Ma, Bo Yang, and Ajith Abraham. A template-based model transformation approach for deriving multi-tenant SaaS applications. In *Acta polytechnica Hungarica*. Óbuda University, 2012.
- [87] Alexandre Michetti Manduca, Ethan V. Munson, Renata P. M. Fortes, and Mariada Graça C. Pimentel. A nonintrusive approach for implementing single database, multitenant services from web applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14*, pages 751–756, New York, NY, USA, 2014. ACM.
- [88] Mohammad Abu Matar, Rabeb Mizouni, and Salwa Alzahmi. Towards software product lines based cloud architectures. In *2014 IEEE International Conference on Cloud Engineering*, pages 117–126, 2014.
- [89] Joe McKendrick. What cloud computing customers want: Clarity, simplicity, support, 2014.
- [90] Peter Mell and Timothy Grance. SP 800-145. the NIST definition of cloud computing, 2011.
- [91] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, December 2005.
- [92] Metacase. MetaEdit+ workbench user’s guide.
- [93] Microsoft. Code generation and T4 text templates, 2016.

- [94] Ralph Mietzner, Andreas Metzger, Frank Leymann, and Klaus Pohl. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, PESOS '09, pages 18–25, Washington, DC, USA, 2009. IEEE Computer Society.
- [95] Ralph Mietzner, Andreas Metzger, Frank Leymann, and Klaus Pohl. Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications. In *International ICSE Workshop on Principles of Engineering Service-Oriented Systems, PESOS*, pages 18–25, Vancouver, BC, Canada, May 2009.
- [96] Hendrik Moens and Filip De Turck. Feature-based application development and management of multi-tenant applications in clouds. In *Proceedings of the 18th International Software Product Line Conference-Volume 1*, pages 72–81. ACM, 2014.
- [97] Fatma Mohamed, Mohammad Abu Matar, Rabeb Mizouni, Mahmoud Al-Qutayri, and Zaid Al Mahmoud. SaaS dynamic evolution based on model-driven software product lines. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom)(CLOUDCOM)*, pages 292–299, December 2014.
- [98] Jürgen Müller, Jens Krüger, Sebastian Enderlein, Marco Helmich, and Alexander Zeier. Customizing enterprise software as a service applications: Back-end extension in a multi-tenancy environment. In *International Conference on Enterprise Information Systems*, pages 66–77. Springer, 2009.
- [99] Dinh Khoa Nguyen, Francesco Lelli, Yehia Taher, Michael Parkin, Mike P. Papazoglou, and Willem-Jan van den Heuvel. Blueprint template support for engineering cloud-based services. In *Proceedings of the 4th European Conference on Towards a Service-based Internet, ServiceWave'11*, pages 26–37, Berlin, Heidelberg, 2011. Springer-Verlag.
- [100] Daniel Nurmi, Rich Wolski, Chris Grzegorzczak, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131. IEEE Computer Society, 2009.
- [101] Antonio Navarro Pérez and Bernhard Rumpe. Modeling cloud architectures as interactive systems. In *Proceedings of the 2nd International Workshop on*

