On the Principles of Microservice-NoSQL-Based Design for Very Large Scale Software: A Cassandra Case Study

Duc Minh Le[1](✉), Van Dai Pham[1], Cédrick Lunven[2], and Alan Ho[2]

[1] Department of Information Technology, Swinburne Vietnam, FPT University, Hanoi, Vietnam
{duclm20,daipv11}@fe.edu.vn
[2] DataStax, Inc., Santa Clara, USA
{cedrick.lunven,alan.ho}@datastax.com

Abstract. Developing very large scale distributed software systems is challenging from both functional and data management perspectives. Methods based on Microservices Architecture (MSA) have gained popularity for addressing the functional challenges. On the other hand, cloud-aware, very large scale NoSQL data management systems have also proved their effectiveness in tackling data management's scalability challenges. Recent work have studied the combined approach for specific methods and systems. However, there has been no work that propose a complete method or study the underlying design principles. In this paper, we present the result of our initial research on this subject. We choose Cassandra as a case study as it is a popular system that supports cloud-aware, very-large-scale NoSQL data management. We propose the CaMSAndra software development method that combines the MSA and Cassandra methods. We define a UML metamodel for CaMSAndra and uses it as the basis for discussing the design principles. We analyse the relationship between bounded context and application workflow and, based on this, define a hierarchical service design that builds a service hierarchy by transforming an application workflow. We also discuss a data-driven cluster design in connection to the microservices. We demonstrate CaMSAndra with a well-known software domain called Hotel Reservation. We contend that our method is promising for developing very large microservice-based, NoSQL-based systems in general.

Keywords: Software Design · Microservices Architecture · NoSQL · Cassandra

1 Introduction

Developing very large scale distributed software systems is challenging from both functional and data management perspectives. Methods based on Microservices

Architecture (MSA) have gained popularity for addressing the functional challenges. On the other hand, cloud-aware, very large scale NoSQL data management systems have also proved their effectiveness in tackling data management's scalability challenges. In fact NoSQL systems and their SQL counterparts have been studied in the context of the well-known CAP theorem [2]. This theorem basically states that distributed systems can at most achieve two out of the following three design properties: consistency (C), availability (A) and partition-tolerance (P). Most SQL-based systems are classified as CA with strict data consistency rules. In contrast, CP and AP systems both focus on partition-tolerance and provide non-strict forms of consistency. High consistency enforcement in very large scale distributed systems is extremely difficult to achieve without incurring some level of penalty in availability. In such systems, the continuity of operation in the face of failures and network changes is given a higher priority. Popular examples of AP systems include Cassandra, DynamoDB, CouchDB and Riak, while CP systems include MongoDB, Redis and HBASE. Among these, Cassandra, MongoDB and Redis are three popular very large scale NoSQL systems. In particular, Cassandra prioritises availability over consistency to provide what is known as eventual consistency [2].

Recent work have studied the combined approach for specific methods and systems. However, there has been no work that propose a complete method or study the underlying design principles. In this paper, we present the result of our initial research on this subject. We choose Cassandra as a case study for two main reasons. First, Cassandra is one of the most favourable NoSQL systems [1] that supports cloud-aware, very large data management scaling. Second, the authors have had extensive experiences in using and developing for this system. In particular, a leading multi-cloud, database-as-a-service version of Cassandra, named AstraDB, has been developed by Datastax.

We propose the CaMSAndra software development method that combines MSA and the Cassandra data modelling method. We define a UML metamodel for CaMSAndra and uses it as the basis to discuss the design principles. We analyse the relationships between the key concepts of the two component methods and discuss the principles for combining these concepts in the metamodel. In particular, we present a synthesis of bounded context and application workflow and, based on this, a hierarchical microservice design that transforms the application workflow to build the microservice hierarchy of a software. Further, we discuss a data-driven cluster design that explores the relationship between microservice and data distribution cluster in Cassandra. We demonstrate CaMSAndra with a well-known software domain called Hotel Reservation. We contend that our method is applicable to developing very large scale microservice- and NoSQL-based software systems in general.

