applied sciences

MDPI

Article

An Approach to Migrate a Monolith Database into Multi-Model Polyglot Persistence Based on Microservice Architecture: A Case Study for Mainframe Database

Justas Kazanavičius *, Dalius Mažeika and Diana Kalibatienė

Department of Information Systems, Vilnius Gediminas Technical University, 10223 Vilnius, Lithuania; dalius.mazeika@vilniustech.lt (D.M.); diana.kalibatiene@vilniustech.lt (D.K.)

* Correspondence: justas.kazanavicius@vilniustech.lt, Tel.: +370-6-795-9290

Abstract: Migration from a monolithic architecture to a microservice architecture is a complex challenge, which consists of issues such as microservices identification, code decomposition, comminations between microservices, independent deployment, etc. One of the key issues is data storage adaptation to a microservice architecture. A monolithic architecture interacts with a single database, while in microservice architecture, data storage is decentralized, each microservice works independently and has its own private data storage. A viable option to fulfil different microservice persistence requirements is polyglot persistence, which is data storage technology selected according to the characteristics of each microservice need. This research aims to propose and evaluate the approach of monolith database migration into multi-model polyglot persistence based on microservice architecture. The novelty and relevance of the proposed approach are double, that is, it provides a general approach of how to conduct database migration from monolith architecture into a microservice architecture and allows the data model to be transformed into multi-model polyglot persistence. Migration from a mainframe monolith database to a multi-model polyglot persistence was performed as a proof-of-concept for the proposed migration approach. Quality attributes defined in the ISO/IEC 25012:2008 standard were used to evaluate and compare the data quality of the microservice with the multi-model polyglot persistence and the existing monolith mainframe database. Results of the research showed that the proposed approach can be used to conduct data storage migration from a monolith to microservice architecture and improve the quality of the consistency, understandability, availability, and portability attributes. Moreover, we expect that our results could inspire researchers and practitioners toward further work aimed to improve and automate the proposed approach.

Keywords: software migration; microservice architecture; polyglot persistence; SQL; NoSQL; cloud computing; software engineering

Citation: Kazanavičius, J.; Mažeika, D.; Kalibatienė, D. An Approach to Migrate a Monolith Database into Multi-Model Polyglot Persistence Based on Microservice Architecture: A Case study for Mainframe Database. *Appl. Sci.* **2022**, *12*, 6189. https://doi.org/10.3390/app12126189

Academic Editor: Paolino Di Felice

Received: 26 April 2022
Accepted: 15 June 2022
Published: 17 June 2022

1. Introduction

The International Data Corporation predicts that 90% of all new applications in 2022 will be developed based on microservice architectures [1]. Microservice architecture as well as software development and IT operations (DevOps) practice, improve software development agility and flexibility, and allows enterprises to bring their digital products and services to a very competitive market faster [1–3]. In order to remain competitive, companies have started to modernize their legacy monolithic systems by decomposing them into microservices [4–9].

Although the topic of monolithic software migration into microservice architecture has already been explored by scientists and software engineers, there is little research on

the database adaptation during the migration from a monolith architecture to a microservice architecture [4,10–24]. Despite this, it is recognized that data management is a major challenge in microservices [25–31]. The primary focus of most of the research is the microservice identification within monolith application and source code decomposition into microservices. All of the existing migration methods provide very little or no recommendations on how to adopt data storage to a microservice architecture during the migration from a monolith to microservice architecture. To the best of the authors' knowledge, aside from A. Levcovitz et al., who proposed a technique of microservice extraction from monolith enterprise systems, there have been no further migration methods that have investigated the adaption of data storage to a microservice architecture [32].

To bridge this gap, the authors of this research aim to propose and evaluate a new migration approach that would allow to migrate the data store from a monolith to a microservice architecture. In order to propose a migration approach, the authors first had to conduct a literature review to answer to questions: What data store to use? What database pattern to use? To evaluate the proposed approach, a proof-of-concept of migration from a monolith mainframe database to a multi-model polyglot persistence was performed. The mainframe application was chosen because it best represents a typical monolith legacy application as used by 71 percent of Fortune 500 companies [33–35]. Quality attributes defined in the ISO/IEC 25012:2008 standard were used to evaluate the data quality of the microservice with a multi-model polyglot persistence and compare it with the existing monolith mainframe application.

Our main scientific contribution and novelty of this paper are as follows:

1. A new approach on how to conduct database migration from a monolith architecture into a microservice architecture is proposed.
2. A new approach on how to transform the monolith data model into multi-model polyglot persistence is proposed.

This article is a continuity on the research of legacy monolith software migration to a microservice architecture. A literature review and monolith decomposition into microservices methods have been analyzed in previous articles [36,37].

The rest of this paper is organized as follows. Section 2 presents a literature review on the topics of data storage in microservices, SQL vs. NoSQL, and polyglot persistence. Section 3 provides a review of the related work. Section 4 describes the proposed approach and its evaluation. Section 5 reports the results of the experimental investigation. Section 6 provides an evaluation of the data quality results. Section 7 discusses the advantages and disadvantages of the proposed migration approach. Finally, Section 8 concludes this work.

2. Background

To better understand the decisions made by the authors while creating the proposed approach, this section provides the background of a literature review conducted on the following topics: SQL vs. NoSQL, polyglot persistence, and data storage in microservices.

2.1. SQL Vs. NoSQL

For the last 40 years, relational databases (SQL) have been the market leader because of their ability to solve most of the challenges. Such a long existence has given a high level of maturity and it is still the most recommended storage for many applications. However, SQL databases are not capable of solving all of today's challenges, and inspired by SQL limitations, NoSQL has emerged as a solution to fulfil these gaps [38–39].

The key feature of relational databases is the high consistency guarantee provided by ACID (Atomicity, Consistency, Isolation, and Durability) properties. Many NoSQL databases have focused on high levels of availability and resilience, even though this may compromise consistency for a few moments. To achieve availability and resilience, NoSQL

databases work with BASE (Basically available, Soft state, and Eventually consistent)) properties [39].

The CAP theorem (Consistency, Availability, and Partition tolerant), also known as Brewer's theorem, states that it is impossible to provide all three guarantees simultaneously [40]. While SQL primary is focusing on consistency, NoSQL is giving up either consistency or availability and embracing partition tolerance [38]. There is no perfect database that could solve all the problems and fits all the requirements. Polyglot persistence is a single storage system that combines the SQL and NoSQL database features.

In relational databases, the stored data are managed and represented as tables. Each table can have a relation to an arbitrary number of tables. A table consists of rows and columns. A row represents a dataset item, a column represents a dataset item's field. In NoSQL, the data store management can be grouped into four types: Key-value, Wide-column, Document, and Graph. Data in key-value stores is managed and represented as key and value pairs stored in efficient, highly scalable, key-based lookup structures. A value represents data with an arbitrary type, structure, and size is uniquely identified by an indexed key. Indexing and querying based on values are not supported, so in cases where querying is needed, it has to be implemented on the client side. The conception of wide-column stores (also known as column-family stores) was taken from the Google Bigtable store. Data are represented in a tabular format of rows and column-families. A column-family is an arbitrary number of columns logically connected to each other. A wide-column store is an extended key-value store in which the value is represented as a sequence of nested (key, value) pairs. An extended key-value store in which the value is represented as a document encoded in standard formats such as XML, JSON, or BSON (Binary JSON) is a Document store. The biggest difference from the key-value store is that document stores know the format of the documents and support querying based on value functionality. Graph stores are based on graph theory, in which a graph consists of vertices representing entities and edges represent the relationships between them. The datasets of graphs are stored efficiently to provide effective operations for their querying and analyzing. Because the data relationship variety can be very different in many aspects, many types of graphs such as undirected, directed, labelled, etc. are used to represent a different type of data [39–45].

According to Nayak et al., the advantages of NoSQL are: it provides a wide range of data models to choose from, is easily scalable, no database administrators are needed, can handle hardware failures, is faster and more flexible, and evolves at a very high pace. The disadvantages of NoSQL are its immaturity, no standard query language exists, not all NoSQL databases are ACID compliant, a standard interface not exist, and difficult maintenance [45].

According to the DB-Engines initiative, relational databases are still the most popular database technology, even though NoSQL popularity has emerged in the last few years. Twelve out of the 20 most popular database technologies are relational databases. The most popular NoSQL database is MongoDB (Document store). There is only one representative for each NoSQL technology in the TOP 20 list: #5 MongoDB (Document store), #7 Redis (Key-Value store), #9 Casandra (Wide-Column store), and #20 Neo4j (Graph store) [46]. Based on the data from the DB-Engines initiative, we can state that relational databases are more mature, have more products to choose from, and have a bigger user community. All of these three points are valid points when considering what database to choose.

2.2. Polyglot Persistence

The general polyglot persistence conception was evaluated from the polyglot programming conception proposed by Neal Ford in 2006. The main idea of both conceptions is choosing the right tool for the given task. In polyglot programming, it is a programming language, in polyglot persistence, it is a data storage engine. Polyglot persistence defines

a hybrid approach where different kinds of data are best dealt with in different data stores [47–48].

No single database technology, be it SQL or NoSQL, can satisfy all of the business needs and solve all technological challenges. To choose the right database, a set of criteria has to be considered: the data model, CAP support, capacity, performance, query API, reliability, data persistence, rebalancing, and business support. It is also important to evaluate databases from different viewpoints: technical, business, system domain, and environmental. Polyglot persistence technology has the potential to scale to millions of users a day and be able to store an incredible amount of data by combining SQL and NoSQL technologies into one solution [38–49].

L. Wiese categorized polyglot database architectures into three types: polyglot persistence, lambda architecture, and multi-model databases [48]. L. Wiese recommends using polyglot persistence only if several diverse data models have to be supported, otherwise there is a risk to overhead maintenance. The lambda architecture is recommended for real-time data analytics applications. The lambda architecture relies on the same data stores as polyglot persistence and has similar disadvantages. Multi-model databases store data in a single store but provide access to the data with different APIs according to different data models. This type of polyglot database architecture is recommended if only a limited set of data models is required by the accessing applications.