- Model-Driven Engineering for High Performance and Cloud computing co-located with 16th International Conference on Model Driven Engineering Languages and Systems (MODELS*, pages 15–24, Miami, Florida, USA, Sep 2013.
- [102] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin, Heidelberg, 2005.
- [103] Ansar Rafique, Dimitri Van Landuyt, and Wouter Joosen. PERSIST: Policy-based data management middleware for multi-tenant SaaS leveraging federated cloud storage. *Journal of Grid Computing*, 16(2):165–194, Jun 2018.
- [104] Ansar Rafique, Dimitri Van Landuyt, and Wouter Joosen. PERSIST: Policy-based data management middleware for multi-tenant SaaS leveraging federated cloud storage. *Journal of Grid Computing*, 16(2):165–194, Jun 2018.
- [105] Mietzner Ralph, Unger Tobias, Titze Robert, and Leymann Frank. Combining different multi-tenancy patterns in service-oriented applications. In *2009 IEEE International Enterprise Distributed Object Computing Conference*, pages 131–140, Sept 2009.
- [106] 451 Research. *Hosting and cloud study 2016: The digital revolution, powered by cloud*, 2016.
- [107] Ariyattu C. Resmi and François Taiani. Fluidify: Decentralized overlay deployment in a multi-cloud world. In Alysson Bessani and Sara Bouchenak, editors, *Distributed Applications and Interoperable Systems*, pages 1–15, Grenoble, France, Jun 2015.
- [108] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering*, 14(2):131, 2009.
- [109] David S Wile. Supporting the DSL spectrum. *Journal of Computing and Information Technology*, 9(4):263–287, 2001.
- [110] Sherif Sakr, Anna Liu, Daniel M. Batista, and Mohammad Alomari. A survey of large scale data management approaches in cloud environments. *IEEE Communications Surveys and Tutorials*, 13(3):311–336, March 2011.
- [111] Faiza Samreen, Yehia El-khatib, Matthew Rowe, and Gordon S. Blair. Daleel: Simplifying cloud instance selection using machine learning. In *2016 IEEE/I-FIP Network Operations and Management Symposium, NOMS*, pages 557–563, Istanbul, Turkey, Apr 2016.

- [112] Oliver Schiller, Benjamin Schiller, Andreas Brodt, and Bernhard Mitschang. Native support of multi-tenancy in RDBMS for software as a service. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 117–128, New York, NY, USA, 2011. ACM.
- [113] Klaus Schmid and Andreas Rummler. Cloud-based software product lines. In *16th International Software Product Line Conference, SPLC*, volume 2, pages 164–170, Salvador, Brazil, Sep 2012.
- [114] Julia Schroeter, Sebastian Cech, Sebastian Götz, Claas Wilke, and Uwe Aßmann. Towards modeling a variable architecture for multi-tenant SaaS-applications. In *Sixth International Workshop on Variability Modelling of Software-Intensive Systems*, pages 111–120, Leipzig, Germany, Jan 2012.
- [115] Lutz Schubert, Keith Jeffery, and Burkhard Neidecker-Lutz. The future of cloud computing: Opportunities for European cloud computing beyond 2010. *Expert Group report, public version*, 1, 2010.
- [116] Rami Sellami, Sami Bhiri, and Bruno Defude. Supporting multi data stores applications in cloud environments. *IEEE Transactions on Services Computing*, 9(1):59–71, January 2016.
- [117] Ashraf A Shahin. Multi-dimensional customization modelling based on meta-graph for SaaS multi-tenant applications. *arXiv preprint arXiv:1402.6045*, 2014.
- [118] Ashraf A. Shahin. Variability modeling for customizable SaaS applications. *CoRR*, abs/1409.2156, 2014.
- [119] Gabriel Costa Silva and Louis M. Rose and Radu Calinescu. Cloud DSL: A language for supporting cloud portability by describing cloud entities. In *Proceedings of the 2nd International Workshop on Model-Driven Engineering on and for the Cloud co-located with the 17th International Conference on Model Driven Engineering Languages and Systems, CloudMDE@MoDELS*, pages 36–45, Valencia, Spain, September 2014.
- [120] Ian Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, USA, 9th edition, 2010.
- [121] Borja Sotomayor, Rubén Montero, Ignacio M Llorente, and Ian Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet computing*, 13(5):14–22, 2009.