The paper is structured as follows. Section 2 reviews the background knowledge and the related work. Section 3 presents an overview of our proposed our CaMSAndra method. Sections 4.1–4.3 discuss the core design principles in the context of CaMSAndra. Section 5 concludes the paper.

2 Background and Related Work

In this section, we introduce a motivating example and review a number of background concepts in the context of the related works. Contextualy, we define **very large-scale software** as a microservice-based software that stores a very large volume of data. The volume of data that is comparable to those managed by such global-scale software systems as Facebook and Netflix. In this context, we position our work as being related to Microservices Architecture and Cassandra.

2.1 Motivating Example: Hotel Reservation

To illustrate the concepts presented in this paper, we adopt the Hotel Reservation software example from Carpenter and Hewitt [5].

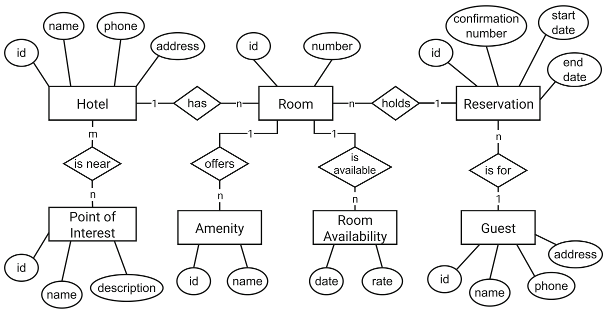

Fig. 1. Conceptual model of the Hotel Reservation domain (Adapted from [5]).

Figure 1 shows an entity relationship diagram (ERD) that represent the conceptual model of the hotel reservation domain. It consists of seven core concepts and the relationships between them. A hotel is located near some points of interests (POIs) and this relationship is used to answer user search queries about hotels and POIs. Once a hotel has been located, the user can proceed to make a reservation for the available rooms, each of which offers a number of amenities. Each reservation has a reservation period (start and end dates) and must include the guest details. A guest is the user who made the reservation.

2.2 Microservices Architecture

Microservices Architecture (**MSA**) is a modern scalable Internet-based software architecture. Each microservice represents a domain functionality that can be performed with a high degree of autonomy. In MSA, the software development process generally proceeds in a top down fashion, which starts with a high-level design to identify the bounded contexts. A **bounded context** defines the boundary of a microservice, which is represented by a domain model that captures the requirements of a business function or capability. Once the bounded contexts have been identified, the development process proceeds to tactical design to construct the domain model in each context. MSA has been discussed in the literature [3,4,9,10] to possess the following properties: (1) *service-based componentisation*, (2) *business-capability-driven*, (3) *distributed development*, (4) *modularity*, (5) *high autonomy*, (6) *infrastructure automation*, (7) *resilience*, (8) *observable*, and (9) *evolutionary design*.

Hierarchical Service Design. A number of recent works [4,8,13,15] have suggested to use a layered MSA style, in which microservices are organised into layers based on their domain dependencies. Two main benefits of this architecture style are that it helps (*i*) control the complexity of the system by reducing the service dependency and (*ii*) ease security enforcement. The former has recently been reported in [13] as a topic that requires further research. The latter is recently studied in the IoT context [11,14], where secured and resource-efficient access to the edge devices is a main concern. It is noted that both [11,14] use a form of layered, tree-based architecture to effectively organise services and manage the network complexity. A **service tree** [8] is a rooted tree in which the root node is a service and the non-root nodes are either another service or a non-service software module. An edge in the service tree represents functional dependency between its two nodes.

2.3 Cassandra Method for Data Modelling

Among the key concepts of Cassandra that are relevant to software design are query-driven data modelling with application workflow and peer-to-peer data distribution.

