Institute of Architecture of Application Systems

Using Patterns to Move the Application Data Layer to the Cloud

Steve Strauch, Vasilios Andrikopoulos, Uwe Breitenbücher, Santiago Gómez Sáez, Oliver Kopp, Frank Leymann

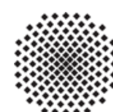
Institute of Architecture of Application Systems,
University of Stuttgart, Germany,
lastname@iaas.uni-stuttgart.de

BIBTeX:

```
@inproceedings{Strauch2013,
  author    = {Steve Strauch and Vasilios Andrikopoulos and Uwe Breitenbücher
               and Santiago Gómez Sáez and Oliver Kopp and Frank Leymann},
  title     = {Using Patterns to Move the Application Data Layer to the Cloud},
  booktitle = {Proceedings of the 5th International Conference on Pervasive
               Patterns and Applications, PATTERNS 2013, 27 May – June 1 2013,
               Valencia, Spain},
  year      = {2013},
  pages     = {26-33},
  publisher = {Xpert Publishing Services (XPS)}
}
```

The full version of this publication has been presented at
PATTERNS 2013.
http://www.iaria.org/conferences2013/PATTERNS13.html

© 2013 Xpert Publishing Services



Universität Stuttgart
Germany

Using Patterns to Move the Application Data Layer to the Cloud

Steve Strauch, Vasilios Andrikopoulos, Uwe Breitenbücher, Santiago Gómez Sáez, Oliver Kopp, Frank Leymann
Institute of Architecture of Application Systems (IAAS),
University of Stuttgart, Stuttgart, Germany
{firstname.lastname}@iaas.uni-stuttgart.de

***Abstract*—Cloud services allow for hosting applications partially or completely in the Cloud by migrating their components and data. Especially with respect to data migration, a series of functional and non-functional challenges like data confidentiality arise when considering private and public Cloud data stores. In this paper we identify some of these challenges and propose a set of reusable solutions for them, organized together as a set of Cloud Data Patterns. Furthermore, we show how these patterns may impact the application architecture and demonstrate how they can be used in practice by means of a use case.**

***Keywords*—Data layer; Cloud applications; Data migration; Cloud Data Patterns; Cloud data stores.**

I. INTRODUCTION

Cloud computing has become increasingly popular with the industry due to the clear advantage of reducing capital expenditure and transforming it into operational costs [1]. To take advantage of Cloud computing, an existing application may be moved to the Cloud or designed from the beginning to use Cloud technologies. Applications are typically built using a three layer architecture model consisting of a presentation layer, a business logic layer, and a data layer [2]. The presentation layer describes the application-users interactions, the business layer realizes the business logic and the data layer is responsible for application data storage. The data layer is in turn subdivided into the Data Access Layer (DAL) and the Database Layer (DBL). The DAL encapsulates the data access functionality, while the DBL is responsible for data persistence and data manipulation. Figure 1 visualizes the positioning of the various layers.

Each application layer can be hosted using different Cloud deployment models. Possible Cloud deployment models, also shown in Figure 1, are: Private, Public, Community, and Hybrid Cloud [3]. Figure 1 shows the various possibilities for distributing an application using the different Cloud types. The "traditional" application not using any Cloud technology is shown on the left of the figure. In this context, "on-premise" denotes that the Cloud infrastructure is hosted inside the company and "off-premise" denotes that it is hosted outside the company.

In this work, we focus on the lower two layers of Figure 1, the DAL and DBL layers of the application. Application data is typically moved to the Cloud because of, e. g., Cloud bursting, data analysis or backup and archiving. Using Cloud technology leads to challenges such as incompatibilities with the database layer previously used or the accidental disclosing of critical data by, e. g., moving them to a Public Cloud. Incompatibilities in the database layer may refer to inconsistencies between the functionality of an existing traditional database layer, and the functionality and characteristics of an equivalent Cloud Data Hosting Solution [4]. For instance, the Google App Engine Datastore [5] is incompatible with Oracle Corporation MySQL, version 5.1 [6], because the Google Query Language [7] supports only a subset of the functionality provided by SQL, e. g., joins are not supported. An application relying on such functionalities cannot therefore have its data store moved to the Cloud without deep changes to its implementation. It has to be noted here that, for the purposes of this work, we assume that the decision to migrate the data layer to the Cloud has already been made based on criteria such as cost, effort etc. [8], [9].

The contribution of this paper is the identification of such challenges and the description of a set of *Cloud Data Patterns* as the best practices to deal with them. As defined in [10], a Cloud Data Pattern describes a *reusable and implementation technology-independent solution for a challenge related to the data layer of an application in the Cloud for a specific context*. For this purpose, in the following we present an initial catalog of Cloud Data Patterns dealing with functional, non-functional and privacy-related aspects of having the application data layer realized in the Cloud. The Cloud Data Patterns are geared towards the Platform as a Service (PaaS) delivery model [3]. The presented list of the patterns is a result of our collaboration with industry partners and research projects. We do not claim that the list of patterns is complete and we plan to expand it in the future.