- [122] Diomidis Spinellis. Notable design patterns for domain-specific languages. *J. Syst. Softw.*, 56(1):91–99, February 2001.
- [123] Shekhar Srikantaiah, Aman Kansal, and Feng Zhao. Energy aware consolidation for cloud computing. In *Proceedings of the 2008 Conference on Power Aware Computing and Systems, HotPower’08*, pages 10–10, Berkeley, CA, USA, 2008. USENIX Association.
- [124] Leonardo Tizzei, Marcelo Nery dos Santos, Vinícius Segura, and Renato FCerqueira. Using microservices and software product line engineering to support reuse of evolving multi-tenant SaaS. In *Proceedings of the 21st International Systems and Software Product Line Conference, SPLC 2017*, volume A, pages 205–214, Sevilla, Spain, September 2017.
- [125] Dykstra Tom, Anderson Rick, and Wasson Mike. *Building Real-World Cloud Apps with Windows Azure*. Microsoft Corporation, 2014.
- [126] Eddy Truyen, Dimitri Van Landuyt, Vincent Reniers, Ansar Rafique, Bert Lagaisse, and Wouter Joosen. Towards a container-based architecture for multi-tenant SaaS applications. In *Proceedings of the 15th International Workshop on Adaptive and Reflective Middleware*, page 6. ACM, 2016.
- [127] Chang-Hao Tsai, Yaoping Ruan, Sambit Sahu, Anees Shaikh, and Kang G Shin. Virtualization-based techniques for enabling multi-tenant management tools. In *International Workshop on Distributed Systems: Operations and Management*, pages 171–182. Springer, 2007.
- [128] Wei-Tek Tsai and Xin Sun. SaaS multi-tenant application customization. In *Seventh IEEE International Symposium on Service-Oriented System Engineering, SOSE 2013*, pages 1–12, San Francisco, CA, USA, March 2013.
- [129] Michael Wahler. Using OCL to interrogate your EMF model. *IBM, August*, 2004.
- [130] Stefan Walraven, Dimitri Van Landuyt, Eddy Truyen, Koen Handekyn, and Wouter Joosen. Efficient customization of multi-tenant Software-as-a-Service applications with service lines. *Journal of Systems and Software*, 91:48 – 62, 2014.
- [131] Stefan Walraven, Eddy Truyen, and Wouter Joosen. Comparing PaaS offerings in light of SaaS development. *Computing*, 96(8):669–724, August 2014.
- [132] Craig D Weissman and Steve Bobrowski. The design of the force. com multi-tenant internet application development platform. In *Proceedings of the 2009*

- ACM SIGMOD International Conference on Management of Data*, pages 889–896. ACM, 2009.
- [133] Jules White, Douglas C Schmidt, and Aniruddha Gokhale. Simplifying autonomic enterprise Java bean applications via model-driven development: A case study. In *International Conference on Model Driven Engineering Languages and Systems*, pages 601–615. Springer, 2005.
- [134] Zhang Xuesong, Shen Beijun, Tang Xucheng, and Chen Wei. From isolated tenancy hosted application to multi-tenancy: Toward a systematic migration method for web application. In *2010 IEEE International Conference on Software Engineering and Service Sciences*, pages 209–212, July 2010.
- [135] Haitham Yaish, Madhu Goyal, and George Feuerlicht. Elastic extension tables for multi-tenant cloud applications. *International Journal of Computer Science and Information Security*, 2016.
- [136] Carlo Zaniolo. Database relations with Null values. *Journal of Computer and System Sciences*, 28(1):142–166, 1984.

Appendix A

Comparing Data Storage Services of Cloud Providers

This section describes characteristics of different types of data storage offered by major cloud service providers, namely, Alibaba, [AWS](#), Azure, and Google. The first three tables present blob storage, where the rest three tables discuss non-relational databases of each cloud service provider.

Table A.1: Characteristics of Alibaba Object Storage Service ([OSS](#))

Feature	Alibaba OSS
Unit of Deployment	Bucket
Deployment Identifier	Globally unique key
File System Emulation	Limited
Object Metadata	Yes
Object Versioning	No
Object Lifecycle Management	No
Update Notifications	No
Service Classes	Standard Storage Infrequent Access Storage Archive Storage
Deployment Locality	Regional
Encryption at rest	Yes

Table A.2: Characteristics of Amazon S3

Feature	Amazon S3
Unit of Deployment	Bucket
Deployment Identifier	Globally unique key
File System Emulation	Limited
Object Metadata	Yes
Object Versioning	Yes
Object Lifecycle Management	Yes
Update Notifications	Event notifications
Service Classes	Standard Reduced Redundancy Infrequent Access Amazon Glacier
Deployment Locality	Regional
Encryption at rest	Yes