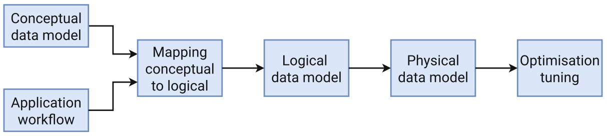

Fig. 2. Query-driven domain modelling method of Cassandra (Adapted from [5]).

As far as software development methodology is concerned, a unique feature of Cassandra-based system is its query-driven domain modelling. We prefer the more general term "domain modelling" to Cassandra's "data modelling" because, as will be explained below, in our view the method is actually a combination of data and behaviour modelling. In Cassandra, the idea is to combine the traditional conceptual data modelling (typically expressed in entity-relationship diagram (ERD) [6]) and the domain-specific behaviour requirements. These requirements constitute what is called in the Cassandra's literature the *application workflow*. We define **application workflow** as a layered, directed graph that represents a functional decomposition of a software, in which the bottom-level nodes are queries over the conceptual model of the software. In this paper, we will call this graph the **workflow model**.

To ease discussion in this paper we will refer to the Cassanda's domain modelling method simply as the **Cassandra method**. Further, we will refer to the aforementioned combined model of the Cassandra method as *Cassandra domain model*. When the context is clear, we will refer to this simply as **domain model**. Conceptually, we define the **Cassandra domain model** as a unified model consisting of a data model, describing the domain concepts and relationships among the concepts, and a relevant application behaviours that constitute a query-driven directed graph of functional decomposition over this data model.

For example, Figs. 1 and 3 show two component models that make up the domain model of hotel reservation. Figure 3, in particular, depicts the workflow model, expressed as a directed graph of function decomposition. Following the arrow paths lead us to the query functions that are defined in terms of the concepts in the conceptual model. For instance, the query function "*Q7. Find reservation by guest name*" defines a query on the two entities Reservation and Guest and the relationship between them. To ease discussion, we will refer to query function simply as **query**.

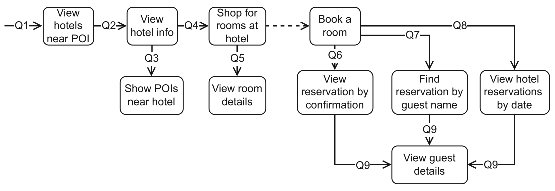

Fig. 3. Query-driven domain modelling for Hotel Reservation (Adapted from [5]).

Very Large Scale Data Management. As far as data management is concerned, Cassandra is well-known for its very large, horizontal scaling data management method. The method employs a peer-to-peer (P2P) data distribution scheme, which means that the overal storage capacity scale linearly with the number of nodes that participate in the system. Horizontal scaling is easier to manage than vertical scaling, which depends on a few high-performance server nodes. In Cassandra, data records (a.k.a *rows*) that share the same key prefix (called the *partition key*) form a data *partition*. Each partition is stored in a node that is responsible for the target token range containing the partition key. The partition key is hashed into token using a consistent hashing function.

DataStax Enterprise (DSE). DSE [7] and its cloud-based product line, named AstraDB[1], are highly scalable Cassandra-based data management systems. In particular, AstraDB is a multi-cloud database-as-a-service platform that consists

[1] https://www.datastax.com/products/datastax-astra.

in a suit of tools to ease system administration. These include DataStax's in-house tools as well as other open-source tools, notably those from the Apache Foundation.

3 Method Overview: CaMSAndra and the Metamodel

A key issue to address when designing microservice-based software with Cassandra is how to map the MSA concepts to the Cassandra method. Figure 4 shows a combined MSA-based and Cassandra-based software development method. We call this the Cassandra-MSA method or **CaMSAndra method** for short.

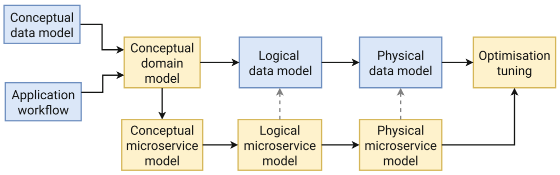

Fig. 4. The CaMSAndra method: MSA modelling with Cassandra.

