View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Lancaster E-Prints

Using DSML for Handling Multi-tenant Evolution in Cloud Applications

Assylbek Jumagaliyev*, Jon Whittle†, Yehia Elkhatib*

*MetaLab, School of Computing and Communications, Lancaster University, UK
Email: {i.lastname}@lancaster.ac.uk
†Faculty of Information Technology, Monash University, Melbourne, Australia

***Abstract*—Multi-tenancy is sharing a single application's resources to serve more than a single group of users (*i.e.* tenant). Cloud application providers are encouraged to adopt multi-tenancy as it facilitates increased resource utilization and ease of maintenance, translating into lower operational and energy costs. However, introducing multi-tenancy to a single-tenant application requires significant changes in its structure to ensure tenant isolation, configurability and extensibility. In this paper, we analyse and address the different challenges associated with evolving an application's architecture to a multi-tenant cloud deployment. We focus specifically on multi-tenant data architectures, commonly the prime candidate for consolidation and multi-tenancy. We present a Domain-Specific Modeling language (DSML) to model a multi-tenant data architecture, and automatically generate source code that handles the evolution of the application's data layer. We apply the DSML on a representative case study of a single-tenant application evolving to become a multi-tenant cloud application under two resource sharing scenarios. We evaluate the costs associated with using this DSML against the state of the art and against manual evolution, reporting specifically on the gained benefits in terms of development effort and reliability.**

1. Introduction

Many new applications are released following the Software as a Service (SaaS) model that enables consumers to access the provider's applications running on a cloud over the Internet [1]. A SaaS application can be designed as *single-tenant* or *multi-tenant*. A *tenant* is a group of users that belongs to an organization who has access with specific privileges to an application [2]. In single-tenancy, each tenant is served by a dedicated application instance which runs on a logically isolated hardware, such as a Virtual Machine (VM) or a container. In multi-tenancy, multiple tenants share hardware and software cloud resources, while the application distinguishes the requests and data of each tenant [3]. Tenants must be able to configure and extend the application to their needs as it runs in a dedicated environment.

Multi-tenancy is generally preferred as it enables increased resource utilization, and reduces the operational and energy costs of resource provisioning and software maintenance through consolidation [4], [5], [6], [7]. These benefits encourage application providers to adopt multi-tenancy to their existing applications. However, introducing multi-tenancy affects all layers of the application structure which requires developers to address multiple challenges and find a balance between several architectural trade-offs [2]. This, consequently, increases development and maintenance complexity [8], [9]. The following are multi-tenancy concerns along with design factors that influence the architecture.

- **Configurability and extensibility:** Sharing application logic and data across different tenants requires the application to be widely configurable and extensible to cater for tenant-specific needs. This concerns different functional and non-functional aspects of the application such as adding or removing features, changing the user interface, and customizing the business logic.
- **Tenant isolation:** The application must ensure that tenants only view and edit their own data. This requirement must also ensure that tenant-specific configurations and extensions do not directly affect the application for other tenants.
- **Scalability:** A successful SaaS should be able to scale horizontally as the number of tenants changes. During horizontal scaling, cloud resources are created or deleted to match application performance requirements [10].
- **Ease of development and maintenance:** The complexity in development and maintenance processes of a multi-tenant SaaS may increase application cost, potentially leading to concerns about tenant satisfaction.

In multi-tenancy, one of the highest priorities is to create a data architecture that supports multi-tenancy requirements, and one that is also efficient and cost-effective to implement and maintain [11]. Such data architecture typically requires a partitioning scheme that enables tenant isolation and scalability. In addition, the architecture must be extensible to support customization. There are three common multi-tenant data segmentation patterns [11]: (i) separate databases where each tenant is deployed on a dedicated database instance, (ii) a shared database with separate schemas for each tenant, and (iii) a shared database with a shared schema. However,

finding the desired level of flexibility with sufficient isolation depends on technical consideration (*i.e.* scalability, customizability and development effort) and tenant requirements [9].