Table A.3: Characteristics of Azure Blob Storage

Feature	Azure Blob Storage
Unit of Deployment	Container
Deployment Identifier	Account-level unique key
File System Emulation	Limited
Object Metadata	Yes
Object Versioning	Manual
Object Lifecycle Management	No
Update Notifications	No
Service Classes	Redundancy Levels: Locally Redundant Storage Zone-redundant Storage Geo-redundant storage Read-access-geo-redundant storage Tiers: Hot and Cool
Deployment Locality	Zonal and regional
Encryption at rest	Requires enabling

Table A.4: Characteristics of Google Cloud Storage

Feature	Google Cloud Storage
Unit of Deployment	Bucket
Deployment Identifier	Globally unique key
File System Emulation	Limited
Object Metadata	Yes
Object Versioning	Yes Requires manually enabling
Object Lifecycle Management	Yes
Update Notifications	Object change notification
Service Classes	Durable Reduced Availability Nearline Coldline Redundancy Levels: Region Multi-regional Standard
Deployment Locality	Regional and multi-regional
Encryption at rest	Yes

Table A.5: Characteristics of Alibaba Table Store

Feature	Alibaba Table Store
Object category	Table
One object	Data
Individual data for an object	Attribute
Unique ID for an object	Primary Key
Data types for unique ID	String Integer Binary
Secondary Index	No
Data Types	Integer, String, Boolean, and Double
Read Consistency	Eventual Consistency (Default) Strong Consistency
Throughput Capacity for Reads and Writes	Must be specified when creating a table
Auto Scaling	Upper and lower limits for read and write capacity units must be defined
Auto Back-up	Across different servers in different racks
Auto Data Partitioning	Yes
Object-centric support for SDK	No
Read Operations	Single a single row operation
Range read operation	
Batch Transactions	Up to 100 items per transaction
Versioning	N/A
Limitations	10 instances of databases can be created under an Alibaba Cloud user account 64 tables per instance
Server-side Encryption at rest	Manual
Update Notification	N/A
Batch Operations	Write: Up to 200 data to a table Read: Up to 100 data to a table
Query	Retrieves all items that have a specific partition key.
Limitations	Item size is up to 400KB
Lifecycle Management	N/A
Geo-replication	N/A

Table A.6: Characteristics of Amazon DynamoDB

Feature	Amazon DynamoDB
Object category	Table
One object	Item
Individual data for an object	Attribute
Unique ID for an object	Partition Key Sort Key Combination of both
Data types for unique ID	String Number Binary
Secondary Index	Two kinds of indexes: 1. Global secondary index 2. Local secondary index Up to 5 of each per table
Data Types	Scalar Types: Number, String, Binary, Boolean, and null Document Types: List, Map (e.g., JSON) Set Types: sets of Number, String or Binary Values
Read Consistency	Eventual Consistency (Default) Strong Consistency
Throughput Capacity for Reads and Writes	Must be specified when creating a table
Auto Scaling	Upper and lower limits for read and write capacity units must be defined
Auto Back-up	Across multiple availability zones
Auto Data Partitioning	Yes
Object-centric support for SDK	Yes
Read Operations	Query only items with composite primary key Scan
Batch Transactions	Up to 100 items per transaction
Versioning	Need to be enabled
Limitations	256 tables per region The maximum item size is 400KB
Server-side Encryption at rest	Manual
Update Notification	With Stream record and AWS Lambda
Batch Operations	Write: Up to 25 items to a table Read/Delete: Up to 100 items from one or more table
Query	Retrieves all items that have a specific partition key. Table must have a composite key Sort by a sort key
Limitations	Item size is up to 400KB
Lifecycle Management	Yes
Geo-replication	Manual

