# CadaML: A Modeling Language for Multi-Tenant Cloud Application Data Architectures

Assylbek Jumagaliyev, Yehia Elkhatib
School of Computing and Communications, Lancaster University, United Kingdom
{*i.lastname*}@lancaster.ac.uk

***Abstract*—Multi-tenancy is used for efficient resource utilization when cloud resources are shared across multiple customers. In cloud applications, the data layer is often the prime candidate for multi-tenancy, and usually comprises a combination of different cloud storage solutions such as relational and non-relational databases, and blob storage. Each of these storage types is different, requiring its own partitioning schemes to ensure tenant isolation and scalability. Current multi-tenant data architectures are implemented mainly through manual coding techniques that tend to be time consuming and error prone. As an alternative, we propose a domain-specific modeling language, CadaML, that provides concepts and notations to model a multi-tenant data architecture in an abstract way. CadaML also provides tools to validate the data architecture and automatically produce application code to implement said architecture.**

***Index Terms*—Domain-Specific modeling, Cloud Computing, Multi-tenancy, Software Evolution, Code Generation**

## I. Introduction

*Multi-tenancy* is an architectural pattern where a single instance of an application and its associated data stores serve multiple tenants [1]. In this context, a *tenant* is a group of users that belongs to an organization who has access with specific privileges to an application [2]. Multi-tenancy is commonly adopted in cloud environments as it enables efficient resource utilization, and leads to lower operation and maintenance costs through consolidation [3].

Introducing multi-tenancy affects development and evolution overheads at all layers of the application structure, and the data layer is no exception. Multi-tenancy at the data layer requires a data architecture to ensure isolation of tenant data and requests, on top of the ability to scale. The data architecture typically also needs to be extensible to support tenant-specific customizations.

A further complication in the case of multi-tenant cloud applications is the tendency to store data in different storage types [4]–[6], *i.e.,* relational and non-relational databases, and blob storage. These are conceptually diverse, with each having its own partitioning and extensibility approaches to support multi-tenancy. Thus, building a data architecture that maximizes resource sharing with the optimal degree of isolation requires developers to address multi-tenancy challenges and to find a balance between several architectural trade-offs.

Domain-Specific Languages (DSLs) have been long proposed to address multi-tenancy concerns, specifically to generate and/or maintain cloud implementations. DSLs are concise, simple and expressive languages that aim to address problems of a specific domain through high level and abstract notions [7]. There have been some successful approaches to describe deployment configurations, provisioning and management of cloud services (*e.g.,* [8]–[11]), but multi-tenancy at the data layer has not yet been appropriately addressed. For example, CloudDSL [10] allows the specification of deployment information for a data layer on different cloud storage services, but the actual data architecture and multi-tenancy management patterns are not supported. Other model-based techniques [12], [13] capture multi-tenancy patterns in the data layer as a part of the application structure, but offer no support for implementation of the expressed multi-tenancy model.

In this paper, we propose CadaML: a modeling language for the design and implementation of cloud application data architectures. CadaML is specifically designed to support multi-tenant data architectures on different cloud storage types. It provides graphical and automated support to build a data architecture as a model, validate the model, and generate the appropriate source code including various ways of managing multi-tenancy.

The paper is the first to propose a domain-specific modeling language for building multi-tenant data architectures of cloud applications. CadaML is graphical and does not require any syntax to be learned. It includes validation rules to ensure reliable model-to-code transformation.

The rest of the paper is organized as follows. Section II describes the problem domain and gives an overview of different cloud data storage solutions with their partitioning approaches. Section III explains the meta-model design of CadaML, while Section IV details its implementation details. We compare against related work in Section V, and finally draw conclusions and outlines future work in Section VI.

## II. Problem Description

When developing a multi-tenant application, one of the highest priorities 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 [14], [15]. However, cloud applications have a variety of data storage requirements and are often served best by a combination of multiple data storage options [4], [5], [16]. These storage options differ in storing and organizing data. Moreover, each storage option has its own partitioning and extensibility approaches to support multi-tenancy.