C. Zdepski et al. proposed a modeling methodology capable of unifying design patterns for polyglot persistence, bringing an overview of the system as well as a detailed view of each database design [47]. The proposed methodology consists of three steps: (1) conceptual design, (2) logical design, and (3) physical design. The conceptual design translates the requirements into a conceptual database schema. The logical design realizes the translation of the conceptual model to the internal model of a database management system. The physical design implements all peculiarities of each database software.

According to C. Shah et al., a crucial part of the efficiency of a polyglot system is the selection of a database engine [40]. The authors proposed the design of a polyglot persistence system for an e-commerce application and compared it with a system where data were stored only in the SQL or NoSQL databases. The most optimum results were obtained from the polyglot system with three databases: (1) document type (Mongo DB), and (2) key-value type (Redis), and (3) relational database (SQLite).

An evaluation of the NoSQL Multi-Model data stores in polyglot persistence applications were conducted by F. R. Oliveira et al. [30]. Multi-model databases (ArangoDB and OrientDB) were compared with a combination of the document type database (MongoDB) and graph type database (Neo4j). The experimental results showed that in some scenarios, multi-model data stores had similar or even better performance than a combination of different data stores.

2.3. Data Storage in Microservices

A microservice architectural style is an approach for developing an application as a suite of small services where every service communicates with other services via lightweight mechanisms such as HTTP API. Services are built around business capabilities and are independently deployable by fully automated deployment machinery. There is a bare minimum of the centralized management of services that may be written in different programming languages and use diverse data storage technologies [2].

H. Chawla et al., in the book *Building Microservices Applications on Microsoft Azure*, discuss the various critical factors of designing a database for microservice architecture-based applications [50]. The authors recommend that each microservice should have a separate database because data access segregation helps fit the best technology to handle the respective business problem. The authors, based on CAP theorem, suggested choosing an intersection of two functionalities: consistency and availability, or availability and partition. The database should depend on the nature of the application. While monolith applications usually use a single data store, microservices use many data stores, both SQL

and NoSQL. SQL is recommended for use where transactional consistency is critical and structured data are stored. NoSQL is recommended for microservices where schema changes are frequent, maintaining transactional consistency is secondary, and semi-structured or unstructured data are stored. Microservice architecture offers the flexibility to use polyglot persistence.

According to H. Chawla et al., there are four main challenges to using microservice architecture and polyglot persistence: (1) maintaining the consistency for transactions spanning across microservice databases; (2) sharing, or making the master database records available across microservices databases; (3) making data available to reports that need data from multiple microservices databases; and (4) allowing effective searches that receive data from multiple microservices databases [50]. To ensure efficient changes transferring across the microservices, the authors suggest using two approaches: (1) a two-phase commit for managing transactions in SQL databases, and (2) eventual consistency to managing any distributed application.

R. Laigner et al. attempted to bridge the gap of a lack of thorough investigation on the state of the practice and the major challenges faced by microservice architecture practitioners about data management [25]. The authors identified three main reasons as why a microservice architecture should be adopted regarding data management: (1) functional partitioning is used to support scalability and high data availability; (2) decentralized data management provides the ability to manage data store schemas for each microservice independently; (3) even driven architecture allows one to build a reactive application.

Database and deployment patterns were investigated by R. Laigner [25]. Three mainstream approaches for using database systems in microservice architectures were identified: (1) private tables per microservice, sharing a database server and schema; (2) schema per microservice, sharing a common database server; and (3) database server per microservice. Based on the conducted survey, the authors stated that the most preferred and efficient way for data persistence in a microservice architecture is to encapsulate a microservice state within its own managed database server and avoid any resource sharing between different microservices. The most widely used databases in a microservice architecture are Redis, MongoDB, MySQL, PostgreSQL, and MS SQL.

K. Brown et al. researched the implementation patterns for microservice architectures and proposed a pattern language [51]. Part of the proposed pattern language consisted of scalable store patterns used to build a scalable and stateless data store for a microservice architecture-based application. The key in these patterns is that the database must be naturally distributed and able to both scale horizontally and survive the failure of a database node. The authors suggest choosing a database based on the need: if the application strongly depends on the SQL-centric complex query capability, then a solution such as a SQL database or a distributed in-memory SQL database may be more efficient, otherwise, the recommendation is to use NoSQL databases.

The importance of data persistence choice in microservice architecture-based applications was highlighted by N. Ntentos et al. in the article "Assessing Architecture Conformance to Coupling-Related Patterns and Practices in Microservices" [52]. According to the authors, three things have to be taken into account while choosing data storage: reliability quality, scalability quality, and adherence to microservice architecture best practices. The most recommended option is the database per service pattern and the second option is to use a shared database, but this way negatively affects the loose coupling quality.

A. Messina et al. proposed and tested a simplified database pattern for microservice architecture where a database is a separate microservice itself [53]. The proposed data persistence pattern was based on four patterns: (1) the API gateway pattern; (2) the client-side discovery and server-side discovery patterns; (3) the service registry pattern; and (4) the database per service pattern. Proposed pattern benefits are: no traditional service layer, microservices has no third-party dependencies, database microservice encapsulated all specific database details, less involved components, and less complexity. The main

drawback is the dependency on the chosen database. Proof-of-concept showed an improved performance compared with the standard SQL based storage.

L. H. Villaca et al. evaluated the use of a multistore database canonical data model in microservice architecture [54]. The authors proposed and implemented an architecture for microservices with polyglot persistence based on the strategy of a canonical data model. The benefits found during the evaluation were: (1) usability—high understandability and operability; (2) high performance—better resource utilization and shorter response time; (3) compatibility—the proposed architecture has enabled systems implemented with different technologies to coexist in an encapsulated form; and (4) maintainability—the API structure provides processing of the linked objects (as defined in the scheme) in a segregated manner, facilitating the decomposition processing logic and improving the readability of the mediator node code.

A different approach of data persistence in microservice architecture was presented by N. Viennot et al. in the paper: "Synapse: A Microservices Architecture for Heterogeneous-Database Web Applications" [55]. The authors developed a framework called Synapse, which supports data replication among a wide variety of SQL and NoSQL databases including MySQL, Oracle, PostgreSQL, MongoDB, Cassandra, Neo4j, and Elasticsearch. With Synapse, different microservices that operate on the same data but demand different structures can be developed independently and with their own database. Synapse transparently synchronizes shared data subsets between different databases in real-time. Synchronization is conducted via a reliable publish/subscribe communication mechanism. The biggest advantage of the synapse is that it enables microservices to use any combination of heterogeneous databases (SQL and NoSQL) in an easy-to-use framework.

2.4. Summary

To sum up, it can be stated that there is no one criteria based on which SQL or NoSQL could clearly be chosen as a database for microservice. Instead, there are recommendations when SQL or NoSQL could be a better option. For example, the SQL is a recommended technology if transactional consistency is critical and NoSQL is a recommended technology if schema changes are frequent, etc. In theory there are clear boundaries between SQL and NoSQL, but in practice it is much more complicated. For example, even though the transaction consistency is considered as a benefit of SQL, there are NoSQL databases such as RavenDB or MongoDB that also support it.

On the other hand, the nature of microservice architecture offers the flexibility to use a polyglot persistence and leverage different data store models and engines. Polyglot persistence based on supported models can be grouped into two types: single-model and multi-model. The biggest advantage of multi-model polyglot persistence is that it uses only one database engine to support all models, while in single-model polyglot persistence, each model is supported by a separate database engine. According to L. Wiese, multi-model polyglot persistence is recommended only if a limited set of data models is required to access.

There are many different suggestions on how to implement data persistence for microservice architecture, but a common consensus among practitioners is that good practice is to use a separate database for each microservice. However, an actual implementation depends on many different factors such as the size of a microservice, the actual need of the database for each microservice, the limitations of the existing infrastructure and architecture, security requirements, consistency requirements, code quality, etc. The most common patterns used for data persistence in microservice architecture are: table per microservice, schema per microservice, database per microservice, and database as microservice.

What data storage to use? Each microservice can be different in a variety of aspects and there is no one database that could potentially satisfy all the needs, which naturally leads to the usage of polyglot persistence as a microservice data store. The multi-model polyglot

persistence model was chosen to be used because it uses only one data store, which supports different data model types and at the same time, according to F. R. Oliveira et al., multi-model data stores have similar or even better performance than a combination of single-model data stores.

What database pattern to use? The authors decided to use the simplified database pattern for microservice architecture proposed by A. Messina et al. where a database is a separate microservice itself. The decision to use this pattern was made based on the benefits this pattern provides: less complexity and better performance results.

3. Related Work

Despite the increased attention on the migration from monolith architecture to microservice architecture, there is a lack of research that has focused on the database part. The primary focus of most of the research is source code decomposition or extraction from monolith applications [4,10–24]. R. Laigner et al. stated that essentially, there is little research on the characteristics of data management in microservices in practice [25].

For example, when P. Cruz et al. conducted a migration from a monolith to microservice architecture, the database was not changed at all [25]. In another example, G. Mazlami et al. proposed a microservice extraction from the monolith application approach, which excluded the database part and considered ORM technology as a representative layer of the database tables [21]. In C. Fan et al., they proposed migration from a monolith to a microservice architecture method using the domain-driven design (DDD) principle to identify microservices and amend the database schema accordingly, but did not provide any instructions or recommendations on how to amend database or which database technology to choose [22].

A thorough search of the relevant literature yielded only one related method, which contained detailed instructions on how to adapt data storage during the migration from the monolith architecture to microservice architecture [32]. A. Levcovitz et al. proposed a method that only groups tables within the microservice scope and keeps the data in the same data storage. The authors of this paper aim to contribute by filling this gap by proposing and evaluating a new migration method that would provide guidelines on how to transform database storage from a monolith relational database management system into a polyglot persistence, which is more suited for microservice architecture.