Table A.7: Characteristics of Azure Table Storage

Feature	Azure Table Storage
Object category	Table
One object	Entity
Individual data for an object	Property
Unique ID for an object	Partition Key Row Key
Data types for unique ID	String
Secondary Index	No
Data Types	Byte array, Boolean, DateTime, Double
Read Consistency	Optimistic Concurrency (Default) Pessimistic Concurrency Last write wins
Throughput Capacity for Reads and Writes	Fixed: 1KB entities up to 2000 entities per second
Auto Scaling	Yes
Auto Back-up	According to redundancy level
Auto Data Partitioning	Yes
Object-centric support for SDK	Yes
Read Operations	Query Query Scan Scan
Batch Transactions	Entity Group Transactions Up to 100 entities with the same partition key
Versioning	Auto (Timestamp)
Limitations	The maximum entity size is 1MB Up to 252 properties per entity
Server-side Encryption at rest	Manual
Update Notification	No
Batch Operations	No
Query	No
Limitations	Entity size is up to 1 MB
Lifecycle Management	No
Geo-replication	Auto

Table A.8: Characteristics of Google Cloud Datastore

Feature	Google Cloud Datastore
Object category	Kind
One object	Entity
Individual data for an object	Property
Unique ID for an object	Key
Data types for unique ID	String Numeric ID (Auto-generated)
Secondary Index	No
Data Types	String, DateTime, Int, Double, Boolean, Key, Geopoint, Array, Embedded entity, Null, Text
Read Consistency	Strong consistency for lookups by key and ancestor queries Eventual consistency for other queries
Throughput Capacity for Reads and Writes	N/A
Auto Scaling	Yes
Auto Back-up	Regions/Zones
Auto Data Partitioning	Yes
Object-centric support for SDK	Third party APIs
Read Operations	Query
Batch Transactions	Yes Up to 25 entity groups
Versioning	Requires enabling
Limitations	The maximum entity size is 1MB
Server-side Encryption at rest	Auto
Update Notification	No
Batch Operations	Yes
Query	Filters can be applied to any indexed property Sort order by any indexed property
Limitations	Maximum entity size is ~1MB Maximum transaction size is ~10MB
Lifecycle Management	No
Geo-replication	Auto

Appendix B

Comparing Data Storage APIs of Cloud Providers

This section characterizes differences in using [APIs](#) of available data storage solutions offered by major cloud service providers, namely, Alibaba, [AWS](#), and Azure. Section [B.1](#) describes relational databases, Section [B.2](#) discusses non-relational databases, and Section [B.3](#) illustrates blob storage of each cloud service provider.

B.1 Relational Databases

Deploying and running a relational database require manually creating a database instance and getting configuration information of the created database to programmatically interact with it through the application. Manual creation process differs based on each cloud service provider, while the application code remains the same since it is provided by [JDBC API](#). The following lists the launching process of a database instance using the web user interface of each cloud service provider.

B.1.1 Alibaba ApsaraDB

Creating a relational database instance using Alibaba Cloud console:

1. Choose **ApsaraDB for [RDS](#)** from **Products**.
2. Choose **Create Instance**.
3. Specify **Region**, **Database ([DB](#)) Engine**, **Version**, **Edition** (Availability), **Zone**, **Network Type**, **Instance Type**, **Capacity**, **Duration** and **Quality**.

ApsaraDB offers MySQL, Microsoft [SQL](#) Server, PostgreSQL, PPAS, and MariaDB TX.

B.1.2 Amazon **RDS**

Creating a relational database instance **AWS** console:

1. Choose **Amazon **RDS**** from **Services**.
2. Select the region from the dropdown list in the right top corner.
3. Choose **Instances** from the navigation panel.
4. Choose **Launch **DB** Instance**.
5. Choose a database engine from the available engine options.
6. Choose a use case for your database engine.
7. Specify details of the database engine.

Amazon **RDS** offers Amazon Aurora, MySQL, MariaDB, PostgreSQL, Oracle, and Microsoft **SQL** Server.

B.1.3 Azure **SQL** Databases

Creating a relational database instance using Azure portal:

1. Create a **SQL** server (logical server) on Azure portal.
2. Specify **Server name**, **Server admin login**, **Password**, **Subscription**, **Resource group** and **Location**.
3. Choose **SQL** database from the left menu panel.
4. Choose **+Add** in the left top corner.
5. Specify **Database name**, **Subscription**, **Resource group**, **Select source**, **Server**, **SQL** elastic pool, **Pricing tier**, and **Collation**.