### A. Data Store Types

In this paper, we focus on the Platform-as-a-Service (PaaS) provisioning level of cloud computing as the one most common with developers [17]. We now describe PaaS storage types, and illustrate the differences in storing and organizing data with each type.

- *Relational databases* are appropriate for structured data with a well-defined schema. Data is organized in tables of rows and columns, with a *primary key* identifying each row. Relationships are strongly defined in the data model.
- *Non-relational databases* (also called *NoSQL*) mainly support key-value store models, and are suitable for flexible data schemas. A *partition key* determines the partition in which data will be stored, and a *row key* identifies data within each partition.
- *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*.

### B. 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. Each cloud storage type has its own partitioning techniques. In this subsection, we summarize these techniques after analyzing academic and industrial literature, as well as cloud provider guidance.

In general, relational databases can be partitioned using: (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 (with a tenant identifier in the table name, or a separate schema can be used for each tenant); or (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* 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.

Given these varying partitioning schemes, manually implementing a multi-tenant data architecture can be highly time-consuming and error-prone 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, *cf.* [8], [9], [12], [13], [18]. However, existing approaches 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, etc.

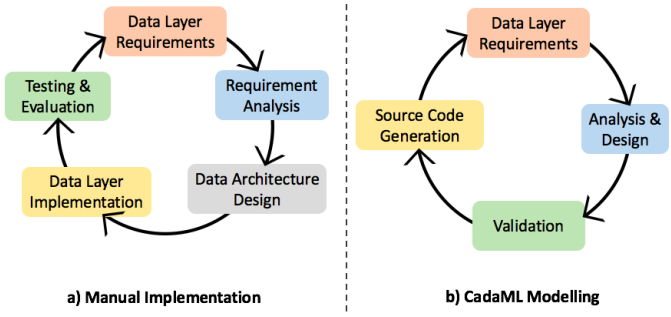


Figure 1. The implementation process of the manual approach and CadaML.

## III. Design

To address the limitations of previous work, we present CadaML to design a multi-tenant data architecture, and generate source code that is suitable for different cloud storage types that are required.

### A. Objectives

Through this work we aim to achieve the following:

- Describe an abstract multi-tenant data architecture.
- Generate data layer code to implement multi-tenancy over different storage types.
- Reduce the development effort during the implementation of a multi-tenant cloud data architecture.

### B. Overview

A multi-tenant data architecture design varies depending on the type of storage, and it even differs from its implementation. Typically, the data architecture is implemented manually by following guidance and patterns from cloud providers. A traditional manual implementation process covers the following five steps as illustrated in Figure 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 designed using database modeling tools, *e.g.,* Database Deployment Manager, Database Workbench, and ER/Studio;

(iv) developers implement a data access layer from the data architecture model; and

(v) developers systematically discover and debug errors in the code.

In this manual approach, whenever the data layer requirements change, developers have to go through all these steps and modify the existing code. This process is usually time-consuming and error prone.

To ease the development process of multi-tenant data architectures, we propose CadaML that allows to describe a data

architecture in an abstract level by hiding the implementation details of different cloud storage types. As shown in Figure 1b, a data layer implementation workflow using CadaML involves the following four steps:

(i) First, as in the manual approach, data layer requirements are captured;

(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.

In this scenario, changes in the requirements can be directly reflected in the model, thus, code is generated from the model.

Compared to the manual approach, CadaML automates the data access layer implementation by generating source code from the data architecture model. In addition, CadaML eliminates testing and evaluation phase through the validation tool that handles errors at the model level before generating any artefact from the model.