In addition, we conducted a proof-of-concept migration from a legacy monolith mainframe application to a microservice architecture. The term mainframe is used in this paper to refer to the computers that have IBM System 390, zSeries, or compatible computer architecture, which was invented in the 1950s. The mainframe application was chosen as a source because it best represents a typical monolith legacy application, as it was used by the majority of the biggest companies [33–35]. Moreover, the mainframe uses a relational database, which is the most widely used database type [46]. As microservice architecture is becoming a standard architecture and many enterprises still run their digital services through a mainframe, the challenge of how to migrate from the mainframe to microservice is important, as it never was.

4. Proposed Migration Approach and Its Evaluation

The authors of this paper proposed and evaluated the approach of monolith database migration to multi-model polyglot persistence based on the microservice architecture. The purpose of the proposed approach is shown in Figure 1.

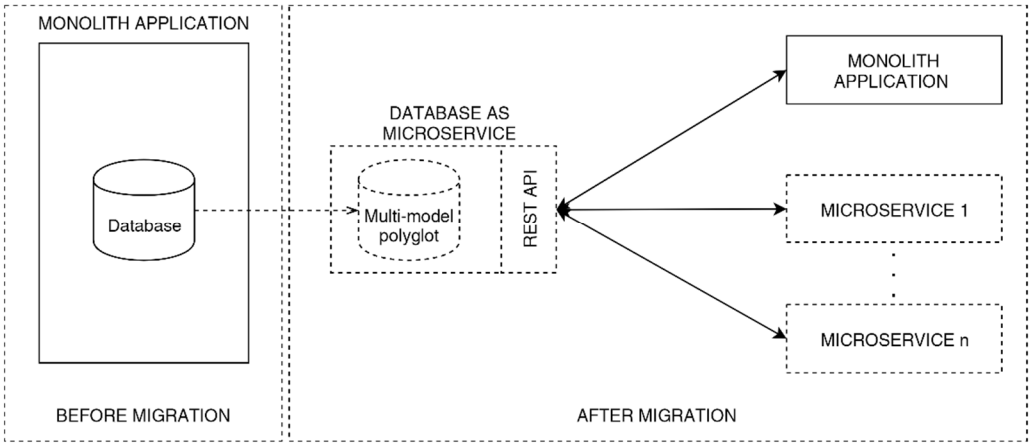

Figure 1. The purpose of the proposed approach.

The approach can extract a database from a monolith application and transform it into a multi-model polyglot persistence, which is encapsulated as a microservice itself and exposes data access through a representational state transfer (REST) application programming interface (API). Multi-model polyglot persistence allows us to better utilize the benefits of microservices such as agility and scalability [18,26,39–40]. The encapsulation of a database into a microservice reduces the complexity and increases the performance [53]. After migration, the data are accessible not only to an existing monolith application but also to any microservice within an ecosystem. This allows source code migration to be conducted from the monolith architecture to microservice architecture gradually, without taking into account the database that is already adopted into the microservice architecture.

As a proof-of-concept for the proposed approach, the migration has been executed from an existing mainframe monolith application to a new microservice architecture-based application with multi-model polyglot persistence. The migration results were evaluated by the chosen criteria.

This section describes the proposed migration approach and its evaluation. The migration approach used in the experiment is explained in Section 4.1, the criteria used for the evaluation are described in Section 4.2, the multi-model polyglot database software is described in Section 4.3, and the tools, libraries, and IT equipment used to perform the experiment are listed in Section 4.4.

4.1. Migration Approach

The proposed approach of migration from a monolith database to multi-model polyglot persistence based on microservice architecture is shown in Figure 2. It consists of six steps where each step is divided into sub-steps. A detailed explanation of each step and its sub-steps are provided below.

4.1.1. Analysis of an Existing Monolith Application

The aim of the first step is to identify the functional requirements and the data model of an existing monolith application. Domain experts and IT experts have to work together in order to identify all functional requirements and build the most optimum data model. The first sub-step is to conduct a business analysis and identify business processes that depend on an existing monolith application. Understanding business logic is crucial to listing out the essential business rules. The second sub-step is code base and database analysis with the aim of identifying the technical implementation details such as code components or database tables, related to essential business rules. Based on the identified essential business rules and technical implementation details, the functional requirements have to be specified in sub-step three. Finally, in sub-step four, the domain data model has to be defined based on an existing database data model and the business processes

that it caters for. To achieve the optimal data model results, a top–down approach has to be used instead of a bottom–up one [56].

4.1.2. Data Model Development

During the second step, the data model for multi-model polyglot persistence has to be created based on the defined model of an existing monolith database. The proposed data model creation process is inspired by C. Zdepski et al., who proposed an approach to model the polyglot persistence that consists of five sub-steps (Figure 3):

1. *Conceptual design*—based on the gathered functional requirements to build a conceptual database schema as an entity–relationship model. A conceptual database schema is a foundation that will be used in the next sub-steps to develop a new data model.
2. *Segmentation design*—divides the conceptual database schema into independent function units and defines borders between these units. The cut points defined on the existing data model during segmentation design will be used to split the current data model into different data models suitable for multi-model polyglot persistence.
3. *Target data model design*—choose the best data structure for each identified segmentation unit from the different data structures supported by multi-model polyglot persistence.
4. *Physical design*—implement the built target data model into a multi-model polyglot persistence database. As each database is different, the aim of this sub-step is to implement all technical peculiarities needed to support the developed target data model in the database.

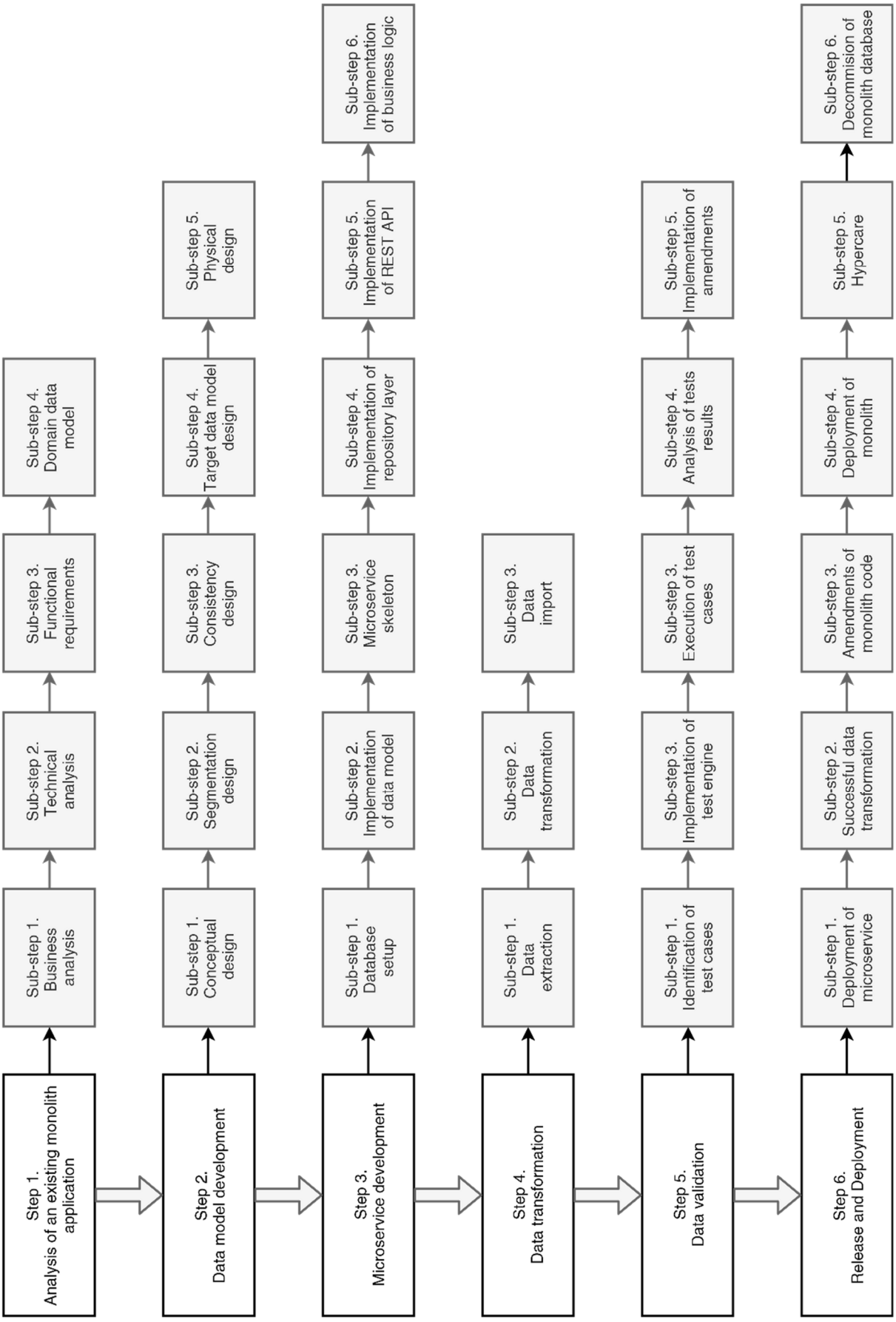

Figure 2. The proposed migration approach.

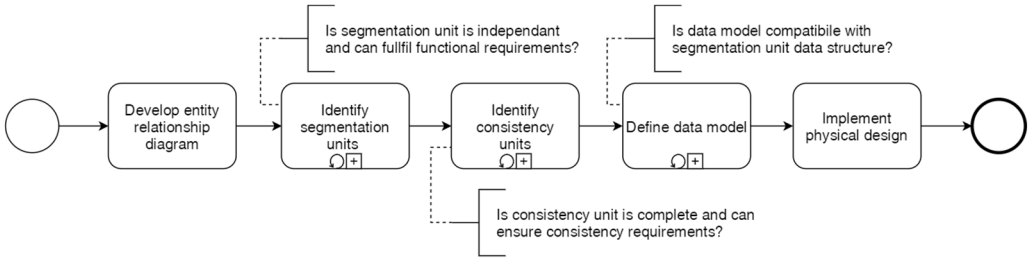

Figure 3. The data model development.