Azure offers Microsoft **SQL** Server only.

B.1.4 Creating connection and interacting with a database using **JDBC** **API**

Once a database engine is launched, it can be manipulated using **JDBC** **API**. The **API** requires configuration information and provides a set of operations that are described below:

1. Specify database name, username, password, hostname (endpoint) and port to your database engine.

2. Specify your database engine.
3. Initialize a connection to **SQL** database.
4. **CRUD** operations:
 - (a) Insert a data.
 - (b) Batch insert.
 - (c) Select a list of data.
 - (d) Select a data.
 - (e) Update a data.
 - (f) Delete a data.
 - (g) Delete a list of data.

B.2 Non-relational Databases

All cloud service providers require security credentials to interact with a non-relational storage when using Java **API**. Obtaining security credentials is a manual process and differs in each cloud service provider. Among analyzed cloud service providers, **AWS** and Azure allow to programmatically perform storage related operations such as creating a table and **CRUD** operations. Interestingly, Alibaba firstly requires manual effort to create an instance of a Table Store, and then enables using **API** to perform **CRUD** operations.

B.2.1 Alibaba Table Store Service

Manually creating a non-relational database using Alibaba Cloud console:

1. Choose **Table Store** from **Storage & Content Delivery Network** (**CDN**) category in **Products**.
2. Select the region from the dropdown list in the left top corner near **Products** menu.
3. Choose **Create Instance** to create a table store instance.
4. Specify **Instance Name**, **Instance Type** and **Instance Description**.
5. Choose your table store instance from the list.

Manipulating a non-relational database using Alibaba Cloud Java **SDK**:

1. Specify **Access Key Id** and **Access Key Secret**.
2. Initialize a connection to Table Store.
3. Create a table.
4. Perform operations (Table Store supports only partition key):
 - (a) Save an item.
 - (b) Save a list of items.
 - (c) Get an item by a partition key.
 - (d) Delete an item by a partition key.
 - (e) Delete a list of items.

B.2.2 **AWS** DynamoDB

Creating a non-relational database and manipulating it using **AWS SDK** for Java:

1. Specify access key and secret key.
2. Specify region where you would store your data.
3. Initialize a connection to DynamoDB.
4. Create a table.
5. Perform operations:
 - (a) Save an item.
 - (b) Save a list of items.
 - (c) Get a list of items.
 - (d) Delete a list of items.
 - (e) Get an item by a partition key.
 - (f) Delete an item by a partition key.
 - (g) Get an item by a composite **PK** key.
 - (h) Delete an item by a composite key.

¹A composite key consists of both partition and row keys

B.2.3 Azure Table Storage

Creating a non-relational database and manipulating it using Azure Storage [SDK](#) for Java:

1. Specify storage account name and key.
2. Region is specified within storage account.
3. Initialize a connection to Table Storage.
4. Create a table.
5. Perform operations (Table Storage supports only composite key):
 - (a) Save an item.
 - (b) Save a list of items.
 - (c) Get a list of items.
 - (d) Get an item by a composite key.
 - (e) Delete an item by a composite key.
 - (f) Delete a list of items by a partition key.
 - (g) Delete an item by a composite key.
 - (h) Delete a list of items.

B.3 Blob Storage

As with non-relational storage, blob storage services of all cloud providers also require manually obtaining security credentials to interact with them.

B.3.1 Alibaba [OSS](#)

Manipulate object storage using Alibaba [OSS](#) Java [SDK](#):

1. Specify Access Key Id and Access Key Secret.
2. Specify region where you would store your data.
3. Initialize a connection to [OSS](#).
4. Create a bucket.
5. Perform operations:

- (a) Upload a blob.
- (b) Retrieve a blob.
- (c) Retrieve a list of blobs.
- (d) Delete a blob.

B.3.2 Amazon **S3**

Manipulate Amazon **S3** using **AWS SDK** for Java:

1. Specify access key and secret key.
2. Specify region where you would store your data.
3. Initialize a connection to **S3**.
4. Create a bucket.
5. Perform operations:
 - (a) Upload a blob.
 - (b) Retrieve a blob.
 - (c) Retrieve a list of blobs.
 - (d) Delete a blob.