### C. The Meta-model

Modeling languages are defined in a meta-model that describes language elements and relationships among them [7]. The concepts and notations of CadaML should correspond to terminology that cloud data layer architects and developers are familiar with. Thus, the meta-model has been created by thoroughly analyzing common storage types characteristics of different cloud providers (namely, Alibaba Cloud, Amazon Web Services (AWS), Google Cloud Platform, and Microsoft Azure), features of existing modeling languages (*e.g.,* [12], [13]) that support cloud application development, and peculiarities of cloud data storage partitioning techniques that were described in academic and industrial papers (*e.g.,* [12]–[14]).

Figure 2 presents the concepts of the CadaML meta-model and the interrelations therein. The main element of the meta-model is *DatabaseDiagram* that represents a diagram in a graphical editor where a developer designs a data architecture. A diagram may include *NoSQL Database*, *SQL Database* and *Object Storage*. *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*. The relationships among tables are represented by *NoSQL reference*, where *source table* and *target table* parameters refer to multiplicity of relationships (*i.e., ZERO*, *ONE*, and *MANY*) between tables.

Relational databases are expressed by *SQL database*. *SQL-Partition* of a relation database is classified according to partitioning schemes that were described in Section II-B. 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.

*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 II-B 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.

### D. Multi-tenancy Management

Multi-tenancy is supported by capturing data segmentation patterns of different data storage solutions in the meta-model. Hence, developers can specify a desired partitioning scheme for each storage type while modeling a data architecture. Partitioning schemes are defined at bucket level for blob storage, and database instance level for relational and non-relational databases. Based on defined schemes, CadaML produces corresponding implementation of the data access layer. For example, in a shared bucket tenant isolation is implemented by appending the tenant identifier to the blob key when uploading and retrieving blobs.

The described meta-model offers benefits above existing work, which mostly capture cloud services for defining provisioning and deployment configurations of application components. A few meta-models includes multi-tenancy patterns for relational databases but do not comprise concepts to model a cloud data architecture. In contrast, CadaML allows to explicitly model a data architecture for different cloud storage types with partitioning schemes, in line with current best practices in this space, *e.g.,* [19].

## IV. Implementation

We decided to implement CadaML as a graphical modeling language 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 [20]. Finally, visualization of a model allows non-developers to get