4.1.3. Microservice Development

The main goals of step three are to set up the multi-model database and encapsulate it into the microservice. This allows us to implement the database as a service pattern, where a database is a microservice itself.

The first sub-step is to install a multi-model polyglot database and set up technical peculiarities such as creating cluster, users, firewall rules, etc. The database setup can be different in many aspects such as the operating system, virtual machine or Docker, cluster or single instance, cluster type, etc. The decision on how to install and set up a database has to be determined based on the application of non-functional requirements, the capabilities of the existing company infrastructure, database capabilities, availability requirements, security requirements, scalability requirements, etc.

During the next sub-step, the physical design of the data model created in the second step has to be implemented into the installed database. All models and data structures defined in step 2 have to be implemented and ready to be used. This sub-step could be skipped if the database supports a code-first approach where models and data structures are defined in an application.

The purpose of the third sub-step is to create a microservice skeleton that has to be able to be deployed and run as a Docker container. At this stage, a microservice should only contain the code and settings needed to run it as a Docker container in the company's infrastructure. An infrastructure has to be created to run a Docker container, for example, it could be an OpenShift project in a private cloud. The number of active containers and scalability settings have to be determined based on the non-functional requirements, the capabilities of the existing company infrastructure, database capabilities, availability requirements, security requirements, scalability requirements, etc. A continuous integration and continuous deployment (CI/CS) pipeline has to be set up in order to automate build, test, and deploy activities and ensure security that only the entitled person can deploy a new version of the microservice. Microservice capabilities to log have to be ensured. A good practice for a microservice architecture is to use centralized logging solutions. An example of a good logging solution could be ELK Stack.

The repository layer has to be built in order to provide microservice accessibility to the database in sub-step four. All actions needed to establish a connection between a database and a microservice have to be executed first, for example, firewall rules, service account access rights, connection string, etc. The next step is the implementation of a repository layer. The code able to communicate with a database and manipulate its data has to be written. For each data model defined in step 2 and implemented in the database, a repository has to be created and support four main operations: create, read, update, and delete.

During sub-step five, the REST API has to be built and exposed with all of the necessary methods to support the interfaces for all identified functional requirements. For example, if the functional requirements consist of *creating customer*, *viewing customer*, *updating customer*, *deleting customer*. All four methods have to be created in the customer's controller. The authentication and authorization functionality have to be implemented in order to fulfil the security requirements and manage the accessibility to different methods.

The aim of the last sub-step is to implement the business logic layer, which has to connect the REST API layer and the repositories layer. Because the REST API layer operates with business domain data models and the repository layer operates with database specific data models, they cannot work directly. The business logic layer works as an intermediate layer that contains all of the logic needed to implement all of the functional requirements identified in step one and connects the REST API and repository layers.

An example of one possible implementation is presented in the sequence diagram below (Figure 4). The REST API layer exposes a method *GetCustomer*, which can be called by a client application to obtain all the customer details. Once the call is received, it is routed to the business logic layer, which calls the repository twice to obtain different details about the customer: *GetCustomerInfo* and *GetCustomerHistory*. *GetCustomerInfo* obtains the general customer information such as name, surname, address, etc. *GetCustomerHistory* obtains the customer's payment history. The repository layer is called twice because *CustomerInfo* and *History* data are stored in separate data models within a database and two separate calls to a database are needed. In the business layer, *CustomerInfo* and *History* data received from the repository layer are combined and mapped into one consistent domain data model—*Customer*, which is used as a response to a client's *GetCustomer* request. To sum up, the repository layer is responsible for data manipulation within the database, it encapsulates all of the technical implementation peculiarities such as connection establishment, data mapping, etc. The REST API layer is responsible for data exposure to clients via the REST API interface and encapsulates all technical implementation peculiarities such as connection establishment, authorization, authentication, etc. The business logic layer is responsible for building a consistent domain data model.

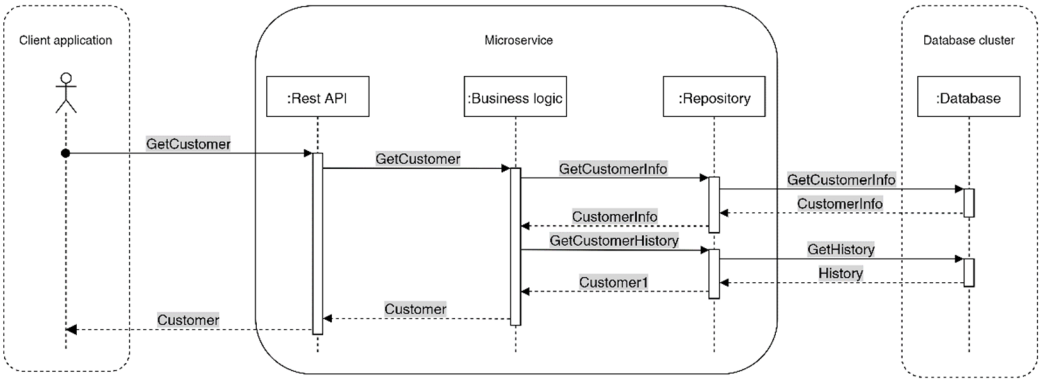

Figure 4. An example of the function *GetCustomer* implementation within the microservice layers.

4.1.4. Data Transformation

Once a microservice is created, the next step is to transform the data from a monolith database into multi-model polyglot persistence. The biggest challenge here is that both databases use different data models, so it is not possible to directly transfer data from one to another, it has to be transformed. The aim of this step is to create an application that can execute data transformation between databases. The proposed data transformation process is shown in Figure 5.

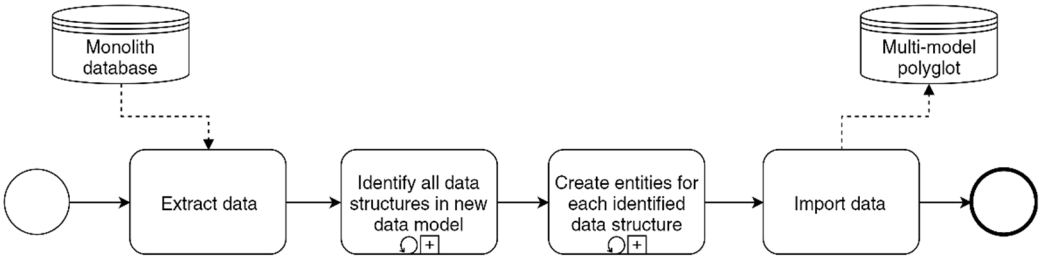

Figure 5. The proposed data transformation process.

The first sub-step is to extract everything needed to transform the data from a monolith database. A code that is able to read data from a monolith database and transform it into data models that represent the used data structure has to be written. The authors recommend creating a repository layer with a repository for each data table in a monolith database. An example of the simplified repository and model implementations written in the C# programming language are shown in Figure 6.

```
public class MonolithModel1Repository
{
    private readonly IDatabase<MonolithModel1> _database;

    public CustomerRepository(IDatabase<MonolithModel1> database)
    {
        _database = database;
    }

    public IEnumerable<MonolithModel1> GetAllRecords()
    {
        return _database.GetAllRecords(tableName: "Model1Table");
    }
}
```

```
public class MonolithModel1
{
    public long Id { get; set; }
    public string PropertyA { get; set; }
    public long MonolithModel2Id { get; set; }
}
```

Figure 6. An example of the simplified repository and model implementations.

The *MonolithModel1* is a model that represents the data in the *Model1Table* data table. The *MonolithModel1Repository* has one method, *GetAllRecords,* which calls the generic interface *IDatabase* that executes the SQL query to obtain all records from the specific table *Model1Table* and maps the result to the defined model *MonolithModel1*. Finally, read-only access rights should be granted and firewall rules should be set up in order for the application to access the data in a monolith database.

The purpose of the next sub-step is to transform the extracted data into a data model that is supported by multi-model polyglot persistence. As the data models and repository layer for multi-modal polyglot persistence have already been implemented in the third step, the code can be reused. Once both data models for the monolith database and multi-modal polyglot persistence are created, the mapping logic between models has to be implemented. Each field of each data model for polyglot persistence has to be mapped.

The simplified example of the data model of multi-model polyglot persistence is shown in Figure 7. It is a combination of two data models used in monolith application. The *MonolithModel1* and *MonolithModel2* models represent two data tables in the monolith database and *PolyglotModel* represents a document with the embedded sub-document *PolyglotChildModel*. Even though the given example looks simple and straightforward, in practice, mapping logic can be more complicated: the data model for the polyglot can be a combination of dozens of data tables, fields from the same data table can be part of many data models of the polyglot, data types for fields could be different, etc. The complexity of data model mapping strongly depends on the quality of the monolith database data model where a lower quality means a higher complexity.

```
public class MonolithModel1
{
    public long Id { get; set; }
    public string PropertyA { get; set; }
    public long MonolithModel2Id { get; set; }
}
public class MonolithModel2
{
    public long Id { get; set; }
    public string PropertyB { get; set; }
    public string PropertyC { get; set; }
}
```

```
public class PolyglotModel
{
    public long Id { get; set; }
    public string PropertyA { get; set; }
    public PolyglotChildModel Child { get; set; }
}
public class PolyglotChildModel
{
    public long Id { get; set; }
    public string PropertyB { get; set; }
    public string PropertyC { get; set; }
}
```

```
public PolyglotModel Map(MonolithModel1 model1, MonolithModel2 model2)
{
    return new PolyglotModel
    {
        Id = model1.Id,
        PropertyA = model1.PropertyA,
        Child = new PolyglotChildModel
        {
            Id = model2.Id,
            PropertyB = model2.PropertyB,
            PropertyC = model2.PropertyC,
        }
    };
}
```

Figure 7. An example of simplified data model mapping.

The next action in the second sub-step is to create all records for polyglot persistence based on records in a monolith database. If we follow the examples defined in Figures 6 and 7, the number of records for the *PolyglotModel* model should be equal to the number of records in the *Model1Table* table. For each data model of polyglot persistence, a main data table in a monolith database has to be identified. A simplified example of records creation is shown in Figure 8. The *PolyglotModelTransformer* class uses *MonolithModel1Repository* and *MonolithModel2Repository* classes to obtain *MonolithModel1* and *MonolithModel2* records from the monolith database and passes these to the *PolyglotModelMapper*, which maps all of the fields and creates *PolyglotModel* records.