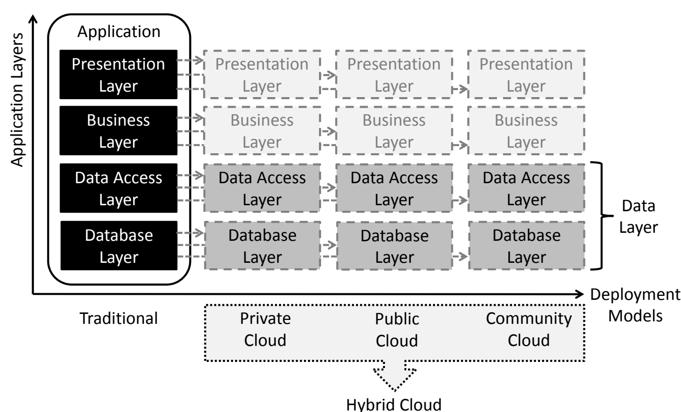


Figure 1: Overview of Cloud Deployment Models and Application Layers

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9

The presentation of the patterns uses the format defined by Hohpe and Woolf [11], consisting of the description of a *context* where the pattern is applicable, the *challenge* posed, external or internal *forces* that impose constraints that make the problem difficult to solve, a proposed *solution* for the challenge, detailed technical issues (as *sidebars*), the *results* of applying the proposed solution in the defined context, an *example* of use, and other patterns to be considered (*next*). A representative *icon* and a graphical *sketch* of the pattern are also provided. In addition, we show how these patterns can be used in practice by means of a use case.

The remainder of this paper starts with providing a motivating scenario (Section II) highlighting some of the challenges that need to be addressed in the following sections. A set of functional Cloud Data Patterns are presented in Section III as best practices for addressing these challenges. Sections IV and V summarize and update some of the patterns we defined for the same purposes in [4] and [10], respectively, but with respect to Quality of Service and data confidentiality. Section VI discusses the impact of applying these patterns to the application layers and evaluates them in practice using the motivating scenario of Section II. A presentation of related work is contained in Section VII; conclusions and future work in Section VIII.

II. Motivating Scenario

For purposes of an illustrative example let us consider the case of a health insurance company in Germany. As a result of the increase of the numbers of its clients, the company stores their data in two data centers in different parts of Germany. The data centers, covering geographical regions A and B, respectively, form a private Cloud data hosting solution that offers a uniform access point and view of the data to the various applications used by the employees of the company. The company is required to provide access to an external auditor to the financial transactions processed by the company. The auditor essentially executes a series of predefined complex queries on the financial transactions data at irregular intervals and reports back to the company and the responsible authorities their findings. The health insurance company however is also obliged by law to protect the personal data privacy and the confidentiality of the medical record of its clients. For this purpose, the company takes special care to anonymize the results of the queries executed by the auditor in order to ensure that no client information is accidentally exposed.

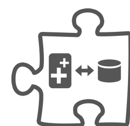
Providing the external auditor with direct access to the database of the company raises a series of concerns about a) ensuring the security of the company-internal data, and b) the performance of the company systems, as an indirect result of the unpredictable additional load imposed by the complex queries executed by the auditor. As a solution to these issues, it is proposed to use a public Cloud data hosting solution provider and migrate a consistent replica of the financial transaction records to the public Cloud, stripped of any personal data. The auditing company would then be able to retrieve the necessary information without burdening the company systems. Such a migration to the Cloud however, even if only partial, requires addressing different kinds of challenges: confidentiality-related (ensuring that it is impossible to recreate the medical records and other personal information of the company clients using the data in the public Cloud), functionality-related (providing both all the necessary data and the querying mechanisms for the auditor to operate as required), and non-functional (ensuring that the partial migration does not encumber in any way the performance of the company systems). The following sections discuss a series of Cloud Data Patterns that address these issues.

III. Functional Patterns

Cloud data stores can be considered as appliances where a fixed set of functionality is provided [12]. Cloud data stores include SQL and NoSQL solutions [13], [14]. Each solution is geared towards a specific application domain and therefore does not come with all possible features. Furthermore, the offered functionalities may be configurable but not extensible. Functional Cloud Data Patterns provide solutions for these challenges. More specifically:

A. Data Store Functionality Extension

The *Data Store Functionality Extension* pattern adds a missing functionality to a Cloud data store.



Context: A Cloud data store does not inherently support all functionalities usually offered by a traditional data store. For instance, the Cloud data store might not support data joins. The choice of which data store to use is fixed by the application requirements or contractual obligations and therefore it is not possible to replace the data store with an equivalent one offering the missing functionality.

Challenge: How can a Cloud data store provide a missing functionality?