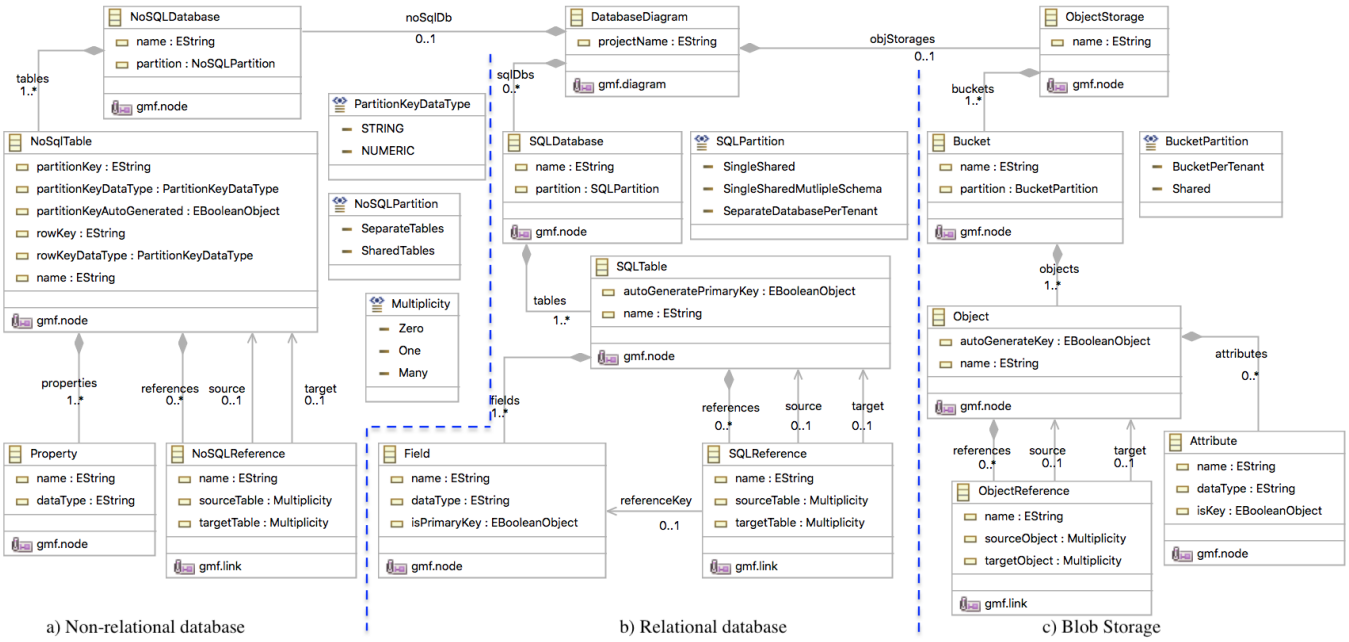


Figure 2. The meta-model

an overview of a data architecture and intuitively develop an understanding of the data layer design.

The implementation was done in Java using a total of 18,757 lines of code. This is broken down to 14,709 for the implementation of the meta-model, 83 for adjusting the graphical editor, 299 for validation, and 3,666 for code generation. More concretely, the meta-model was created using the Emfatic syntax [21], and annotated using EuGENia [22] which is then transformed into a concrete Graphical Modeling Framework editor in the Eclipse IDE.

### A. Validation Rules and Constraints

A key advantage of CadaML is that the model is validated before generating other artefacts from it. Constraints and validation rules are enforced at the level of the model that can handle many kinds of errors. For example, names of storage type elements (*e.g.,* tables, fields in a table, buckets, objects within a bucket) in data architecture must be unique valid identifiers. As another example, non-relational tables must have both partition and row keys, while relational tables must have primary keys. In CadaML, validation rules and constraints are defined in Epsilon Validation Language (EVL) [22]. Most of the constraints are written based on characteristics of cloud storage types and principles of the Java programming language.

### B. Code Generation

In order to increase developer productivity and code reliability, CadaML includes a code generator that automatically transforms a model created by a developer to executable Java code for AWS. The code generator is written in Epsilon Generation Language [22], and it produces 1) data models, 2) storage context classes, 3) Java interfaces, and 4) cloud specific classes for each storage type. All code that is specific to a storage type is located in different packages. In addition, the generated code decouples the data access logic from other layers of the application. This separation provides ease of code maintenance, and allows to independently scale the data layer.

Blobs, relational and non-relational tables are each transformed into a data model, a Java class with appropriate getters and setters. For relational and non-relational tables, data models are annotated using Java Persistence API and DynamoDB Java Annotations, respectively. The annotations are used to map object fields to actual attribute names in database tables. A storage context class contains storage related fields, such as provider name, storage credentials, region, and replication to initialize a storage connection. In the meantime, a Java interface contains generic method signatures that are further implemented by provider-specific classes.

Blob storage interface contains methods to initialize storage, create a bucket, upload a blob, retrieve a single blob, retrieve a list of blobs and delete a blob. While, the relational database interface includes insert, select, update and delete methods. Finally, non-relational database interface has methods to create a table, save an item in a table, retrieve an item, and delete an item from a table. It is worth noting that cloud provider specific classes implement methods in a generic way that work on different data models.

## V. Related Work

An XML-based modeling language [23] is provided by Topology and Orchestration Specification for Cloud Applications (TOSCA) to define application components and their relationships. Similarly, CloudML-SINTEF [24] is proposed as a standalone DSL 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. Another XML-based modeling language, CloudML-UFPE [25], allows the description of the data layer in terms of cloud resources and services with their requirements. Using StratusML [26], a developer can specify a storage group that will be used to persist application's data and describe different data partitioning strategies. Similar approaches are adopted by Holmes [27] and Blueprint [28]. Multi-tenancy at the data layer in terms of quality requirements has been captured by leveraging Orthogonal Variability Modeling Language [13]. In [12], feature modeling is used to express data segmentation schemes for each functional part of an application that interacts with the data layer. Tenants select partitioning options, and a configuration information is generated based on the selected options. A modelling language is proposed in [29] to handle non-functional requirements across multiple providers, but not to handle data layer evolution.