The last sub-step imports all records created in the second sub-step into a multi-model polyglot database installed in the third step. The authors suggest reusing the repository layer created in the microservice.

Even though the data transformation process could be implemented in different ways, the authors recommend building a separate application for this purpose. This would allow for the process to be repeated as many times as needed in event that errors or failures occur. It also would allow for the transformation process to be executed gradually in case it is planned to transform the data in stages.

```
public class PolyglotModelTransformer
{
    private readonly MonolithModel1Repository _monolithModel1Repository;
    private readonly MonolithModel2Repository _monolithModel2Repository;
    private readonly PolyglotModelMapper _polyglotModelMapper;

    public PolyglotModelTransformer(
        MonolithModel1Repository monolithModel1Repository,
        MonolithModel2Repository monolithModel2Repository,
        PolyglotModelMapper polyglotModelMapper
        )
    {
        _monolithModel1Repository = monolithModel1Repository;
        _monolithModel2Repository = monolithModel2Repository;
        _polyglotModelMapper = polyglotModelMapper;
    }

    public IEnumerable<PolyglotModel> Transform()
    {
        var monolithModels1 = _monolithModel1Repository.GetAllRecords();
        var monolithModels2 = _monolithModel2Repository.GetAllRecords();

        foreach (var model1 in monolithModels1)
        {
            var model2 = monolithModels2.Single(x => x.Id == model1.MonolithModel2Id);
            yield return _polyglotModelMapper.Map(model1, model2);
        }
    }
}
```

Figure 8. An example of the simplified record creation class.

4.1.5. Data Validation

The purpose of the fifth step is to create automatic data validation. Transformed data have to be validated before releasing it into production. In the first sub-step, test cases have to be created based on the functional requirements and data in the mainframe monolith database. The second sub-step is to create a test engine that has to be able to execute created test cases in the previous sub-step. The purpose of the last three sub-steps is to execute the test cases and make amendments if needed (Figure 9). The step is finished only when all of the test cases are passed.

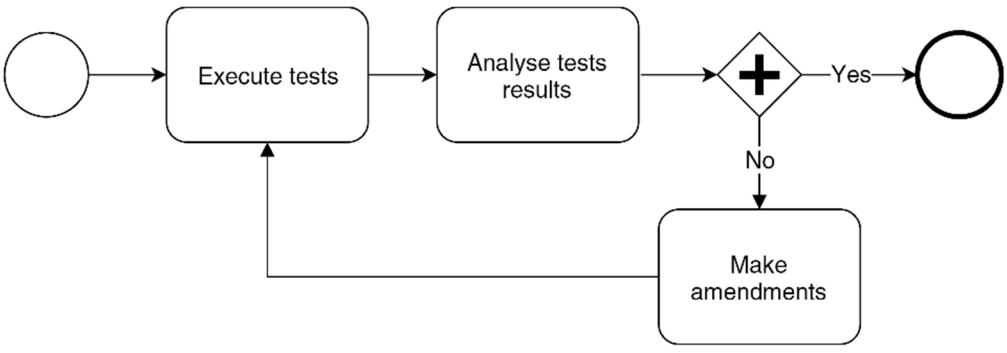

Figure 9. The test case execution.

For example, the functional requirements for the data records of *PolyglotModel* defined in Figure 7 are: *read*, *create*, *update*, and *delete*. Four test cases have to be created in

order to validate the data integrity and persistence, one test case for one functional requirement. The first functional requirement is the possibility to read the data. In this example, it is the possibility of reading data records of *PolyglotModel*. Two main criteria have to be verified. First, in each data record of *PolyglotModel*, all fields have to be mapped correctly and the data have to be consistent. Second, the multi-model polyglot persistence has to contain the same number of records as the data table *Model1Table* in the monolith database. Figure 10 contains an example of the possible records. The *monolithModel1Record* represents a record of the data table *Model1Table* in the monolith database, the *monolithModel2Record* represents a record of the data table *Model2Table* in the monolith database, and the *polyglotModelRecord* represents a record of the *PolyglotModel* in multi-model polyglot persistence. The test case for functional requirement *read* has to verify that all fields that exist in the *MonolithModel1* and *MonolithModel2* models also exist in *PolyglotModel* and the values are the same. For example, the *PropertyB* value in *monolithModel2Record* should be the same as the PropertyB value in *polyglotModelRecord.* To verify that all records were transformed to multi-model polyglot persistence during step 4, the test case has to be executed as many times as the *Model1Table* table has records.

```
var monolithModel1Record = new MonolithModel1
{
   Id = 1,
   PropertyA = "PropertyAValue",
   MonolithModel2Id = 2
};

var monolithModel2Record = new MonolithModel2
{
   Id = 2,
   PropertyB = "PropertyBValue",
   PropertyC = "PropertyCValue"
};
```

```
var polyglotModelRecord = new PolyglotModel
{
   Id = 1,
   PropertyA = "PropertyAValue",
   Child = new PolyglotChildModel
   {
      Id = 2,
      PropertyB = "PropertyBValue",
      PropertyC = "PropertyCValue"
   }
};
```

Figure 10. An example of the data records in the monolith database and multi-model polyglot persistence.

Three more test cases have to be created to validate *create*, *update* and *delete* functional requirements. The test case for *create* should try to create a new record of *PolyglotModel* and verify that the record is actually created and that all of the fields are filled correctly. The test case for *update* should try to update all of the value fields in a record of *PolyglotModel* and verify that all of them are updated correctly. Finally, the test case for *delete* should try to delete a record of *PolyglotModel* and verify that it was actually deleted.

4.1.6. Release and Deployment

The aim of the last step is to release and deploy the developed microservice and the amended monolith application into a production environment. The first sub-step is to deploy the developed microservice into the production environment. It includes all the technical peculiarities needed to deploy and run the microservice as a Docker container. All preparation actions such as the creation of the OpenShift project, creation of the CI/CD pipeline, etc., should be conducted during step 3. In essence, it should be just a run of the CD part in the CI/CD pipeline.

During the next three sub-steps, the monolith application should start using the microservice as a data source instead of the monolith database (Figure 11). At first, the monolith application has to be stopped and data transformation has to be executed with the application created in the fourth step. Then, the monolith code has to be amended to use the microservice created in the third step instead of the monolith database and be deployed into the production environment. The precondition for the deployment of the amended monolith application is successful data transformation. The application created

in the fifth step has to be executed to verify that the data are valid after transformation. If data validation fails, the release operation must be stopped and the old version of the monolithic program used until the cause of the failed validation is identified and eliminated.

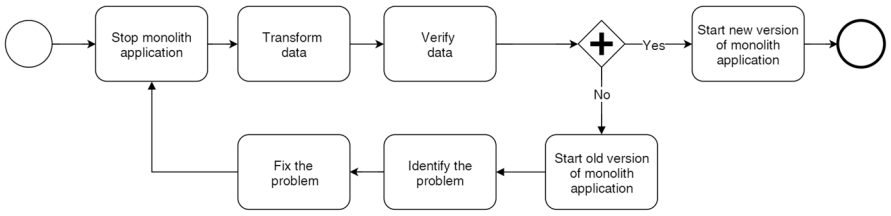

Figure 11. Release and deployment execution.

The fifth sub-step is hyper care, during which the domain and IT experts have to give hyper attention to newly released software and fix any last errors in case they appear. The last sub-step is decommissioning a not used monolith database.

Another important aspect that has to be taken into account during this step is to ensure that only the monolith application is using its database. In the case that other programs are using the same database, then the migration process has to be coordinated with the teams who are responsible for those programs to ensure that after migration, the database is not used by any program.

4.2. Criteria

The ISO/IEC 25012:2008 standard quality attributes were used to evaluate and compare the data quality of the proposed multi-model polyglot persistence model and the existing monolith mainframe persistence model. The quality attributes used in the evaluation were: Accuracy, Completeness, Consistency, Credibility, Correctness, Accessibility, Compliance, Confidentiality, Efficiency, Precision, Traceability, Understandability, Availability, Portability, and Recoverability.

4.3. Multi-Model Polyglot Database Software

ArangoDB is an open-source multi-model polyglot persistence system that implements a data model integrating the document, graph, and key-value models with one database core. It supports the transactions, partitioning, and replication. ArangoDB has its own query language, AQL, which allows for joins, operations on graphs, iterations, filters, projections, ordering, grouping, aggregate functions, union, and intersection. The ArangoDB supports all of the ACID properties [57].

4.4. Tools

The ArangoDB database was used as the multi-model polyglot database engine [57]. The microservice that exposes multi-model polyglot persistence was written using the C#.NET5 framework [58]. All coding and testing were conducted using Microsoft Visual Studio IDE and Arango Management Interface [59]. All libraries used in the research were downloaded from NuGet gallery [60]. The experiment was performed on a computer with the following specification: CPU—Core i7 9850H, memory—32 GB RAM, storage—512 GB SSD, and OS—Windows 10 Enterprise. All applications were run on a computer, no external devices or networks were used.

5. Result of Experiment

This section provides the results obtained during the evaluation of the approach of the mainframe monolith database migration to multi-model polyglot persistence based on microservice architecture. The results of each step of the proposed approach are explained in separate subsections: 5.1. Research of an existing monolith application; 5.2.

Data Model Creation; 5.3. Microservice Creation; 5.4. Data Transformation; 5.5. Data Validation; 5.6. Release into Production.

5.1. Analysis of an Existing Monolith Application with a Mainframe Database

The primary function of the standard settlement instructions (SSI) application is to store and provide standard settlement instructions to other information systems across the organization. Standard settlement instructions are used to execute payments between banks and organizations. A simplified model of the SSI application is shown in Figure 12.

The SSI application is implemented with IBM mainframe and Microsoft.NET framework technologies. The data are persisted in 35 tables in the DB2 database and can be accessed and edited through IBM mainframe modules. SSI data are exposed to other information systems across the organization through Rest API, which is implemented with the Microsoft.NET framework.