Forces: The missing (but required) functionality might be implemented in the business layer. An example of missing functionality are joins. Implementation of the missing functionality on a higher application layer requires all data to be retrieved from the database layer and leads to increased network load.

Solution: A component implements the required functionality as an extension of the data store, either by offering an additional functionality, or by adapting one or more of the existing functionalities offered by the data store. The extension component is placed within the Cloud infrastructure of the Cloud data storage. A low distance (in terms of network performance) ensures low latency between the extension and the data store.

Sidebars: The additional or extended functionality code has to be wrapped into an application, which can be hosted in the Cloud. The access to the data store of the Cloud provider from this application is done via the API supplied by the provider. The code in the data access layer has to be adjusted accordingly, denoted by “Data Access Layer*” in Figure 2 case (a). This means that each data access call using the required functionality has to be replaced by a call to the component implementing the corresponding data store functionality extension.

Results: The Cloud data store functionality is extended. Assuming all additional functionality required (and not provided by the Cloud data store) can be implemented within the component implementing the functionality extension, there is no adjustment or modification of the business layer required.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9

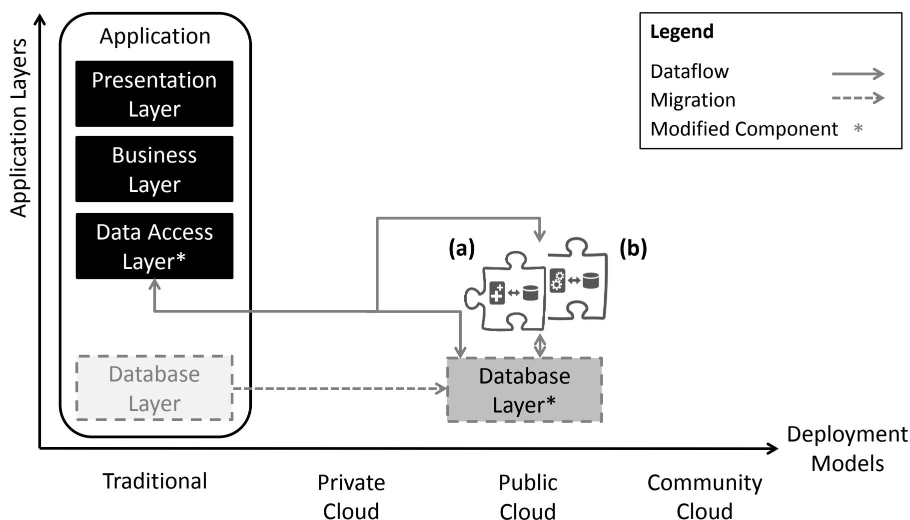


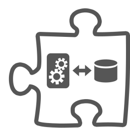
Figure 2: Sketch of (a) Data Store Functionality Extension and (b) Emulator of Stored Procedures

Example: The database layer of an application built on Oracle Corporation MySQL version 5.1 [6], is moved to Google App Engine Datastore [5] by Google Inc. As a result, the database layer functionality is incompatible, because the Google Query Language [7] supports only a subset of the functionality provided by SQL, e. g., join functionality is not supported. As the application requires join functionality, this additional functionality has to be provided by the component implementing the join functionality.

Next: In case functionality for stored procedures has to be added, the “Emulator of Stored Procedures” pattern has to be considered.

B. Emulator of Stored Procedures

The *Emulator of Stored Procedures* pattern is a special case of the Data Store Functionality Extension pattern see case (b) in Figure 2, where an extension component is built outside the data store, containing a set of predefined groups of commands to be executed by the data store. While this is a very common mechanism in traditional data stores, many Cloud data stores do not support it natively. Due to its ubiquity and usefulness we therefore define a separate pattern for it.



Context: Stored procedures “are application programs that execute within the database server process” [15]. A Cloud data store does not inherently support stored procedures as most traditional data stores do. Changing the provider of the Cloud data store might not be an option, because of other advanced features provided or due to specific customer requirements.

Challenge: How can a Cloud data store not supporting stored procedures provide such functionality?

Forces: To keep network traffic low, the number of requests to the database layer should be minimized. Thus, the stored procedure code should not run on-premise, but within the Cloud infrastructure of the Cloud data store.

Solution: An emulator of stored procedures is placed within the Cloud infrastructure of the Cloud data storage. A low distance (measured in terms of network performance) ensures low latency between the emulator and the data store and reduces communication overhead.

Sidebars: The stored procedure code has to be wrapped into an application, which can be hosted in the Cloud. The access to the data store of the Cloud provider is done via the API supplied by the provider. The code of the data access layer has to be adjusted accordingly, denoted by “Data Access Layer*” in Figure 2 case (b). This means that each call to the stored procedure has to be replaced by a call to the emulator.