These modeling languages automate the software provisioning and migration by generating deployment specification models. However, none of them allows the capture of multi-tenancy patterns in the data layer, nor produce the data access code from the model (only data definition scripts, in the case of CloudML-SINTEF).

Although most real-world cloud applications use a combination of different cloud storage types, current works are geared mainly towards only partitioning relational databases; *e.g.,* [30], [31]. Employing different cloud storage types, and accordingly dealing with their conceptual differences and partitioning implementation peculiarities, is something that is not addressed in the state of the art [32].

## VI. Conclusion

Introducing multi-tenancy at the data layer makes for a relatively laborious and error-prone development process. To overcome this, we present a domain-specific modeling language CadaML that provides support to create an abstract data architecture model, as well as automated model-to-text transformation to interpret the model and generate appropriate source code for different cloud data storage types. Along with its model validation support, CadaML relieves developers from the need to create their own multi-tenant-safe implementation and the details of managing different storage types, and instead allows them to focus on their abstract data architecture model. CadaML is a graphical language so no syntax needs to be learned. For future work, we will evaluate CadaML by applying it on the data layer of real cloud applications. We will compare it against manual implementation methods in terms of development overhead and code reliability.

## References


[1] C.-P. Bezemer and A. Zaidman, "Multi-tenant SaaS applications: Maintenance dream or nightmare?" in *EVOL and IWPSE Workshops*, 2010.

[2] R. Krebs, C. Momm, and S. Kounev, "Architectural concerns in multi-tenant SaaS applications," in *CLOSER*, 2012.

[3] J. R. Hamilton, "On designing and deploying internet-scale services," in *LISA*, vol. 18. USENIX, 2007.

[4] S. Sakr, A. Liu, D. M. Batista, and M. Alomari, "A survey of large scale data management approaches in cloud environments," *IEEE Comm. Surveys Tuts.*, vol. 13, no. 3, 2011.

[5] T. Dykstra, R. Anderson, and M. Watson, *Building Real-World Cloud Apps with Windows Azure*. Microsoft Corporation, 2014.

[6] B. Varghese, P. Leitner, S. Ray, K. Chard, A. Barker, Y. Elkhatib, H. Herry, C.-H. Hong, J. Singer, F. P. Tso, E. Yoneki, and M. F. Zhani, "Cloud futurology," *IEEE Computer*, 2019.

[7] M. Fowler, *Domain Specific Languages*. Addison-Wesley, 2010.

[8] N. Ferry, F. Chauvel, A. Rossini, B. Morin, and A. Solberg, "Managing multi-cloud systems with CloudMF," in *Nordic Symposium on Cloud Computing & Internet Technologies*, 2013.

[9] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "TOSCA: Portable automated deployment and management of cloud applications," in *Advanced Web Services*. Springer, 2014, pp. 527–549.

[10] G. C. Silva, L. M. Rose, and R. Calinescu, "Cloud DSL: A language for supporting cloud portability by describing cloud entities," in *CloudMDE@MoDELS*, 2014.

[11] A. Bergmayr, U. Breitenbücher, O. Kopp, M. Wimmer, G. Kappel, and F. Leymann, "From architecture modeling to application provisioning for the cloud by combining UML and TOSCA," in *CLOSER*, 2016.

[12] F. Mohamed, M. Abu-Matar, R. Mizouni, M. Al-Qutayri, and Z. Al Mahmoud, "SaaS dynamic evolution based on model-driven software product lines," in *CloudCom*, 2014.