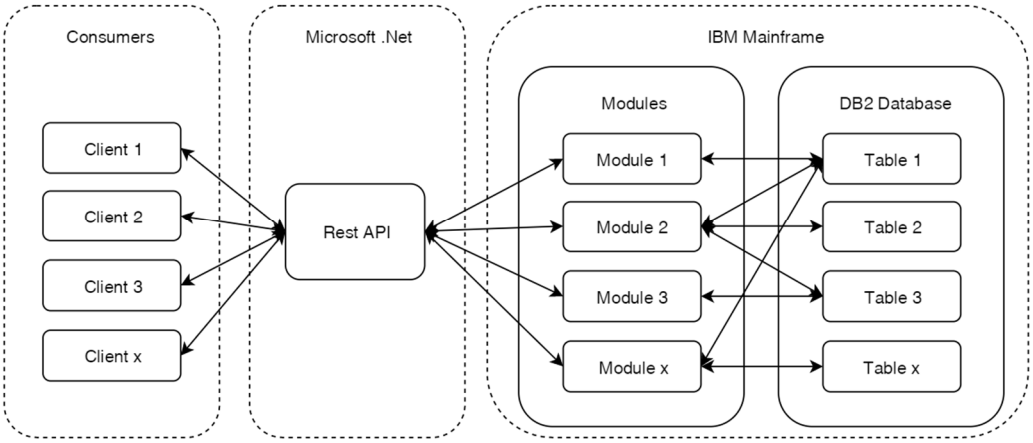

Figure 12. The simplified model of the SSI application.

Based on the top–down approach, functional requirements were gathered in two steps: (1) defining critical functionality by talking with domain experts within the organization, and (2) deep revision of the legacy code. The most important functional requirements gathered during the evaluation are presented in Table 1.

Table 1. The functional requirements of the SSI application.

	Functional Requirements
1.	Ability to view/add/update/delete customers
2.	Ability to view/add/update/delete agreements
3.	Ability to view/add/update/delete standard settlement instruction
4.	Two types of standard settlement instruction: receive and deliver
5.	One customer can have many agreements
6.	One customer can have many confirmation settings
7.	One customer can have one netting settings
8.	An agreement can have many instructions
9.	An agreement can have one account information

5.2. Data Model Development

The aim of this step is to design a new data model that will be used in multi-model polyglot persistence. The creation of a new model process consists of five steps: (1) conceptual design; (2) segmentation design; (3) consistency design; (4) target data model design; and (5) physical design.

5.2.1. Conceptual Design

The aim of the conceptual design step is to translate the identified functional requirements into a conceptual schema. The entity–relationship model is used as a conceptual schema because it is a widely exploited model and allows for a detailed definition of the entities and their relationships on the database. The simplified conceptual database schema of the SSI application is shown in Figure 13. The root element of the system is a customer, which can have one netting agreement and many confirmations and agreements. Netting is an option to merge many payments into one. An agreement is a special contract with a customer, usually for a specific product and currency that have a specific settlement instruction. Each short name can have one account and many receive and deliver instructions. A receive instruction is an instruction for incoming payment and a delivery instruction is an instruction for outgoing payment.

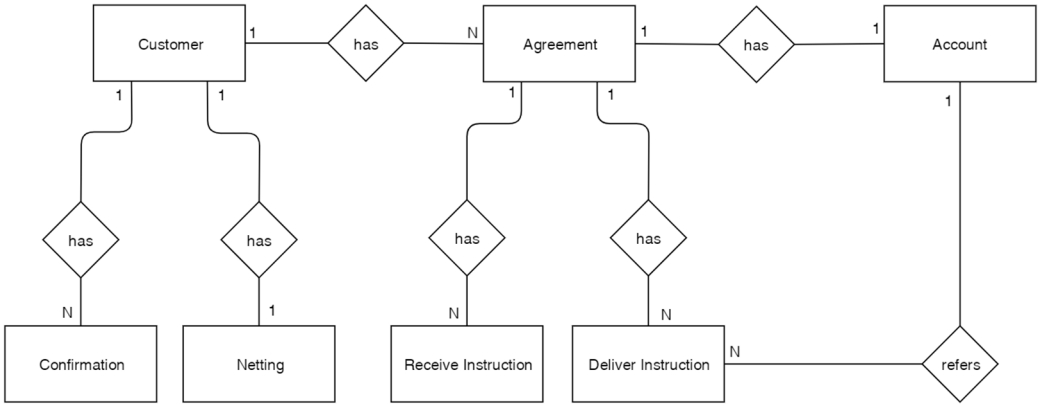

Figure 13. The simplified conceptual schema of the SSI application.

5.2.2. Segmentation Design

The segmentation step identifies independent functional units and defines the borders between them. Segmentation units have to be identified in order to take full advantage of the multi-model polyglot persistence feature, which is the capability of using multiple different data models in the same database. The outcome of this step is defined cut points on the existing data model that can be used to split it into different data models. Any of the segmentation units can be detached from the model and work as an independent system. Segmentation units identified in the simplified conceptual database schema of the SSI application are shown in Figure 14. During the segmentation design sub-step, the SSI application was divided into three independent functional units: customer management, agreement management, and instruction management.

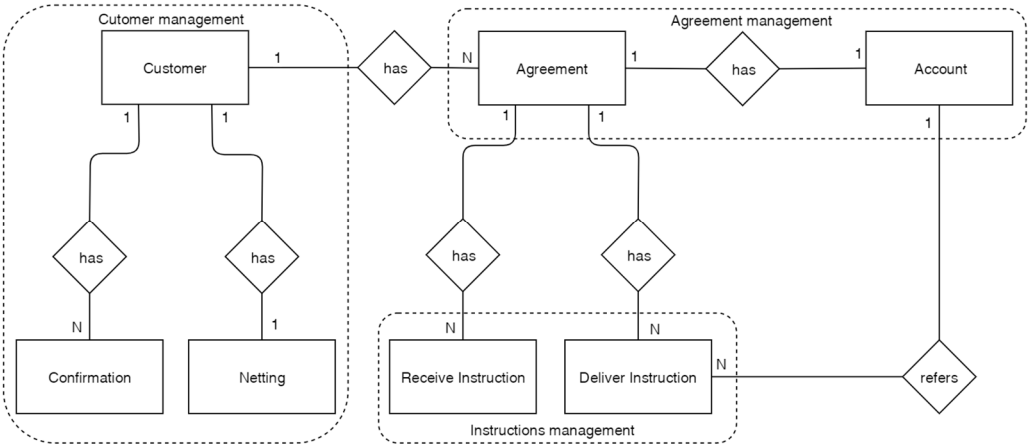

Figure 14. The segmentation units are identified in the simplified conceptual database schema.

5.2.3. Consistency Design

The consistency step ensures consistency of the dataset across all subsystems and allows for data fragmentation. As polyglot supports NoSQL data models, the eventual consistency provided by BASE properties has to be taken into account during the data model creation step. Polyglot persistence does not have to be consistent across the entire database, but some data groups must be consistent to be valid. These groups are called consistency units and play a key role in allowing data fragmentation and horizontal scalability. The consistency unit has to ensure a guarantee that, provided no new updates to an entity are made, all reads of the entity will eventually return the last updated value. An example of one consistency unit in the SSI application is shown in Figure 15. Customer, agreement, and receive instruction comprise a consistency unit, and in the case a query returns the response with different versions of items, we have an inconsistency that may cause a system failure.

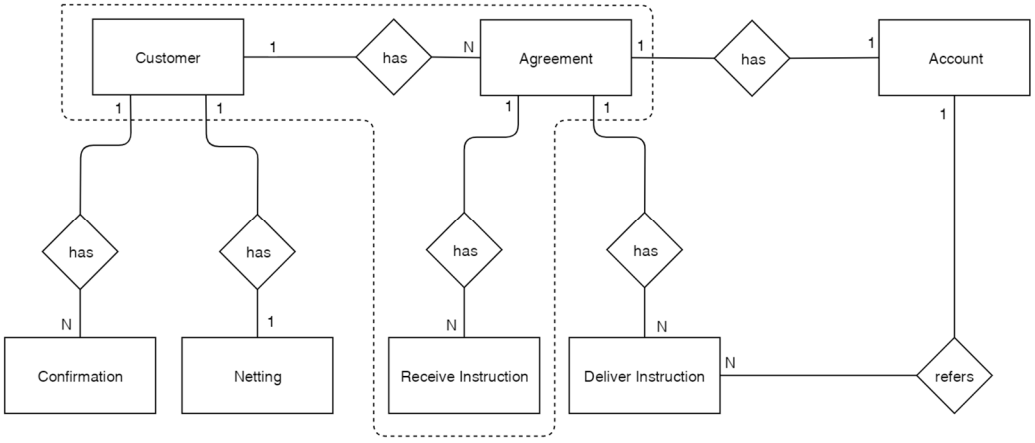

Figure 15. The consistency unit identified in the simplified conceptual database schema.

5.2.4. Target Data Model Design

The target data model step defines the best data model for each segmentation unit. All three subsystems fit into a combination of the key-value and document-oriented data model.

The identified target data model is shown in Figure 16. One customer can have many agreements and each agreement can contain many instructions. Customers, agreements, and instructions are saved as documents in separate collections. The relations between the customers and agreements and relations between the agreements and instructions were defined as collections and also stored in separate collections.

Figure 16. The target data model.

5.2.5. Physical Design

The aim of the physical design step is to implement all peculiarities of the planned to use database, in order to implement the target model. As multi-model polyglot persistence was chosen as data persistence, only one database engine was used. The physical design step is less complex with a multi-model compared to standard polyglot persistence, which is used by many different databases.

In the ArangoDB database, data are stored as documents (JSON format) and each document could be considered a as key–value pair. Documents are grouped into collections. ArangoDB supports two types of collections: document collection and edge collections. Documents are vertices and edges are edges in the context of graphs. Edge collections are used to create relations between documents.