Abu-Matar and Whittle [7] have noted that Domain-Specific Languages (DSLs) could address such multi-tenancy concerns, specifically to generate and/or maintain cloud implementations. DSLs are concise, simple and expressive languages that are intended to address problems of a specific domain through high level and abstract notions [12]. From a software engineering viewpoint, a DSL refers to a modeling language that offers notations and concepts that can be directly manipulated by a domain expert to express a solution as a model [13]. There have been some approaches in this direction. For example, CloudDSL [14] describes different cloud services using a common cloud vocabulary. CloudML [15] is another DSL to specify provisioning, deployment, and adaptation concerns. However, these efforts specifically target provisioning issues, primarily scalability, and do not support multi-tenancy in the data layer of a cloud application.

In this paper, we address the architectural concerns during the evolution process of a single-tenant application to a multi-tenant SaaS. We present a DSML for modeling a multi-tenant database that supports tenant isolation, configurability and extensibility. We focus on the data layer as it is often the prime candidate for multi-tenancy [4] as other layers are typically stateless in cloud applications [16], [17]. The DSML provides an automated support to design a data architecture, generate source code and evolve an application's data layer. The main goal is to reduce time and effort required for development, and to improve maintainability during the evolution of multi-tenant SaaS applications. We report on a case study of applying the DSML to re-engineering a single-tenant web application to a multi-tenant SaaS, and evolving the multi-tenancy data architecture. Evaluation against other approaches shows that using the DSML brings benefits in terms of reliability and maintainability.

2. Domain-Specific Modeling Language

In this section, we present a DSML that allows a modeler to model a database architecture to support development and evolution of SaaS applications. The DSML comprises of the concepts of storage types, database partitioning models, and it supports multi-tenancy. It has been implemented as a Microsoft Visual Studio 2015 extension using the Visualization and Modeling SDK (VMSDK) [18]. VMSDK allows the creation of a meta-model, 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. In modeling languages, a meta-model describes a domain of the language, and a model is designed according to the concepts defined in it.

The DSML consists of seven domain classes and their relationships as depicted in Figure 1 and detailed as follows.

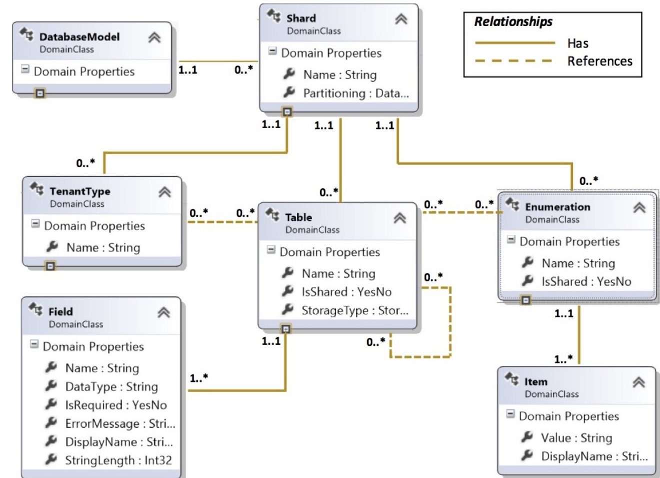

Figure 1: The meta-model of the DSML

1) The *DatabaseModel* domain class represents a model where a developer creates a database structure. A database model must have at least one shard.
2) A *Shard* is a database partition, and it can contain *TenantTypes, Tables* and *Enumerations*. The *Partitioning* property of the *Shard* can be set to one of the data models that were described in Section 1.
3) A *TenantType* describes a type of tenant in a shard. For example, there might be Standard and Premium tenant types that use different sets of tables in a shard.
4) A *Table* is a collection of *Fields*.
5) A *field* is a single piece of data with several properties such as name, data type, and isRequired.
6) An *Enumeration* is a list of *Items*.
7) The *TenantType* can reference to tables, whereas the *Table* can reference to other *Tables* or *Enumerations*.

All domain classes, except *Field* and *Item*, are mapped to their visual representations.