Results: Instead of emulating the stored procedure functionality in the business layer, the functionality is provided as an application in the Cloud. The number of requests from the data access layer to the database layer is reduced as the work is done by the stored procedure emulator. Assuming the data transfer between nodes in the provider’s Cloud infrastructure is free and the stored procedure emulator is hosted there, then costs are also reduced.

Example: The database layer of an application built on Microsoft SQL Server with stored procedures is moved to Microsoft SQL Azure Database. As Microsoft SQL Azure does not support full text search stored procedures [16], this functionality has to be emulated.

Next: In case more functionality has to be added, the “Data Store Functionality Extension” pattern has to be considered.

IV. Non-Functional Patterns

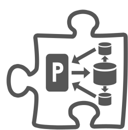
We previously investigated non-functional patterns with focus on providing solutions for ensuring an acceptable Quality of Service (QoS) level by means of scalability in case of increasing data read or data write load [17]. In the following, we provide an overview on these non-functional patterns focusing on their *context*, *challenge*, and *solution*. When considering the data rather than the database system, there are two scaling options available: vertical and horizontal data

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9

scaling. Vertical data scaling can be achieved by moving the data to a more powerful database system, which offers better performance, advanced functionalities, or both. Horizontal data scaling is based on partitioning the data according to functional groups [18]. Examples for functional groups are European customers and American customers. Each functional group may be itself distributed among different database systems to increase search speed. This method is also called *sharding* [19].

A. Local Database Proxy

The *Local Database Proxy* enables read scalability by requiring a master/multiple slave model and forwarding read requests to any read replica.



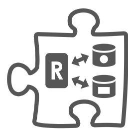
Context: A Cloud data store does not inherently support horizontal scalability for data reads. When the data read load of the application permanently increases, e. g., due to increased user acceptance and usage, a mechanism for horizontally scaling read requests is required. Additionally, the business logic of processing user requests is also moved to the Cloud.

Challenge: How can a Cloud data store not supporting horizontal data read scalability provide that functionality?

Solution: The Cloud data store is configured using a single master/multiple slave model. The master handles data writes and the slaves are used as replicas serving read requests only. In case the application has to deal with stale data, the replication of data may be lazy. A proxy component is locally added below each data access layer. All requests from each data layer are routed through the respective proxy. The proxy routes data read requests to any slave and write requests to the master.

B. Local Sharding-Based Router

The *Local Sharding-Based Router* enables read and write scalability by requiring the independent splitting and distributing of data into functional groups and forwarding read and write requests to the corresponding shard.



Context: A Cloud data store does not inherently support horizontal scalability for data reads and writes. When the data read load of the application permanently increases, e. g., due to increased user utilization, a mechanism for horizontally scaling read requests to the data store is required. Furthermore, a permanent high data update rate of the application requires also horizontally scaling for data writes. The business logic of processing user requests is moved to the Cloud.

Challenge: How can a Cloud data store not supporting horizontal data read and write scalability provide that functionality?

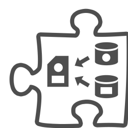
Solution: The data to be stored in the Cloud are split horizontally. This means that tables with many rows are split into several data stores. Each data store is assigned a distinct number of rows of the original table. This technique is called “sharding” [19]. A dedicated sharding-based router is added locally below each data access layer. All requests from each data layer are routed through the respective sharding-based router. The local sharding-based router forwards data read and write requests to the appropriate Cloud data store.

V. Confidentiality Patterns

In our previous work [10], we presented Cloud Data Patterns for confidentiality. They deal with data that have to be kept private and secure, commonly referred to as “critical data”. In the following, we present and update these patterns focusing on their *context*, *challenge*, and *solution*.

A. Confidentiality Level Data Aggregator

Critical data can be categorized into different confidentiality levels. As data are not always categorized by confidentiality, or categorized using different confidentiality categorizations, the confidentiality level has to be harmonized. The *Confidentiality Level Data Aggregator* provides one confidentiality level for data from different sources with potentially different confidential categorizations on different scales.



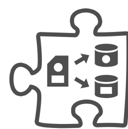
Context: The data formerly stored in one traditionally hosted data store is separated according to the different confidentiality levels and stored in different locations. The business layer is separated into the traditionally hosted part processing the critical data, and the part hosted in the public Cloud processing the non-critical data. As the application accesses data from several data sources, the different confidentiality levels of the data items have to be aggregated to one common confidentiality level. This builds the basis for avoiding disclosure of critical data by passing it to the public Cloud.

Challenge: How can data of different confidentiality levels from different data sources be aggregated to one common confidentiality level?

Solution: An aggregator retrieves data from all Cloud data stores. The aggregator is placed within the Cloud infrastructure of the Cloud data storage with the highest confidentiality level. Since it must be able to process data with the highest confidentiality level, it may not be placed where data with a lower level of confidentiality reside. As a consequence, the aggregator has to be placed in a location where the demands of the highest confidentiality level are fulfilled.