[13] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl, "Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications," in *PESOS*, 2009.

[14] F. Chong and G. Carraro, "Architecture strategies for catching the long tail," Microsoft Corporation, Tech. Rep. 479069, 2006.

[15] Y. Elkhatib, G. S. Blair, and B. Surajbali, "Experiences of using a hybrid cloud to construct an environmental virtual observatory," in *Proc. of the 3rd Workshop on Cloud Data and Platforms*, 2013.

[16] A. Elhabbash, F. Samreen, J. Hadley, and Y. Elkhatib, "Cloud brokerage: A systematic survey," *ACM Comput. Surv.*, vol. 51, no. 6, 2019.

[17] S. Walraven, E. Truyen, and W. Joosen, "Comparing PaaS offerings in light of SaaS development," *Computing*, vol. 96, no. 8, 2014.

[18] M. Abu-Matar, R. Mizouni, and S. Alzahmi, "Towards software product lines based cloud architectures," in *IC2E*, 2014.

[19] S. Strauch, V. Andrikopoulos, T. Bachmann, D. Karastoyanova, S. Passow, and K. Vukojevic-Haupt, "Decision support for the migration of the application database layer to the cloud," in *CloudCom*, 2013.

[20] G. Hogenson, G. Warren, S. Cai, A. Homer, T. Petersen, M. Jones, and M. Blome, *Modeling SDK for Visual Studio - Domain-Specific Languages*, Microsoft, 2016.

[21] C. Daly. Emfatic: A textual syntax for EMF Ecore meta-models.

[22] D. Kolovos, L. M. Rose, and R. F. Paige, *The Epsilon Book*, 2018.

[23] A. Atrey, H. Moens, G. V. Seghbroeck, B. Volckaert, and F. D. Turck, "An overview of the OASIS TOSCA standard: Topology and orchestration specification for cloud applications," IBCN-iMinds, Department of Information Technology, Tech. Rep., 2015.

[24] A. Bergmayr, A. Rossini, N. Ferry, G. Horn, L. Orue-Echevarria, A. Solberg, and M. Wimmer, "The evolution of CloudML and its applications," in *Workshop on MDE on and for the Cloud*, 2015.

[25] G. E. Gonçalves, P. Endo, M. Santos, D. Sadok, J. Kelner, B. Melander, and J.-E. Mangs, "CloudML: An integrated language for resource, service and request description for d-clouds," *CloudCom*, 2011.

[26] M. Hamdaqa and L. Tahvildari, "Stratus ML: A layered cloud modeling framework," in *IC2E*, 2015.

[27] T. Holmes, "Automated Provisioning of Customized Cloud Service Stacks using Domain-Specific Languages," in *CloudMDE*, 2014.

[28] D. K. Nguyen, F. Lelli, Y. Taher, M. Parkin, M. P. Papazoglou, and W.-J. van den Heuvel, "Blueprint template support for engineering cloud-based services," in *European Conf. on a Service-based Internet*, 2011.

[29] A. Elhabbash, Y. Elkhatib, G. S. Blair, Y. Lin, and A. Barker, "A framework for SLO-driven cloud specification and brokerage," in *6th Workshop on CrossCloud Infrastructures & Platforms*, May 2019.

[30] E. J. Domingo, J. T. Nino, A. L. Lemos, M. L. Lemos, R. C. Palacios, and J. M. G. Berbís, "Cloudio: A cloud computing-oriented multi-tenant architecture for business information systems," in *IEEE Cloud*, 2010.

[31] A. Jumagaliyev, J. Whittle, and Y. Elkhatib, "Using DSML for handling multi-tenant evolution in cloud applications," in *CloudCom*, 2017.

[32] R. Sellami, S. Bhiri, and B. Defude, "Supporting multi data stores applications in cloud environments," *IEEE Trans. Serv. Comput.*, vol. 9, no. 1, 2016.