During modeling of a database structure, constraints and validation are important to keep the model consistent. When the model has some conflicts with the semantics of the meta-model, the DSML should notify the modeler. For example, names of database elements must be unique valid identifiers and must not be blank. In our DSML, the validation of a model was implemented using a validation framework that is provided by VMSDK. Thus, a model is checked when it is loaded or saved, and the DSML asks for confirmation if the model is invalid. The modeler decides whether to confirm or discard invalid models to be saved.

Once a database model is ready, source code is generated for each shard from the model. The code generator was implemented using T4 Text Templates [18], which is a built-in Microsoft Visual Studio tool. T4 allows to generate program code and other files according to data from the model. In our DSML, the generated code are data models in C# and a wrap-up class for establishing a database connection. Initially, data models are generated for shared entities, followed

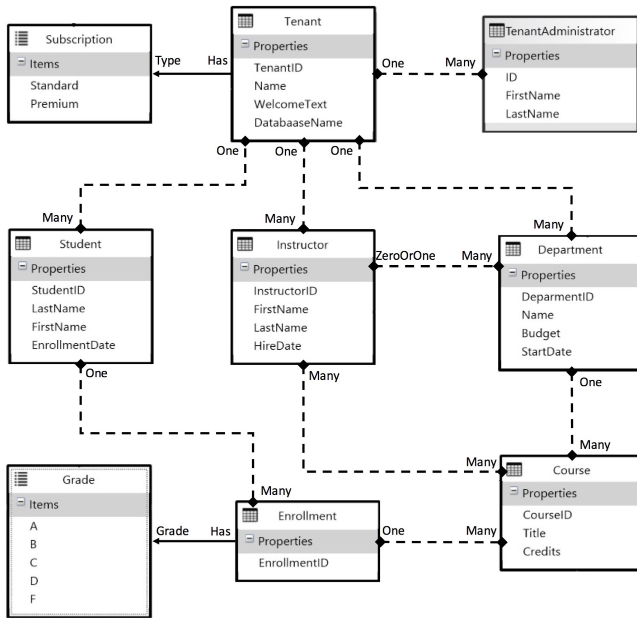

Figure 2: The data architecture of the multi-tenant Education Center application.

by data models for each tenant type. Shared and tenant type specific models are stored in different namespaces.

3. Case Study

To investigate the practical feasibility and to evaluate the utility of applying the presented DSML, we apply it to a case study where we re-engineer a single-tenant web application to introduce multi-tenancy. Further, we evolve the multi-tenant data architecture to provide a more isolated database approach for particular tenants.

The *Education Center* service has been chosen as a prototypical use case. It is a single-tenant application that can be deployed in Azure, and it was developed as a multi-tier .NET application with an ASP.NET MVC front-end and a SQL database for data store. The size of the application is about 1.5k lines of code (LOC). The application consists of presentation, business logic, and data layers. This separation helps to manage complexity during the development and evolution processes, and it also enables loose coupling between the application layers.

Three groups of users interact with the application: application administrator, instructors and students. The application administrator manages instructors, courses, and student admissions. She also enrolls students to courses, and assigns instructors to courses. Instructors mark students who are assigned to the courses they teach. Students view courses they are enrolled on and their progress on each course.

3.1. Introducing Multi-tenancy

During the adoption of multi-tenancy, we tried to minimize the number of changes in the application structure. However, several modifications were performed to isolate tenants in the database and the application. The evolution process was implemented in the following steps:

- *Authentication.* The authentication mechanism was modified to identify end users of tenants. When a user is successfully authenticated, the application looks for a tenant who owns the user from the TenantUsers table. Then, the application loads a tenant-specific user interface and connects to an appropriate database instance from the configuration information of the tenant.
- *Modeling the database using the DSML.* The database structure was modeled following the single shared database for all tenants model as depicted in Figure 2. The Tenant and TenantAdministrator entities were added to the database. The Tenant entity represents a customer with configuration information who has subscribed to the service, whilst the TenantAdministrator is a service administrator of a tenant who is responsible for managing the application. Figure 2 also shows that a tenant has departments, instructors, courses and students. A department has multiple courses, and an instructor can belong to a department. In turn, many instructors can be assigned to many courses. Finally, the data about student enrollments are stored in the Enrollments table. Once the database model has been designed, a source code is generated with a wrapper class to dynamically connect to an appropriate database.
- *Implementing tenant isolation.* To provide data isolation for each tenant, the primary keys of tables that belong to a tenant contain a globally unique identifier TenantID as a prefix, and end with a four-digit integer counter. This ensures tenants access to their data, and it also helps to minimize changes at the data layer during future evolutions of the application.
- *Modifying the business logic.* The business logic has been modified as new entities, Tenant and TenantAdmin, need to be managed by the application provider. For entities that belong to a tenant, queries for creation were modified so that a foreign key will be generated properly. In addition, all classes that interact with the data layer use the generated wrapper class to establish a connection to an appropriate database instance.

3.2. Evolution of Multi-tenant SaaS

There are several reasons that trigger application evolution in a multi-tenant application, including fixing bugs, changes in business environment, improving security and reliability, changes in existent tenant requirements, or new tenant requirements. The application should respond to such changes to maintain tenant satisfaction.

When architecting the application structure, we decided to use a single database instance shared by all tenants

as opposed to a separate database per tenant or a shared database with tenant-specific schemas. The former choice offers lower cost and more ease of management. However, as the number of tenants increases over time, the number of concurrent end users and data stored by each tenant increase as well. Moreover, some tenants may require a separate database due to privacy requirements. These scenarios require a more isolated data storage approach and need changes in the application structure.

The evolution from the single shared database model to the separate databases has been performed by changing the Partitioning property of the shard to 'Separate Database Per Tenant' mode. A dedicated database instance was created for each tenant, and a tenant-specific database name is stored in the tenant's configurations. The modeler creates a shard per tenant to define tenant-specific schemas. The overall application structure was changed, as depicted in Figure 3, based on the external configuration store pattern [10]. The configuration information of tenants is deployed in the default database. The application authenticates tenants and their end users by connecting to the default database. When the authentication is successful, the application redirects tenants and their end users to their own database instance. This provides easier management and control of configuration data. It also allows sharing configuration data across applications and application instances.

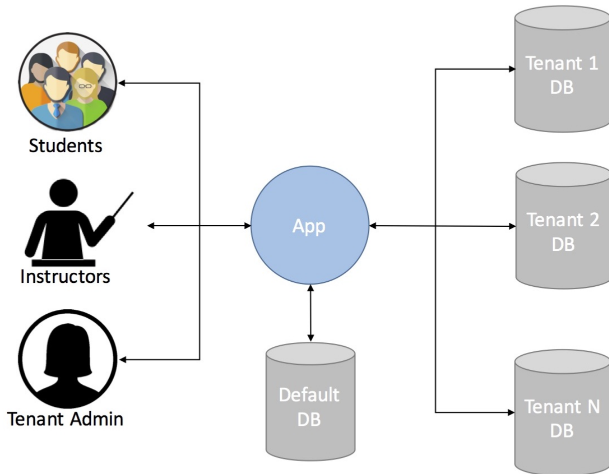

Figure 3: The external configuration storage pattern

4. Evaluation

Our evaluation strategy is to use the case study described in the previous section to compare our DSML against Elastic Tools for Azure SQL databases [19] and against manual code refactoring as a baseline. Elastic Tools provides libraries to simplify and manage large numbers of databases that support a SaaS. It also proposes Row-Level Security [20] that enables the implementation of restrictions on data row access based on user query characteristics. During manual code refactoring, we followed the guidance to develop a multi-tenant SaaS [21] and cloud patterns [10].

In particular, we inspected the time required for evolution and measured code reliability through counting error frequencies for the following two evolution scenarios:

- *Scenario 1*: evolution from single- to multi-tenancy;
- *Scenario 2*: evolution from a single shared database model to a separate database per tenant model.

To implement *Scenario 1*, an application needs to introduce tenant entities (for tenant users and administrators), and implement authentication logic to handle tenant-specific data entities. This entails not only modifications to the database schema, but also to queries through presentation and business logic layers. *Scenario 2* requires the application to map each tenant to its own database instance and to remove row-level security as each tenant connects to a physically isolated database.