B. Confidentiality Level Data Splitter

The *Confidentiality Level Data Splitter* splits data according to pre-configured privacy levels. This is required when an application writes data to multiple data stores with different confidentiality levels.



Context: The data formerly stored in one traditionally hosted data store is separated between data stores with different confidentiality levels. As the application writes data to several data stores, the data have to be categorized and split according to their confidentiality level. This builds the basis for avoiding disclosure of critical data when storing them in the public Cloud.

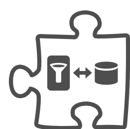
Challenge: How can data of one common confidentiality level be categorized and split into separate data parts belonging to different confidentiality levels?

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9

Solution: A splitter is placed within the infrastructure of the data access layer of the application. Thus, additional data movement, network traffic, and load can be minimized. The splitter writes data to all Cloud data stores. As the splitter processes data with the highest confidentiality level, it has to be placed in a location where the demands of the highest confidentiality level are fulfilled.

C. Filter of Critical Data

The *Filter of Critical Data* ensures that no confidential data are disclosed to the public. The filter enforces that no data leaves the private Cloud by filtering out critical data.



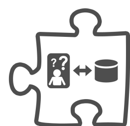
Context: The private Cloud data store contains both critical and non-critical data. To prevent disclosure of the critical data, it has to be enforced that the critical data do not leave the private Cloud. The logic implemented in the business layer is split into one part processing critical and one part processing non-critical data. The party implementing and/or hosting the business logic for processing the non-critical data cannot be trusted.

Challenge: How can data-access rights be ensured when moving the database layer into the private Cloud together with a part of the business layer, as well as a part of the data access layer to the public Cloud?

Solution: A filter for the critical data is placed within the infrastructure of the private Cloud data store. All requests to the private Cloud data store have to be directed to the filter. The private Cloud data store is only reachable through the filter. Requests for critical data originating off-premises are denied by the filter.

D. Pseudonymizer of Critical Data

The *Pseudonymizer of Critical Data* implements pseudonymization. Pseudonymization is a technique to provide a masked version of the data to the public while keeping the relation to the non-masked data in private [20]. This enables processing of non-masked data in the private environment when required.



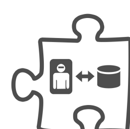
Context: The private Cloud data store contains critical and non-critical data. The business layer is partially moved to the public Cloud and needs access to data. The logic implemented in the business layer is split into one part requiring critical data, and one where critical data in pseudonymized form is sufficient for processing. The party implementing and/or hosting the business logic for processing pseudonymized data may not be trusted. Furthermore, passing critical data may be restricted by compliance regulations. It also is required to be able to relate the pseudonymized data processing results from the public business layer back to the critical data.

Challenge: How can a private Cloud data store ensure passing critical data in pseudonymized form to the public Cloud?

Solution: A pseudonymizer of data is placed within the infrastructure of the private Cloud data storage. All requests to the private Cloud data storage have to be directed to the pseudonymizer. The private Cloud data storage is only reachable by the pseudonymizer. Results of requests for critical data originating off-premises are pseudonymized.

E. Anonymizer of Critical Data

The *Anonymizer of Critical Data* implements anonymization [20]. Anonymization is a technique to provide a reduced version of the critical data to the public while ensuring that it is impossible to relate the reduced version to the critical data.



Context: The private Cloud data store contains both critical and non-critical data. The business layer is partially moved to the public Cloud and needs access to data. To prevent disclosure and misuse, the critical data are anonymized before being passed to the public Cloud. The logic implemented in the business layer is split into one part requiring critical data, and one where critical data in anonymized form are sufficient for processing. The party implementing and/or hosting the business logic for processing the anonymized data cannot be trusted. It is not required to be able to relate the anonymized data processing results from the public business layer back to the critical data.

Challenge: How can a private Cloud data store ensure passing critical data in anonymized form to the public Cloud?

Solution: An anonymizer is placed within the infrastructure of the private Cloud data store. All requests to the private Cloud data store have to be directed to the anonymizer. The private Cloud data store is only reachable through the anonymizer. Results of requests for critical data originating off-premises are anonymized.

For more details on these patterns, the interested reader is referred to [10]. In the following sections we combine these patterns with the ones we defined in Section III and Section IV in order to demonstrate how they can be used in practice.

VI. Cloud Data Patterns in Practice

Table I provides an overview of the impact created by the application of the patterns presented in the previous sections to the various application layers. Table I distinguishes between the layer the patterns are supposed to be realized in, and the ones that may require additional modifications as a result of applying them.

As the functional patterns are used in order to add additional functionality to a Cloud data store, they are realized in the database layer. In case the extended functionality should be used for a data request, the request has to go through the realization of the functional pattern. Thus, adaptations of the data access layer are also required.