B.3.3 Azure Blob Storage

Manipulate Azure Blob Storage using Java Storage **SDK**:

1. Specify storage account name and key.
2. Region is specified within storage account.
3. Initialize a connection to Blob Storage.
4. Create a bucket.
5. Perform operations:
 - (a) Upload a blob.
 - (b) Retrieve a blob.
 - (c) Retrieve a list of blobs.
 - (d) Delete a blob.

Appendix C

Comparing Meta-modeling Environments

In the DSL development, a meta-modeling environment is needed for the meta-model specification. Ecore, GOPPRR, and Domain Model Definition (DMD) are examples of current meta-modeling environments. This section, describes and compares these environments to choose the most suitable one for CadaML implementation.

Ecore is a meta-modeling environment offered by EMF to create and define meta-models. Fundamentally, Ecore is a subset of UML Class diagrams, and it allows to define *EClass*, *EAttribute*, and *EReference*. *EClass* represents a class with attributes and references. *EAttribute* is a fundamental data in a class which has a name and a type. *EReference* defines a relationship between two classes, and it can be represented as a compartment in a target class. An Ecore model is a root object that contains packages, where each package consists classes with attributes and their references.

DMD is a part of the MSDK tool suite to specify meta-models for modeling languages designed for Visual Studio. Similarly in Ecore, DMD adopts UML Class diagram principles, but it uses different terminology, such as *Domain Class*, *Property*, and *Domain Relationship*, that can be associated with *EClass*, *EAttribute*, and *EReference*, respectively. DMD also allows setting a domain class as a compartment of another class.

In the meantime, GOPPRR is a set of metatypes provided by MetaEdit+. It is an acronym from *Graph*, *Object*, *Relationship*, *Role*, *Port* and *Property* metatypes. A *graph* is a root metatype that contains objects, relationships, roles and their bindings. In turn, an *object* is an element in a graph, where a *relationship* is a connection between two or more objects that are attached using *roles*. Moreover, a specific part of an object can be specified as a *port* to which a role can connect. Finally, a *property* is a characteristic to describe objects.

Despite the differences in terminology to define meta-models, all of the described

meta-modeling environments use a common core set of meta-modeling constructs derived from UML Class diagrams. Therefore, they offer similar capability to design meta-models. Nevertheless, Ecore has the following advantages over the remaining meta-modeling environments. Firstly, Ecore is an open source environment for Eclipse IDE and it does not require purchasing any supplementary software to exploit it. Second, Ecore is used by major DSL development and deployment frameworks such as EMF, GMF, and Epsilon. This enables meta-model composition and portability, hence, a meta-model created in Ecore using one framework can be reused in another one. Finally, there are plenty of additional tools tailored for Ecore to support model validation and code generation.

Appendix D

Data Model for Alibaba Table Store

The following listing provides an excerpt of ‘*Artist*’ data model for Alibaba Table Store that is generated by CadaML.

```
@Entity
public class Artist implements ArtistInterface {
    @Id
    @Column(name="ArtistId", columnDefinition="Partition Key")
    private String artistId;
    @Id
    @Column(name="ArtistName", columnDefinition="Row Key")
    private String artistName;
    @Column(name="Genres")
    private String genres;
    @Column(name="Biography")
    private String biography;
    @Column(insertable=false)
    private List<AlbumInterface> albums;

    ...

    public String getArtistId() { return artistId; }
    public void setArtistId(String artistId) { this.artistId = artistId; }

    public String getArtistName() { return artistName; }
    public void setArtistName(String artistName) { this.artistName = artistName; }

    public String getGenres() { return genres; }
    public void setGenres(String genres) { this.genres = genres; }

    public String getBiography() { return biography; }
    public void setBiography(String biography) { this.biography = biography; }

    public List<AlbumInterface> getAlbums () { return albums; }
    public void setAlbums(List<AlbumInterface> albums) { this.albums = albums; }
}
```