4.1. Developer Time

Figure 4 shows the developer time spent to support each evolution scenario. To ensure fair comparison, we excluded the time needed to learn the appropriate APIs for Elastic Tools and tool support for the DSML.

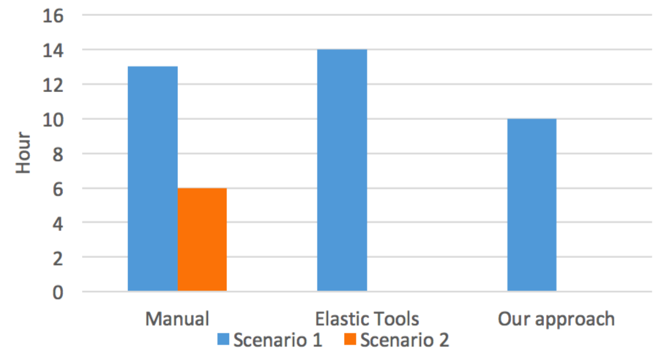

Figure 4: The time spent for the evolution scenarios in hours

In order to adopt multi-tenancy (*Scenario 1*), the following implementation modifications were required: (i) Enhance the authentication mechanism to retrieve tenant configuration information when an end user authenticates; (ii) Modify the data layer by introducing Tenant and TenantAdministrator entities; (iii) Alter the existing database schema by adding TenantID to tenant-specific tables in order to provide row-level access; and (iv) Modify CRUD queries to ensure tenant isolation at the presentation and business logic layers.

Using manual evolution, the above implementation steps required a total of 13 hours. Using Elastic Tools required 14 hours. In contrast, only 10 hours were needed to support *Scenario 1* using our DSML as the required modifications were easier to introduce through the high level database architecture it offers. The graphical representation of the data architecture improves understandability, and the code generator directly transforms changes in the model to the data layer. However, manual changes were required in the business logic and presentation layers.

To support *Scenario 2*, both our DSML and Elastic Tools required no additional time on top of that needed to move to a multi-tenant model. Manual refactoring, on the other hand, required additional code changes that took about 6 hours in order to dynamically connect to a tenant-specific database and to remove row-level access.

4.2. Reliability

We also evaluate the reliability of our DSML through the frequency of making errors in application code during the evolution processes. For this, we used the Microsoft .NET unit testing framework for managed code to identify errors on a module level. We categorized errors according to their appearance at each layer of the application. The total number of errors for both evolution scenarios by each approach is illustrated in Figure 5.

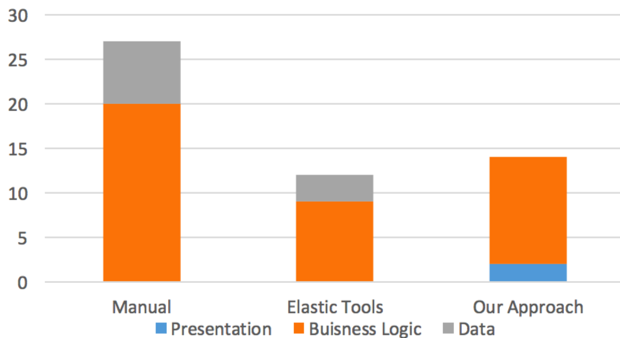

Figure 5: Error counts at each application layer for evolution through *Scenario 1* followed by *Scenario 2*

The manual approach is noticeably the most error-prone one. Specifically, 20 and 7 errors were encountered at the business logic layer and data layer, respectively. This is not particularly surprising given the additional amount of logic that has to be manually implemented. Evolution of the presentation layer to multi-tenancy went smoothly as this layer only required minor changes.