CAMSAndra both revises and extends the Cassandra method shown in Fig. 2 to take into account the MSA design concerns. The extension involves adding a 3-component flow at the bottom that pertains to microservice construction. The three components of this flow are named after the corresponding three model versions of the Cassandra method. As shown in the figure, both logical and physical microservice models depend on the logical and physical data models for data storage design. The revision includes replacing the "Mapping conceptual to logical" component by the "Conceptual domain model" component, which explicitly reflects the existence of the Cassandra domain model. In addition, the "Optimisation and tuning" component is revised to consider both domain modelling and microservice modelling aspects.

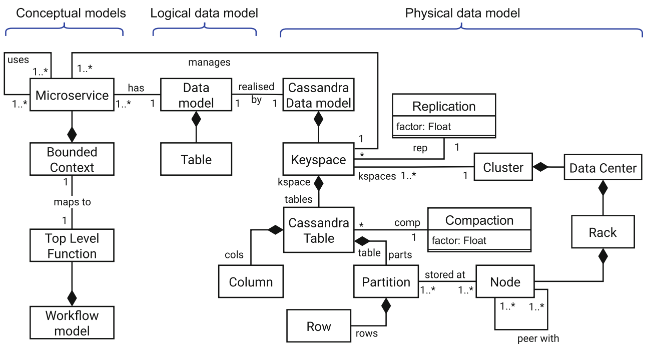

Fig. 5. The core CaMSAndra metamodel.

In the remainder of this paper, we discuss a number of core design principles in the context of the CaMSAndra method. Our discussion will focus on the relationships between key concepts of the Cassandra and MSA methods and how they lead to design insights for MSA-based software. To assist this investigation, we construct a metamodel that provides the foundational structure for the concepts under investigation. Figure 5 shows the UML diagram of the core metamodel of the CaMSAndra method. This model represents the concepts pertaining to microservices, workflow and data modelling components. The labelled curly brackets displayed at the top of the figure explains the connection between the metamodel and the models shown in Fig. 4. The two metamodel's substructures that pertain to logical and physical data models were constructed based on an analysis of Carpenter and Hewitt [5]. More specifically, within the scope of this paper we will investigate the principles that concern these essential relationships in Sect. 4.

To achieve preciseness, we use the Object Constraint Language (OCL) [12] to express the design rules associated with the metamodel. For conciseness, we use a short-hand notation to write OCL expressions on the model elements. For instance, the short-hand expression `Partition.table.cols->size()` (used in formula 1 of Sect. 4.3) means to apply the OCL's navigation rule to navigate from the context of a Partition to its associated Table and then to the set of Columns of this table and to perform the `size`() operation on this set.

4 Principles of CaMSAnda Design

4.1 Bounded Context Design with Workflow Model

Based on the definitions of bounded context and the Cassandra domain model, we map bounded context to a **top-level function** (**TLF**) of the workflow model that pertain to a well-defined domain behavior. Thus, as shown in Fig. 5, a workflow model consists in a set of TLFs. We argue that bounded context provides a necessary layer of abstraction on top of the workflow model that eases domain analysis and maintenance. Therefore, to extend domain requirements for new functions, we map them to relevant contexts or create new ones and add them to the corresponding workflow submodels. The general design rule is represented in Fig. 5, which states: one bound context per TLF.

For example, in the Hotel Reservation domain model shown in Fig. 3, we introduce two TLFs that represent two logical groupings of queries: hotel viewing and room booking. These TLFs represent the two bounded contexts of the domain. Figure 6 shows the bounded contexts of the example as dashed bounding boxes over two workflow submodels. The Hotel viewing context consists in the functions Q1-Q5, while the Room booking context consists in the functions Q6-Q9.