Listing 11: ‘*Artist*’ data model for Table Store generated by CadaML

Appendix E

Test Cases to Verify Multi-tenancy Implementation

Table E.1: The expanded test cases with their status for *Process Definition* entity

	Process Definition	Status
TC1 (a)	Retrieve a process definition	Passed
TC1 (b)	Retrieve a process definitions with custom attributes	Passed
TC1 (c)	Retrieve a process definition with corresponding task definitions and processes	Passed
TC1 (d)	Retrieve a list of process definitions	Passed
TC2 (a)	Store a process definition	Passed
TC2 (b)	Store a process definition with custom attributes	Passed
TC2 (c)	Store a process definition with corresponding task definitions and processes	Passed
TC2 (d)	Store a list of process definitions	Passed
TC3 (a)	Update a process definition	Passed
TC3 (b)	Update a process definition and its custom attributes	Passed
TC3 (c)	Update a process definition and corresponding task definitions and processes	Passed
TC3 (d)	Update a list of process definitions	Passed
TC4 (a)	Delete a process definition	Passed
TC4 (b)	Delete a process definition with corresponding custom attributes	Passed
TC4 (c)	Delete a process definition and corresponding task definitions and processes	Passed
TC4 (d)	Delete a list of process definitions	Passed

Table E.2: The expanded test cases with their status for *Task Definition* entity

	Task Definition	Status
TC1 (a)	Retrieve a task definition	Passed
TC1 (b)	Retrieve a task definition with custom attributes	Passed
TC1 (c)	Retrieve a task definition with corresponding tasks	Passed
TC1 (d)	Retrieve a list of process definitions	Passed
TC2 (a)	Store a task definition	Passed
TC2 (b)	Store a task definition with custom attributes	Passed
TC2 (c)	Store a task definition with corresponding tasks	Passed
TC2 (d)	Store a list of task definitions	Passed
TC3 (a)	Update a task definition	Passed
TC3 (b)	Update a task definition and its custom attributes	Passed
TC3 (c)	Update a task definition and corresponding tasks	Passed
TC3 (d)	Update a list of task definitions	Passed
TC4 (a)	Delete a task definition	Passed
TC4 (b)	Delete a task definition with corresponding custom attributes	Passed
TC4 (c)	Delete a task definition and corresponding tasks	Passed
TC4 (d)	Delete a list of task definitions	Passed

Table E.3: The expanded test cases with their status for *Process* entity

	Process	Status
TC1 (a)	Retrieve a process	Passed
TC1 (b)	Retrieve a process with custom attributes	Passed
TC1 (c)	-	-
TC1 (d)	Retrieve list of processes	Passed
TC2 (a)	Store a process	Passed
TC2 (b)	Store a process with custom attributes	Passed
TC2 (c)	-	-
TC2 (d)	Store a list of processes	Passed
TC3 (a)	Update a process	Passed
TC3 (b)	Update a process and its custom attributes	Passed
TC3 (c)	-	-
TC3 (d)	Update a list of processes	Passed
TC4 (a)	Delete a process	Passed
TC4 (b)	Delete a process with corresponding custom attributes	Passed
TC4 (c)	-	-
TC4 (d)	Delete a list of processes	Passed

Table E.4: The expanded test cases with their status for *Task* entity

	Task	Status
TC1 (a)	Retrieve a task	Passed
TC1 (b)	Retrieve a task with custom attributes	Passed
TC1 (c)	-	-
TC1 (d)	Retrieve a list of tasks	Passed
TC2 (a)	Store a task	Passed
TC2 (b)	Store a task with custom attributes	Passed
TC2 (c)	-	-
TC2 (d)	Store a list of tasks	Passed
TC3 (a)	Update a task	Passed
TC3 (b)	Update a task and its custom attributes	Passed
TC3 (c)	-	-
TC3 (d)	Update a list of tasks	Passed
TC4 (a)	Delete a task	Passed
TC4 (b)	Delete a task with corresponding custom attributes	Passed
TC4 (c)	-	-
TC4 (d)	Delete a list of tasks	Passed