Non-functional and confidentiality patterns are supposed to be realized in the data access layer. The confidentiality patterns Confidentiality Level Data Aggregator, Filter of Critical Data, Pseudonymizer of Critical Data, and Anonymizer of

TABLE I: Relation between Cloud Data Patterns and Application Architecture Layers

Cloud Data Patterns / Application Layers	Business Layer	Data Access Layer	Database Layer
Data Store Functionality Extension	⊘	△	◇
Emulator of Stored Procedures	⊘	△	◇
Local Database Proxy	⊘	△◇	△
Local Sharding-Based Router	⊘	△◇	△
Confidentiality Level Data Aggregator	△	△◇	⊘
Confidentiality Level Data Splitter	⊘	△◇	⊘
Filter of Critical Data	△	△◇	⊘
Pseudonymizer of Critical Data	△	△◇	⊘
Anonymizer of Critical Data	△	△◇	⊘

Legend: ⊘ has no impact on, ◇ is realized in, △ requires adaptations to

Critical Data require also adaptations of the business layer of the application. This is because, by realizing the corresponding patterns, the business logic might not have the same view on the data as before since the business layer has to deal with data in aggregated, filtered, pseudonymized or anonymized form.

As patterns are related to each other — to be considered as a whole and to be composable [11], we have chosen the form of a piece of a puzzle for the pattern icons. Whether two or more Cloud Data Patterns are composable depends on the semantics and functionality of each of the patterns. Moreover, the specific requirements and context of the needed solution effect whether a composition of patterns is required. Thus, we do not claim that all Cloud Data Patterns are composable with each other. A deeper investigation under which conditions a composition of Cloud Data Patterns is possible, and what are the resulting semantics, is required. The investigation and results leading to a Cloud Data Pattern language are part of our future work.

In the following, we discuss how the functional, non-functional, and confidentiality patterns can be used in practice based on the motivating scenario introduced in Section II. Figure 3 provides an overview of the realization of the scenario using Cloud Data Patterns. More specifically, in order to provide horizontal scalability for reads and writes to the data of the clients and their transactions, the Local Sharding-Based Router pattern is used. The data are separated between the two data centers according to geographical location.

The Google App Engine Datastore is chosen for outsourcing the storage of the data on financial transactions, configured accordingly. The client information and their corresponding medical records are critical data to be kept only in the company's private Cloud. Financial transactions information will appear both in the private and in the public Cloud. In order to keep both the data stored in the private data store and the one outsourced to the public Cloud consistent, data updates and inserts should be done in parallel to both the private and the public part of the database layer. However, only a part of the financial data is necessary for the auditing (e. g., client names can be removed) and the remaining can be pseudonymized before moving to the public Cloud (e. g., bank account numbers replaced by serial IDs). A composition of the Filter of Critical Data and the Pseudonymizer of Critical Data is used to fulfill both requirements. The filter is configured so that only data on financial transactions pass it. After passing the filter, the information on financial transactions is pseudonymized before it is stored in the public Cloud. Storage of data on the auditing company side can be done either in a private Cloud or in the traditional manner; this is out of the scope of our discussion.

Due to the challenges identified in the motivating example regarding the data access of the auditing company, the query results must not contain any relations concerning clients and their corresponding medical records or personal data. Therefore, this information has to be completely deleted before passing the query results to the auditing company (instead of simply obfuscating this relation by using pseudonymization). In addition, the queries to be executed by the auditor have to be agreed upon in advance. For these purposes, the realization uses a composition of the Anonymizer of Critical Data and the Emulator of Stored Procedures patterns. The emulator is used to predefine and restrict the data queries allowed to be executed by the auditing company. The Anonymizer of Critical Data additionally ensures the removal of any critical information from the results of the queries. A combination of functional, non-functional, and confidentiality patterns can therefore be used in tandem to address the requirements of the use case posed by the motivating scenario.

VII. RELATED WORK

Pattern languages defining reusable solutions for recurring challenges in architecture have been first proposed by Christopher Alexander [21]. A series of well-established patterns have been previously identified concerning, e. g., software engineering [22], enterprise integration [11] and application architecture [2]. Such general works do not consider building or migrating the database layer in the Cloud. Nevertheless, we reuse the pattern format defined by Hohpe and Woolf [11] for describing our Cloud Data Patterns.

Petcu [23] proposes Cloud usage patterns for Cloud-based applications based on existing use cases. Fehling et al. [24] and Pallmann [25] provide high-level architectural patterns to design, build, and manage applications using Cloud services. None of these works discusses patterns for building and/or moving the data layer to the Cloud. Adler [12] provides contributions regarding best practices for scalable applications in the Cloud. In this paper we reuse some of the results presented in [12] to form the non-functional patterns presented in Section IV.