The physical model created during the experiment is shown in Figure 12. It consists of document collection: (1) Customers—to store the customers' data; (2) Agreements—to store the agreements' data; (3) Instructions—to store the instructions' data. To create relations between the documents, two edge collections were introduced: (1) AgreementsInCustomers—to store the relations between a customer and its agreements, (2) InstructionsInAgreement—to store the relations between an agreement and its instructions. The physical data model can be considered as a graph. An example with two customers is shown in Figure 17.

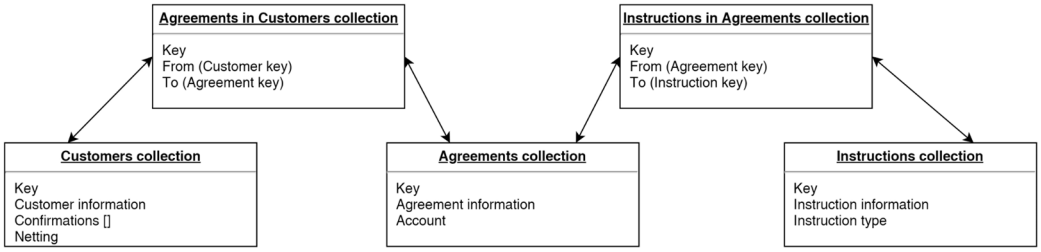

Figure 17. The physical data model.

The physical data model can also be represented as a graph. The example of the data model with two customers where each has one agreement and the first agreement contains two instructions, and the second agreement contains three agreements, is shown in Figure 18.

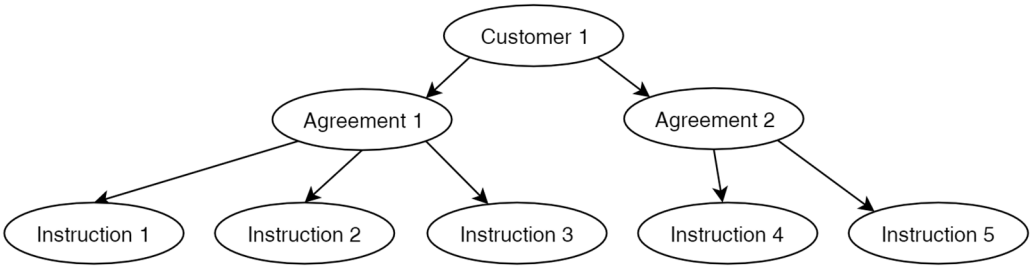

Figure 18. The graphical representation of the physical data model.

5.3. Microservice Development

The simplified model of the built microservice with multi-model polyglot persistence is shown in Figure 19.

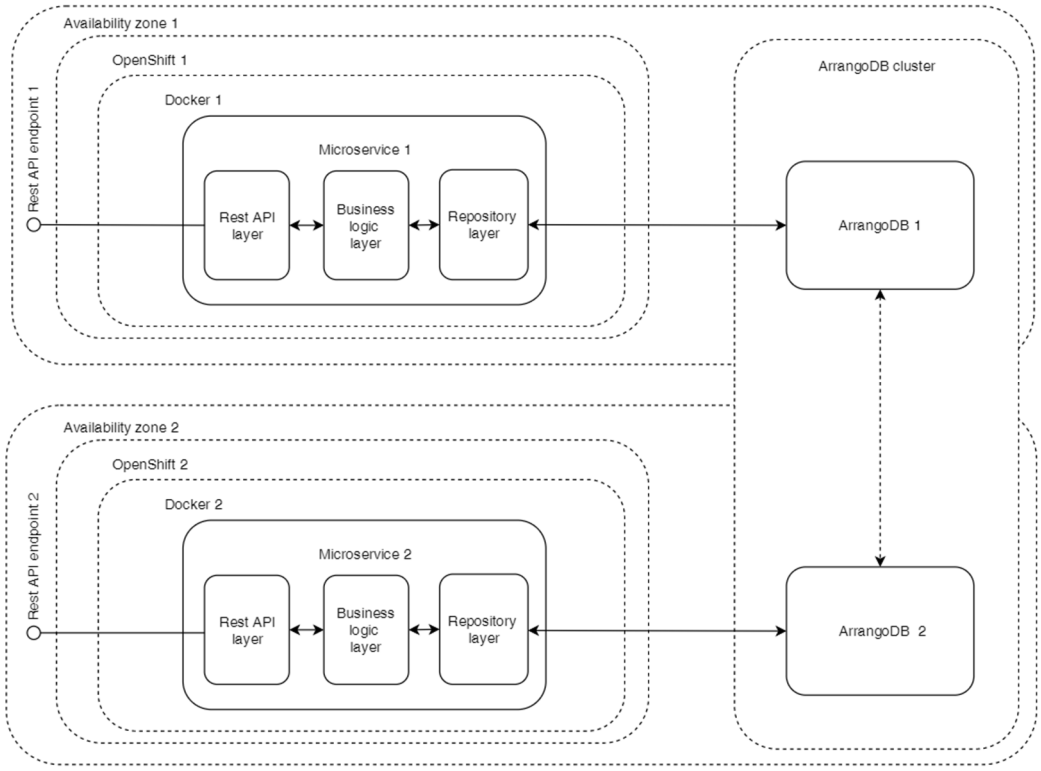

Figure 19. The simplified model of a new SSI application with multi-model polyglot persistence.

The pattern–database as a service was chosen to be used in order to build multi-model polyglot persistence based on microservice architecture. Based on the gathered functional requirements, the application was implemented as a microservice written with the C# programming language within the Microsoft.NET framework. It was deployed to the OpenShift project as a Docker container by the AzureDevOps CI/CD pipeline. Based on the availability and scalability requirements, two separate OpenShift projects were created, each in a separate availability zone. In each availability zone, one Docker container was created with the possibility to scale up on demand automatically. The ArangoDB was used as a multi-polyglot database and its cluster was established with two nodes, one node per availability zone. The security and accessibility were ensured by firewall rules and separate access rights for specific operations.

In the new SSI application, a database and a business logic worked as one unit—the SSI microservice. The data are exposed to the other information systems across an organization via REST API, which is available to new microservices and legacy monolith solutions. The business logic layer interacts with a database through a repository layer that encapsulates the database-specific details. The details on the business logic and database are hidden from the consumers: the only way to manipulate the data is through the REST API by using domain data models.

5.4. Data Transformation

In order to migrate data from a monolith database to a multi-model polyglot persistence database, a data transformation application was written with the C# programming language (Figure 20). The application contained three layers: *extraction*, *transformation*, and *import*. The *Extraction layer* extracts all data from the existing monolith database. Thirty-five repositories and data models were created to extract data from each data table. The *Transformation layer* transforms the extracted data into a data model that is supported by multi-model polyglot persistence. The *Import layer* imports the transformed data into a multi-model polyglot database. The repository layer code from the microservice code base was reused.

The data was extracted from 35 tables in the IBM DB2 database, transformed, and imported into three document collections and two edge collections. To make sure that the data is consistent in both databases, the actual data transformation was conducted during the release and deployment step. The mainframe application was temporarily stopped to transform the data and make all of the amendments needed to use the microservice instead of the existing monolith database.

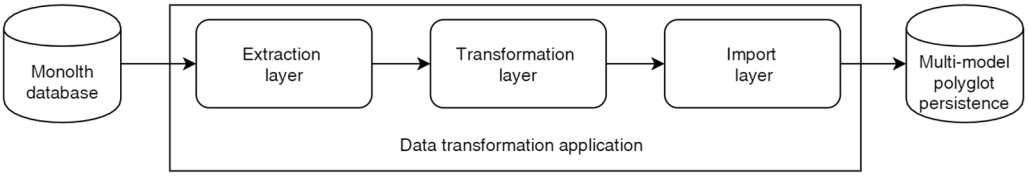

Figure 20. The data transformation from the monolith database to multi-model polyglot persistence.

5.5. Data Validation

Based on specified functional requirements, test cases for data validation were created in a forum of domain experts and IT experts within the organization. The forum consisted of three SSI domain experts, four mainframe software engineers, and four C# software engineers. A test engine was written with the C# programming language to execute automatic data validation (Figure 21) and contained three modules. *The data extraction* module extracts data from the monolith database. The *Test execution* module uses the extracted data to make calls to the Microservice REST API. *The analysis* module compares responses from Microservice REST API and data extracted from the monolith database. For example, the data extraction module extracts all of the existing customers from the monolith database, the test execution module one by one requests customer data from Microservice REST API, and the analysis module validates that all customers exist in multi-model polyglot persistence.

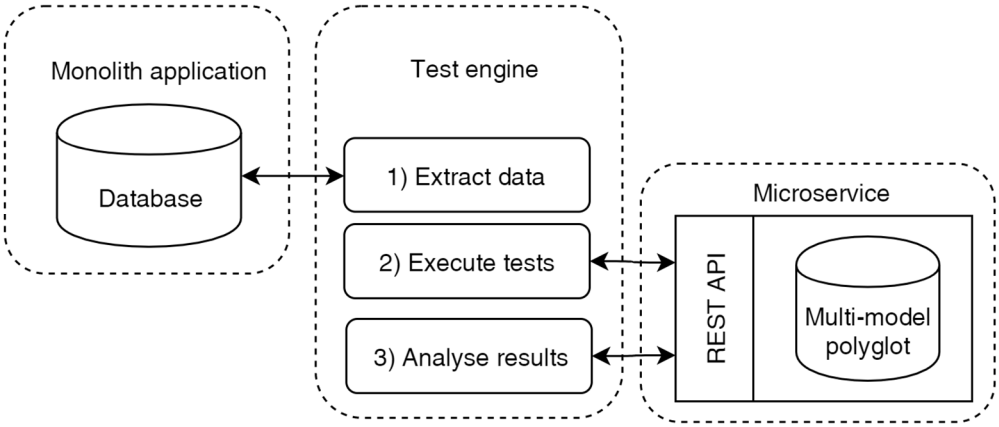

Figure 21. The automatic data validation process.