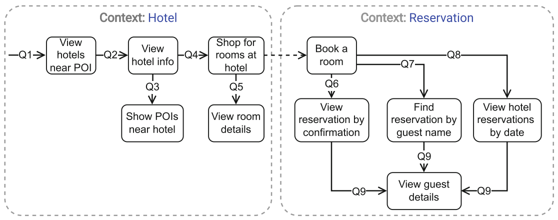

Fig. 6. Bounded contexts of Hotel reservation domain.

4.2 Hierarchical Microservice Design

After defining bounded contexts, the next step is to identify microservices. Typically, each bounded context contains one or more microservices. According to Carpent Hewitt [5], the logical data model (LDM) should be constructed from the domain model, and tables of the LDM are grouped to form the data boundaries of microservices. They recommend grouping denormalized tables representing the same data type to the same microservice.

We aim to generalize the identification of microservices in the conceptual modeling phase and establish specific rules for identifying them. We build on our recent work on hierarchical microservice design [8] and incorporate design considerations from the Cassandra method. The hierarchical workflow model of the Cassandra domain model serves as the basis for the service hierarchy. We convert the workflow model into a service tree using a 2-step procedure, which we call the **service construction procedure**:

1. **Determine a service** for each subset of functions that are associated to a main concept. This step generalises the service identification step described by Carpenter and Hewitt [5] to use the query functions in the workflow model. This helps move service identification step from the logical modelling phase to be performed earlier in conceptual modelling.
2. **Transform the service structure** (using the CRUD pattern) to expose the underlying concepts and, based on this, form a service tree. This step consolidates the functions to the underlying concept and makes this concept explicit in the service design. Focusing on the concept rather than the individual functions that it performs is necessary to avoid the anti-pattern of too-fine-grained service [10].

For example, Figs. 7(A) and 7(B) illustrate procedure. In Fig. 7(A), step 1 involves identifying three microservices based on the main concepts: Hotel, Point of Interest (POI), and Inventory. Step 2 involves transforming the Hotel service's structure to reveal the main concept and label it with the CRUD pattern. The Hotel service consists of the first 3 functions and two associated queries (Q1, Q2) that they serve. The POI service consists of one function that serves the query Q3, and the Inventory service consists of two functions that serve queries Q4 and Q5.

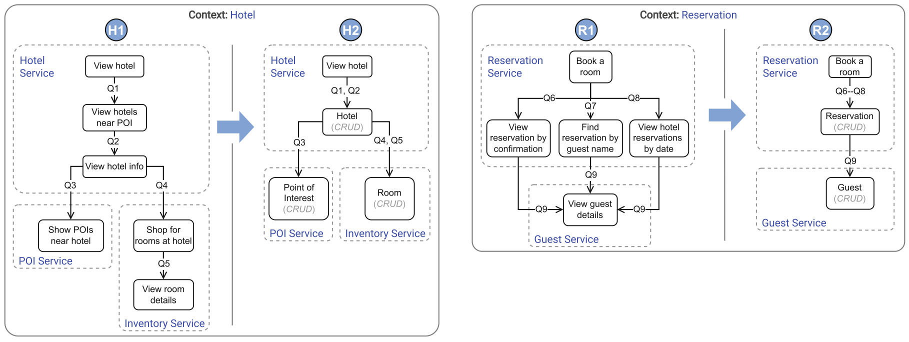

Fig. 7. (**A**) H1 → H2: Transforming Hotel services (LHS) to a service tree (RHS); (**B**) R1 → R2: Transforming Reservation services (LHS) to a service tree (RHS).

Similarly, Fig. 7(B) shows how the Reservation service tree is transformed using the service construction procedure. Step 1 involves identifying two services, Reservation and Guest. The Reservation service includes four functions and three queries (Q6-Q8), while the Guest service includes one function and query Q9. In step 2, the two services are transformed to reveal the underlying concepts and their CRUD operations.