ARISTA Networks, Inc. [26] provides seven patterns for Cloud computing of which only one (the Cloud Storage pattern)

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9

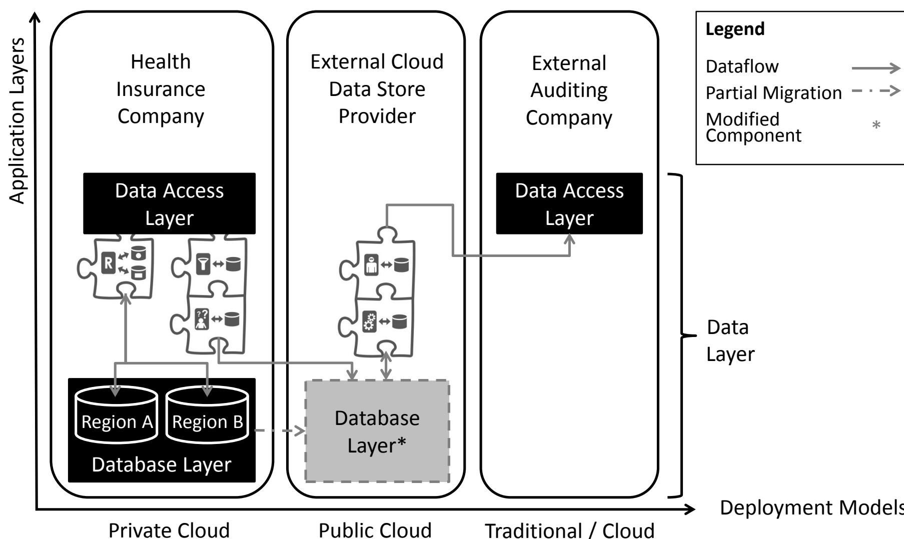


Figure 3: Realization of motivating scenario using Cloud Data Patterns (Data Layer)

deals with data in the Cloud. Nock [27] provides patterns for data access in enterprise applications, without however treating Cloud data stores in the same manner as we do.

Schumacher et al. [28] present reusable solutions for securing applications, but do not deal with data pseudonymization, data anonymization, and data filtering. Hafiz [29] presents a privacy design pattern catalog consisting of nine patterns achieving anonymity by mixing data with data from other sources instead of providing a general pseudonymization, anonymization, or filtering pattern. Creese et al. [30] consider design patterns for data protection of Cloud services. Romanosky et al. [31] describe privacy patterns applicable for online interactions. Schumacher [32] introduces an approach for mining security patterns from security standards and presents two patterns for anonymity and privacy. These works do not consider building a data layer in the Cloud or migrating an existing one there; some of the mechanisms identified however (e. g., pseudonymization) are reused in the Cloud Data Patterns we propose.

Finally, Schuemmer [33] presents patterns filtering personal information to establish boundaries for interactions between users utilizing collaborative systems. Our patterns are more general in the sense that they are not limited to filtering of personal data.

VIII. Conclusions and Future Work

This work presented a set of reusable solutions to face the challenges of moving the data layer to the Cloud or designing an application using a data store in the Cloud. The challenges and proposed solutions were organized as a non-exhaustive catalog of Cloud Data Patterns focusing on the PaaS delivery model. These patterns are the result of our collaboration with industry partners and research projects. Patterns for functional, non-functional, and confidentiality issues were discussed and shown how they can be combined in order to address a use case in practice.

The presentation of the patterns focused on the design issues, rather than the underlying technical challenges, in order to ensure their applicability across different technological platforms. This means that issues requiring a deeper technical insight, like for example scalability, are not covered sufficiently in the scope of this work. Nevertheless, these are issues we are currently looking into. For example, in the discussion in Section VI, implementing the Local Sharding-Based Router as a single component may result in a bottleneck for scalability, or even to a complete failure of the data access/database layer connection. A possible scalability enabling mechanism, and a counter-measure to single points of failure is to implement each pattern using a hot-pool of pattern realizations in the Cloud. A hot-pool consists of multiple instances of the realization component and a watchdog. Such issues and their possible solutions are investigated as part of a larger scale evaluation of our patterns using an industrial case study. Toward this direction, we also plan to formalize a general composition method of Cloud Data Patterns and expand our catalog with identified patterns presented here.

Acknowledgments

The research leading to these results has received funding from projects 4CaaSt (grant agreement no. 258862) and Allow Ensembles (grant agreement no. 600792) part of the European Union’s Seventh Framework Programme (FP7/2007-2013), and the BMWi-project Cloud-Cycle (01MD11023). We thank Tobias Unger for his valuable input.

References

[1] M. Armbrust *et al.*, “Above the Clouds: A Berkeley View of Cloud Computing,” EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, 2009.

[2] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, November 2002.

[3] P. Mell and T. Grance, “Cloud Computing Definition,” *National Institute of Standards and Technology*, July 2009.

[4] S. Strauch, O. Kopp, F. Leymann, and T. Unger, “A Taxonomy for Cloud Data Hosting Solutions,” in *Proceedings of the International Conference on Cloud and Green Computing (CGC ’11)*. IEEE Computer Society, Dezember 2011, pp. 577–584.