The automated approaches fared better in terms of error count. Where these errors occurred, however, differ slightly between them. At the data layer, there were no errors when using our DSML due to the automation. The DSML is specifically designed for this purpose so it makes it easy to model a multi-tenant data structure, model a database schema for each tenant type or even per tenant, manage tenant-specific and shared tables, and generate source code from the model. In contrast, errors at this level were found for the remaining approaches: 7 for manual refactoring (as already described) and 3 for Elastic Tools. The errors were caused by missed TenantID in the tenant-specific entities, failures to connect to a database, incorrect routing of a tenant to its database instance, and faulty implementation of sharding management.

Errors at the business logic layer include incorrect implementation of tenant-isolation, invalid queries to interact with the data layer and issues with mapping an end user to its tenant. The number of such errors is high (20) with the manual approach but relatively low with both our DSML (12) and Elastic Tools (9). Adjusting queries to ensure tenant isolation using the manual approach and Elastic Tools introduced many problems, especially with complex queries. In case of the DSML, errors were found during the generation of a primary key for tenant-specific entities.

Interestingly, presentation layer errors were only found with our DSML (2). In single-tenancy, entities are retrieved by their primary keys using HTTP GET requests. In multi-tenancy, we include TenantID as a prefix of tenant-specific entities' primary keys. This revealed the TenantID in the query string in the web browser address bar when an entity was requested for editing, and the errors occurred when we tried to conceal it from end-users.

As a tool prescribed for highly evolving data applications, Elastic Tools evidently still requires substantial effort and cost for the fairly steady workload presented by our basic use case. Furthermore, Elastic Tools has been designed for Azure SQL databases only, which limits its application for other cloud platforms. Nonetheless, it is representative for our comparative purposes here as it is available for public, is suitable for the application structure, and it supports all multi-tenancy patterns for the data layer.

5. Discussion

Overall, our results indicate that using the DSML to handle evolution in multi-tenant SaaS applications saves development time and greatly minimizes the number of errors at the data layer. However, it also comes with minor errors in other layers of the application, namely the presentation layer. This could be minimized through defining dependencies between the application layers and transforming changes from one layer to subsequent layers.

The DSML allows a developer to model a database structure and generate a source code from the model. During further evolution of the application, sometimes it is easier to make minor changes in the generated code rather than modifying the model and re-generating the application code. Currently, changes in the code are not transformed back into the model. As a result, synchronization issues emerge between the model and the code. We attempted to address this issue, however, the VMSDK tool that we used to implement the language lacks the capability to handle such two-way synchronization. This might be an area of future development, although making reverse synchronization work properly will lead to a higher implementation cost of the DSML.

Moreover, to evaluate the generalizability of the DSML, we need to extend a code generator so that it is compatible with the programming languages and frameworks for other cloud providers. Finally, when the data layer is evolved the relevant layers of the application must be traversed as modification in this layer entails corresponding changes in related layers.

6. Related Work

In this section, we discuss existent literature on handling multi-tenancy and evolution challenges in the context of SaaS cloud applications. We categorized the approaches as DSLs for cloud applications and industrial work that manages multi-tenancy at the data layer.

6.1. DSLs

CloudDSL supports application migration across cloud platforms [14]. It is a graphical DSL for describing different cloud IaaS services. The DSL itself was implemented by exploiting the Epsilon tool and it uses a cloud meta-model that covers a wide range of different cloud platform entities. The meta-model was implemented by extensive analysis of cloud platforms, identification of similarities of entities in cloud platforms, and combining entities and their relationships into a new meta-model.

Another DSL-based approach for cloud application development and deployment has been presented in [22] where a DSL designer implements a graphical DSML using Visual Studio DSL Tools and hosts it as a SaaS. This eliminates the need for installation and configuration of a modeling environment, thus an application designer models an application using a web browser and automatically deploys it to the Google App Engine cloud platform. A further DSL for creating automated functional tests for SaaS applications has been described in [23]. The approach addresses functional user interface and platform compatibility testing problems. The DSL is supported by a Visual Studio extension tool which allows to create, execute and debug tests.