4.3 Data-Driven Physical Design

In CaMSAndra, we observe that service autonomy for data distribution in Cassandra is limited to the cluster level, and beyond that lies the internal workings of the system. The physical data model in Fig. 5 shows that a microservice's data is stored in a keyspace within a single cluster, which can be either dedicated or shared [5]. However, microservices cannot control the placement of their data within specific data centers, racks, or nodes, which are internal to the Cassandra system and not the responsibility of the user application.

As far as microservice is concerned, therefore, an important cluster design concern is how to estimate a cluster's storage space based on the physical data model. To this end, we adapt the estimation technique presented in Chaps. 5 and 13 of Carpenter and Hewitt [5], which consists in 4 formulas for estimating the cluster size. Our contribution is to formulate the fourth formula (hinted at but not defined in [5]) and express all formulas more precisely using the CaMSAndra metamodel (see Fig. 5). In principle, the cluster size is estimated based on an indirect relationship that Cluster has with Partition, via Key Space and Cassandra Table, in the metamodel. Where suitable, we use OCL rules on the metamodel's structure to precisely express the formula terms.

Partition Size. Logically, the partition size (denoted by P_v) is determined by the number of cells (values) that it holds:

$$P_v = N_r \times N_g + N_s \qquad (1)$$

Where: P_v: number of values (or cells) in the partition, N_r = `Partition.rows ->size()`: number of rows of the current partition; N_c = `Partition.table.cols ->size()`: number of columns of the owner table; N_{pk}: number of primary key columns of the owner table; N_s: number of static columns of the owner table; and $N_g = N_c - N_{pk} - N_s$: number of regular columns of the owner table.

Physically, the partition size (denoted by P_t) is measured based on formula 1 as follows:

$$\begin{aligned} P_t = & \sum_{1 \le i \le N_{pk}} \texttt{sizeOf}\left(c_{k_i}\right) + \sum_{1 \le j \le N_s} \texttt{sizeOf}\left(c_{s_j}\right) \\ & + N_r \times \left(\sum_{1 \le k \le N_g} \texttt{sizeOf}\left(c_{r_k}\right) + \sum_{1 \le l \le N_{pk}} \texttt{sizeOf}\left(c_{c_l}\right) \right) \\ & + P_v \times \texttt{sizeOf}\left(t_{avg}\right) \end{aligned} \tag{2}$$

Where: c_k, c_s, c_r, c_c: partition key columns, static columns, regular columns, and clustering columns (*resp.*); t_{avg}: the average number of bytes of metadata stored per cell; N_r, P_v: number of rows and logical partition size (*resp.*) as per formula 1; and `sizeOf`(): function that returns the size (in bytes) of the CQL data type of the involved columns.

Finally, taking into account the replication factor of each partition, the partition size's estimation becomes:

$$P = P_t \times R \times C \tag{3}$$

Where: P_t: the physical partition size as per formula 2; R = `Partition.table.kspace.rep.factor`: the replication factor of the keyspace containing the owner table of the partition; and C = `Partition.table.comp.factor` $\in \{2, 1.25\}$: compaction factor of the compaction strategy of the owner table.

Cluster Size. The total storage size of a cluster is determined by summing the sizes of all the partitions that are stored across all keyspaces and tables in that cluster. Assume that the usable storage space of the disk can be estimated with 90% of the disk size, the cluster size (denoted by S) is measured as follows:

$$S = \frac{\sum_{t \in \mathcal{P}} P_t}{90\%} \tag{4}$$

Where:
$\mathcal{P}$ = `Cluster.kspaces->collect(tables)->flatten()->collect(parts)->flatten ()->asSet()`, i.e. the set of all the partitions across all tables and keyspaces of the cluster.

Discussion. We can extend the above technique to estimate the data storage size of a microservice and the storage size of a typical node. The relationship between Microservice and Keyspace in the metamodel would help define the former. On the other hand, the latter can be used, together with the cluster size, to estimate the number of nodes in a cluster. We plan to investigate these estimations in future work.