[5] Google, Inc., “Google App Engine Datastore,” 2011.

[6] Oracle Corporation, “MySQL,” 2011, http://www.mysql.com 31.03.2013.

[7] Google, Inc., “Google App Engine GQL Reference,” 2011, https://code.google.com/intl/en/appengine/docs/python/datastore/gqlreference.html 31.03.2013.

[8] B. C. Tak, B. Urgaonkar, and A. Sivasubramaniam, “To Move or Not to Move: the Economics of Cloud Computing,” in *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud’11. Berkeley, CA, USA: USENIX Association, 2011.

[9] M. Menzel and R. Ranjan, “CloudGenius: Decision Support for Web Server Cloud Migration,” in *Proceedings of WWW ’12*. New York, NY, USA: ACM, 2012, pp. 979–988.

[10] S. Strauch, U. Breitenbücher, O. Kopp, F. Leymann, and T. Unger, “Cloud Data Patterns for Confidentiality,” in *Proceedings of CLOSER’12*. SciTePress, April 2012, pp. 387–394.

[11] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2003.

[12] B. Adler, “Building Scalable Applications In the Cloud: Reference Architecture & Best Practices, RightScale Inc.” 2011, http://www.rightscale.com/info_center/white-papers/building-scalable-applications-in-the-cloud.php 31.03.2013.

[13] C. Strauch, “NoSQL Databases,” February 2011, http://www.christof-strauch.de/nosqldbs.pdf 31.03.2013.

[14] Stefan Edlich, “List of NoSQL Databases,” July 2011, http://nosql-database.org 31.03.2013.

[15] P. A. Bernstein, *Principles of Transaction Processing (Morgan Kaufmann Series in Data Management Systems)*, 2nd ed. Morgan Kaufmann, 2006.

[16] Microsoft, “System Stored Procedures (SQL Azure Database),” August 2011, http://msdn.microsoft.com/en-us/library/ee336237.aspx 31.03.2013.

[17] S. Strauch, V. Andrikopoulos, U. Breitenbücher, O. Kopp, and F. Leymann, “Non-Functional Data Layer Patterns for Cloud Applications,” in *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science (CloudCom’12)*. IEEE Computer Society Press, Dezember 2012, pp. 601–605.

[18] K. Küspert and J. Nowitzky, “Partitionierung von Datenbanktabellen,” *Informatik-Spektrum*, vol. 22, pp. 146–147, 1999.

[19] J. Zawodny and D. Balling, *High Performance MySQL: Optimization, Backups, Replication, Load-balancing, and More*. O’Reilly & Associates, Inc. Sebastopol, CA, USA, 2004.

[20] Federal Ministry of Justice, “German Federal Data Protection Law,” December 1990, http://www.gesetze-im-internet.de/bdsg_1990/ 31.03.2013.

[21] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language. Towns, Buildings, Construction*. Oxford University Press, 1977.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, October 1994.

[23] D. Petcu, “Identifying Cloud Computing Usage Patterns,” in *2010 IEEE International Conference on Cluster Computing Workshops and Posters (CLUSTER WORKSHOPS)*. IEEE, October 2010.

[24] C. Fehling, F. Leymann, R. Retter, D. Schumm, and W. Schupeck, “An Architectural Pattern Language of Cloud-based Applications,” in *Proceedings of PLoP’11*. ACM, October 2011.

[25] D. Pallmann, “Windows Azure Design Patterns,” 2011, http://www.windowsazure.com/en-us/develop/net/architecture/ 31.03.2013.

[26] ARISTA Networks, Inc., “Cloud Networking: Design Patterns for Cloud-Centric Application Environments,” January 2009.

[27] C. Nock, *Data Access Patterns: Database Interactions in Object Oriented Applications*. Prentice Hall Professional Technical Reference, February 2008.

[28] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating Security and Systems Engineering*. Wiley, 2006.

[29] M. Hafiz, “A Collection of Privacy Design Patterns,” in *Proceedings of PLoP’06*, New York, NY, USA, 2006.

[30] S. Creese, P. Hopkins, S. Pearson, and Y. Shen, “Data Protection-Aware Design for Cloud Services,” in *Proceedings of CloudCom’09*, 2009, pp. 119–130.

[31] S. Romanosky, A. Acquisti, J. Hong, L. F. Cranor, and B. Friedman, “Privacy Patterns for Online Interactions,” in *Proceedings of PLoP’06*. ACM, 2006.

[32] M. Schumacher, “Security Patterns and Security Standards,” in *Proceedings of the 7th European Conference on Pattern Languages of Programs (EuroPLoP)*, July 2002.

[33] T. Schuemmer, “The Public Privacy–Patterns for Filtering Personal Information in Collaborative Systems,” in *Proceedings of the Conference on Human Factors in Computing Systems (CHI’04)*, 2004.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-276-9