6.2. Multi-tenant Databases

Multi-tenancy at the data layer has been addressed using DB2 software for a SQL database by IBM [24]. It supports all data models that were described in Section 1, and the author highlights the benefits and challenges of each data model. In addition, the following approaches to handle configurability have been described: (i) a tenant identifier for each row in a shared schema, and (ii) pureXML to store data. This approach limits flexibility as extensions require to alter the table, and extension columns will be included for all tenants. In pureXML, a table consists of two columns: a tenant identifier and XML data types. This allows tenants to have separate schemas per column. A similar approach has been introduced by [25] where PostgreSQL uses semi-structured data types, such as json, jsonb and hstore, to support multi-tenancy and configurability. The main difference from pureXML is that only variable fields are stored in a semi-structured data type.

As sharding SQL databases was not enough to support scalability and reliability of the AdWords system, Google has designed a distributed relational database system F1 [26]. F1 combines high availability and scalability of NoSQL systems, and consistency and usability of SQL databases. It proposes a hierarchical database model where the child table contains a foreign key to its parent table as a prefix of its primary key. For example, the Education Center schema may contain a table Tenant with primary key (TenantID) that has a child table Department with primary key (TenantID, DepartmentID), which in turn has a table Instructor with primary key (TenantID, DepartmentID, InstructorID). Including a tenant identifier in primary keys ensures that tenant related data are in the same partition. Likewise, Twitter introduced Manhattan, a multi-tenant distributed database, that ensures scalability and low latency in a real-time environment [27]. Customers interact with the storage system through a dedicated interface layer. As in Google's F1, a hierarchical key structure is used to design a data architecture. However, these databases are not publicly available, and they depend on other proprietary technologies.

7. Conclusion and Future Work

Multi-tenant cloud applications have a potential of saving different operational costs through consolidation. However, evolving a single-tenant SaaS application into a multi-tenant one is a relatively costly and error-prone process. In this paper, we presented the architectural concerns during the evolution process of a single-tenant web application to a multi-tenant SaaS. We also proposed a DSML to address multi-tenancy challenges at the data layer. The DSML provides automated support to design a data architecture, generate source code and evolve an application's data layer.

We conducted a case study by applying our DSML on a typical single-tenant three-tier web application, and used that to compare our DSML to a manual code refactoring technique and to Microsoft Elastic Tools for Azure SQL. The outcomes show that using our DSML can decrease the number of errors in application code, and reduce the time and effort during the evolution from single- to multi-tenancy.

For future work, we intend to expand the application of our DSML to include further case studies on more cloud platforms (*e.g.* Amazon Web Services, Google App Engine) to further evaluate the generalizability of the DSML. We plan to conduct another case study with a more complex data model to validate the results. We also plan to improve the DSML to traverse all the relevant layers during evolution.

References

[1] P. M. Mell and T. Grance, "SP 800-145. The NIST Definition of Cloud Computing," National Institute of Standards & Technology, Gaithersburg, MD, United States, Tech. Rep., 2011.

[2] R. Krebs, C. Momm, and S. Kounev, "Architectural concerns in multi-tenant SaaS applications," in *Proceedings of the 2nd International Conference on Cloud Computing and Services Science (CLOSER)*, vol. 12, 2012, pp. 426–431.

[3] C.-P. Bezemer and A. Zaidman, *Challenges of reengineering into multi-tenant SaaS applications*. Delft University of Technology, Software Engineering Research Group, 2012.

[4] J. R. Hamilton *et al.*, "On designing and deploying internet-scale services," in *Large Installation System Administration Conference*, vol. 18. USENIX, 2007, pp. 1–18.

[5] S. Srikantaiah, A. Kansal, and F. Zhao, “Energy aware consolidation for cloud computing,” in *Proceedings of the Conference on Power aware computing and systems*, vol. 10, 2008, pp. 1–5.

[6] R. Boutaba, Q. Zhang, and M. F. Zhani, “Virtual machine migration in cloud computing environments: Benefits, challenges, and approaches,” *Communication Infrastructures for Cloud Computing*, pp. 383–408, 2013.

[7] M. Abu-Matar and J. 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, pp. 1–5.