5 Conclusion

In this paper, we presented CaMSAndra method for developing very large scale microservice- and NoSQL-based software. Our method extended a state-of-the-art query-driven NoSQL data modelling method to take into account microservice design concerns. We constructed a UML metamodel for CaMSAndra and used it as the basis to discuss the core design principles. These principles arise out of the need to overcome the functional challenges associated with the metaconcepts and their relationships. Specifically, we presented a synthesis of bounded context and application workflow and, based on this, a hierarchical microservice design that transforms the application workflow to build the microservice hierarchy of a software. In addition, we discussed a data-driven cluster design that explores the relationship between microservice and data distribution cluster in Cassandra. We chose Cassandra as a case study as it is a popular system that supports cloud-aware, very-large-scale NoSQL data management. We demonstrated CaMSAndra with a well-known software domain called Hotel Reservation. We contend that our method is applicable to developing very large microservice- and NoSQL-based systems in general. Our plan for future work is to extend the method's scope to other comparable NoSQL systems that are in the CP and AP categories (e.g. MongoDB and Redis). We also plan to apply the method to develop real-world very large-scale software.

Acknowledgement. The authors wish to thank Jeffrey Carpenter [5] for granting us the permission to use the Hotel Reservation example in this paper. The authors would also like to thank the anonymous reviewers for their helpful feedbacks.

References

1. Ankomah, E., et al.: A comparative analysis of security features and concerns in NoSQL databases. In: Ahene, E., Li, F. (eds.) FCS 202. CCIS, vol. 1726, pp. 349–364. Springer, Singapore (2022). https://doi.org/10.1007/978-981-19-8445-7_22
2. Brewer, E.A.: Towards robust distributed systems. In: Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing, PODC 2000, p. 7. ACM, New York (2000)
3. Bruce, M., Pereira, P.A.: Microservices in Action, 1st edn. Manning, Shelter Island (2018)
4. Carnell, J., Sánchez, I.H.: Spring Microservices in Action, 2nd edn. Manning, Shelter Island (2021)
5. Carpenter, J., Hewitt, E.: Cassandra: The Definitive Guide: Distributed Data at Web Scale, 3rd edn. O'Reilly Media, Sebastopol (2022)
6. Chen, P.P.S.: The entity-relationship model - toward a unified view of data. ACM Trans. Database Syst. **1**(1), 9–36 (1976)
7. Kashliev, A.: Storage and querying of large provenance graphs using NoSQL DSE. In: IEEE 6th International Conference on BigDataSecurity, HPSC and IDS, pp. 260–262 (2020)
8. Le, D.M.: Managing complexity in microservices architecture: a nested MultiTree domain-driven approach. In: Proceedings of Conference on APSEC 2022, Japan. IEEE Computer Society (2022)

9. Lewis, J., Fowler, M.: Microservices (2014). https://martinfowler.com/articles/microservices.html
10. Newman, S.: Building Microservices: Designing Fine-Grained Systems, 1st edn. O'Reilly Media, Beijing; Sebastopol (2015)
11. Oma, R., Nakamura, S., Duolikun, D., Enokido, T., Takizawa, M.: A fault-tolerant tree-based fog computing model. Int. J. Web Grid Serv. **15**(3), 219–239 (2019)
12. OMG: Object Constraint Language Version 2.4 (2014). http://www.omg.org/spec/OCL/2.4/
13. Waseem, M., Liang, P., Shahin, M.: A systematic mapping study on microservices architecture in DevOps. J. Syst. Softw. **170**, 110798 (2020)
14. Whaiduzzaman, M., Barros, A., Shovon, A.R., Hossain, M.R., Fidge, C.: A resilient fog-IoT framework for seamless microservice execution. In: 2021 IEEE International Conference on Services Computing (SCC), pp. 213–221 (2021). ISSN 2474-2473
15. Zhou, X., et al.: Benchmarking microservice systems for software engineering research. In: Proceedings of 40th International Conference on Software Engineering, ICSE 2018, pp. 323–324. ACM, New York (2018)