[8] P. Bezemer and A. Zaidman, “Multi-tenant SaaS applications: Maintenance dream or nightmare?” in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, ser. IWPSE-EVOL ’10. ACM, 2010, pp. 88–92.

[9] S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, and W. Joosen, “Efficient customization of multi-tenant Software-as-a-Service applications with service lines,” *Journal of Systems and Software*, vol. 91, pp. 48–62, May 2014.

[10] A. Homer, J. Sharp, L. Brader, M. Narumoto, and T. Swanson, *Cloud Design Patterns prescriptive architecture guidance for cloud applications*. Microsoft Corporation, 2014.

[11] F. Chong and G. Carraro, *Architecture Strategies for Catching the Long Tail*. Microsoft Corporation, 2006.

[12] M. Fowler, *Domain Specific Languages*, 1st ed. Addison-Wesley Professional, 2010.

[13] A. van Deursen, P. Klint, and J. Visser, “Domain-specific languages: An annotated bibliography,” *SIGPLAN Not.*, vol. 35, no. 6, pp. 26–36, Jun. 2000.

[14] G. C. Silva, L. M. Rose, and R. 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*, 2014, pp. 36–45.

[15] N. Ferry, G. Brataas, A. Rossini, F. Chauvel, and A. Solberg, “Towards Bridging the Gap Between Scalability and Elasticity,” in *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER)—Special Session on Multi-Clouds*. SCITEPRESS, 2014, pp. 746–751.

[16] L. A. Barroso, J. Clidaras, and U. Hölzle, “The datacenter as a computer: An introduction to the design of warehouse-scale machines,” *Synthesis Lectures on Computer Architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[17] Y. Elkhatib, G. S. Blair, and B. Surajbali, “Experiences of using a hybrid cloud to construct an environmental virtual observatory,” in *Proceedings of the 3rd Workshop on Cloud Data and Platforms, co-located with the 8th European Conference on Computer Systems*, Apr 2013, pp. 13–18.

[18] A. Cameron Wills. (2011) Visualization and modeling sdk,” microsoft. [Online]. Available: https://code.msdn.microsoft.com/Visualization-and-Modeling-313535db

[19] S. Acharya, C. Rabeler, and K. Toliver, “Design patterns for multi-tenant SaaS applications and Azure SQL database,” 2017. [Online]. Available: https://docs.microsoft.com/en-us/azure/sql-database/sql-database-design-patterns-multi-tenancy-saas-applications

[20] R. Byham and C. Guyer, “Row-level security,” May 2016. [Online]. Available: https://msdn.microsoft.com/en-gb/library/dn765131

[21] D. Betts, A. Homer, A. Jezierski, M. Narumoto, and H. Zhang, *Developing Multi-tenant Applications for the Cloud on Windows Azure*. Microsoft patterns and practices, 2013.

[22] K. Sledziewski, B. Behzad, and A. Rachid, “A DSL-based approach to software development and deployment on cloud,” in *24th IEEE International Conference on Advanced Information Networking and Applications*, April 2010, pp. 414–421.

[23] D. Santiago, A. Cando, C. Mack, G. Nunez, T. Thomas, and T. M. King, “Towards domain-specific testing languages for Software-as-a-Service.” in *MDHPCL@MoDELS*, 2013, pp. 43–52.

[24] R. Chong, “Designing a database for multi-tenancy on the cloud considerations for SaaS vendors,” IBM Corporation, Tech. Rep., 2012.

[25] E. Ozgun. (2016) Designing your SaaS database for scale with Postgres. [Online]. Available: https://www.citusdata.com/blog/2016/10/03/designing-your-saas-database-for-high-scalability/

[26] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, J. Cieslewicz, I. Rae, T. Stancescu, and H. Apte, “F1: A distributed SQL database that scales,” *Proc. VLDB Endow.*, vol. 6, no. 11, pp. 1068–1079, Aug. 2013.

[27] S. Peter, “Manhattan, our real-time, multi-tenant distributed database for Twitter scale,” April 2014. [Online]. Available: https://blog.twitter.com/2014/manhattan-our-real-time-multi-tenant-distributed-database-for-twitter-scale