5.6. Release and Deployment

The first sub-step during the release is the deployment of microservice to the production environment. The microservice was deployed to the on premise cloud as a Docker container to OpenShift [61–62]. Four instances of microservice were distributed between two microsegments, two instances in each microsegment. Each microsegment was in different data centers. Microservice deployment into the cloud schema ensures a high resilience and availability level (Figure 22). Kubernetes ensure resilience for containers in each microsegment and the distribution between two microsegments ensures high availability [63]. A load balancer provides one point for the clients to the REST API. The continuous

integration (CI) and continuous deployment (CD) pipeline was created in Azure DevOps [64].

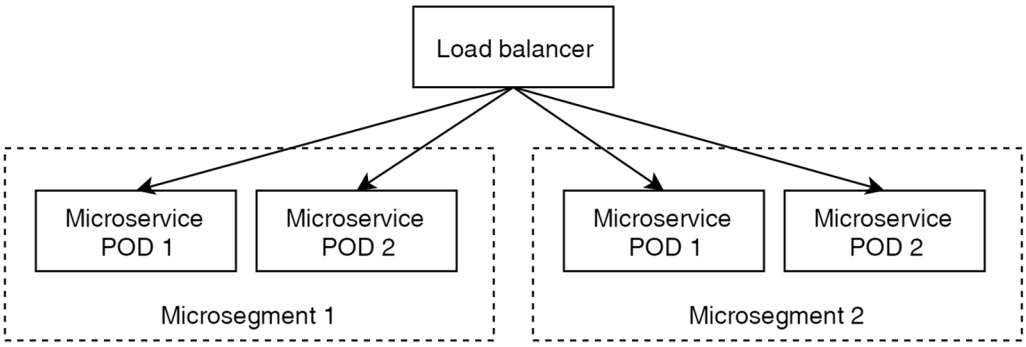

Figure 22. Microservice deployment into the cloud schema.

During the next sub-step, the monolith mainframe application was stopped and data transformation and validation were executed with separate applications. Then, the code of the existing monolith application was amended to use a microservice instead of a monolith database, and all of the SQL queries were changed to calls to the microservice exposed REST API. The new version of the mainframe monolith application was released into production and the hypercare period started. The switch between databases was executed during the weekend. The downtime did not have any impact because the monolith mainframe application was only used on working days. Once the hypercare was over, the legacy monolith mainframe database was decommissioned.

6. Evaluation of the Data Quality of the Proposed Microservice with Multi-Model Polyglot Persistence

Data quality is a key component of the quality and usefulness of information systems. The effectiveness of business processes directly depends on the quality of the data. This section provides the results of the evaluation and comparison of the ISO/IEC 25012:2008 standard quality attributes between the monolith mainframe application and microservice with multi-model polyglot persistence. Each quality attribute was evaluated and graded on a scale from 1 to 5 for each application. A lower value showed a lower quality and a higher value showed a higher quality. Descriptions of the used evaluation grades are provided in Table 2.

Table 2. The description of the used evaluation grades.

Value	Description
1	Lowest quality
2	Low quality
3	Average quality
4	High quality
5	Highest quality

The evaluation was conducted in a forum of domain experts and IT experts within the organization. The forum consisted of three SSI domain experts, four mainframe software engineers, and four C# software engineers. Proof-of-concept of a microservice with multi-model polyglot persistence was compared to an existing monolith mainframe application going through the list of questions for each quality attribute. There were 150 questions, 10 questions for each quality attribute. Each question had to be applied to both applications. Questionnaires were constructed in a way to make answering possible for staff with low IT knowledge levels (domain experts). For example, one of the questions to

evaluate understandability is: "*Is the data model easily understandable*?" Experts had to choose an answer from five possible options: *strongly disagree* (1 point), *disagree* (2 points), *neither agree nor disagree* (3 points), *agree* (4 points), *strongly agree* (5 points). Kendall's W coefficient of concordance was used to assess the agreement among the experts [65]. The coefficient value was 0.76, which indicates a relatively high level of agreement between the experts. If the test statistic W was 1, then all of the survey respondents were unanimous, and each respondent has assigned the same order to the list of concerns. If W was 0, then there was no overall trend of agreement among the respondents, and their responses may be regarded as essentially random. Intermediate values of W indicate a greater or lesser degree of unanimity among the various responses.

The final results of the evaluation and comparison are shown in Table 3. The final value for each quality attribute is an average of the experts' values rounded to the nearest whole number.

Table 3. The results of the evaluation and comparison of the ISO/IEC 25012:2008 standard quality attributes between the monolith mainframe and microservice applications.

Quality Attribute	Monolith	Microservice
Accuracy	5	5
Completeness	5	5
Consistency	3	5
Credibility	5	5
Correctness	4	4
Accessibility	4	4
Compliance	5	5
Confidentiality	5	5
Efficiency	4	4
Precision	5	5
Traceability	5	5
Understandability	3	5
Availability	2	4
Portability	1	5
Recoverability	4	4

Most of the ISO/IEC 25012:2008 standard quality attributes such as accuracy, completeness, credibility, correctness, accessibility, compliance, confidentiality, efficiency, precision, traceability, and recoverability were the same for both applications, but microservice with multi-model polyglot persistence showed better results in consistency, understandability, availability, and portability.

1. *Consistency*—a microservice with multi-model polyglot persistence provides strong data consistency and uses three methods to ensure consistency: eventual, immediate, and OneShard (highly available, fault-tolerant deployment mode with ACID semantics) while mainframe monolith data persistence only uses an immediate method to ensure consistency. In addition to database supported consistency methods, the business layer of microservice ensures that consumers operate only with consistent data models. Consumers, through REST API, can only manipulate data at the domain level as they are not aware of the database schema details and do not have access rights to access it directly.
2. *Understandability*—a new data model with five collections instead of 35 tables that were used in the mainframe application is simpler and easier to understand. The relations between entities are represented as a graph, which is a great help in improving the readability. The AQL query language used to query polyglot persistence is

considered as a human-readable query language and increases understandability compared to the SQL query language used in mainframe application.

3. *Availability*—the biggest advantage of microservice with polyglot persistence in terms of availability is that it supports many resilient deployment modes to meet the different needs of a different project. Active failover deployment is used for smaller projects with fast asynchronous replication from the leading node to passive replicas. OneShard deployment is used for multi-node clusters with synchronous replication from the leading node it provides. A synchronously-replicating cluster technology allows it to scale elastically with the applications and all data models. The last but not least feature of multi-model polyglot persistence is the support for datacenter to datacenter replication.
4. *Portability*—while the mainframe requires a very specific infrastructure to run an application, a microservice with multi-model polyglot persistence can be installed on all main operating systems (Linux, Windows and macOS) and can be deployed to a private or public cloud.

It can be summarized that by using the proposed migration approach, it is possible to execute the migration from the monolith mainframe persistence model to the multi-model polyglot persistence model without losing data quality. Eleven of fifteen ISO/IEC 25012:2008 standard quality attributes were the same for both models and four were even better for the multi-model polyglot persistence model. It must also be noted that the results could be different for different monolith applications.

7. Discussions

This section provides the results of the comparison between the authors' proposed monolith database migration approach and the alternative technique for extracting microservices from monolith enterprise systems [39]. The authors have chosen to compare its approach with A. Levcovitz et al.'s proposed technique, because the other authors' proposed methods do not provide or provide very little details on how to adopt data storage to microservice architecture during the migration from monolith to microservice architecture [4,10–24]. The advantages and disadvantages of the authors' proposed approach compared with the alternative proposed technique are shown in Table 4. The sign "+" in Table 3 means that criteria is an advantage, while the sign "−" means that the criterion is a disadvantage or there is no mention of this criteria. We put the final grades based on common agreements between the authors of this paper.

Table 4. The results of the comparison of the authors' proposed monolith database migration approach and the technique proposed by A. Levcovitz et al.

	Criteria	Authors	Alternative
1.	Possible improvement of quality of consistency, understandability, availability, and portability	+	−
2.	Availability to used different data models for different data structures	+	−
3.	Database adaptation to microservice architecture	+	−
4.	Extensive business experts involvement into migration process	−	+
5.	Ability to divide database per microservice	−	+

Three advantages of our proposed migration approach were identified. First, it allowed us to improve the quality of consistency, understandability, availability, and portability, while A. Levcovitz et al.'s proposed technique does not provide any information about improved quality after migration. Second, it migrates the data store to multi-model polyglot persistence, which allows for the use of different data models for different data structures and better utilizes the advantages of the microservice architecture. While the alternative technique divides

the monolith database by tables and reuses the same legacy relational data store. Third, it allows one to extract the database from the monolith application and adopt it to the microservice architecture. Data are exposed through the REST API and are accessible not only within the microservice ecosystem but also for the legacy monolith application. This allows us to conduct migration gradually and combine other migration methods for code decomposition.

Two disadvantages of our proposed migration approach were identified. First, our proposed migration approach requires extensive involvement of business experts to create a conceptual diagram and identify functional requirements. On the other hand, an alternative technique can be executed without the involvement of business experts. Second, A. Levcovitz et al.'s proposed technique allows one to divide the database per microservice, while our proposed approach extracts the database and converts it to the microservice itself.

In theory, both disadvantages of the proposed approach could be addressed, but a deeper investigation is needed. A hypothetical possible solution to reduce the extensive involvement of business analysts in the first step could be a program that would automatically analyze the existing monolithic program and its database and provide a list of possible functional requirements and an optimal data model. A potential solution for the second disadvantage could be an additional step or an extension of the first step in the proposed approach. The purpose of additional action would be to identify different business domains in the current data model and decompose it into as many data models as business domains are identified. For each identified business domain, steps 2–6 of the proposed approach should be applied separately.

8. Conclusions

In this paper, we proposed and evaluated the approach of monolith database migration into multi-model polyglot persistence based on microservice architecture. As a proof-of-concept, the migration from an existing monolith mainframe application to a microservice was conducted. Existing and new applications were evaluated and compared based on the quality attributes defined in the ISO/IEC 25012:2008 standard.

Based on the results of the research, it can be stated that the proposed approach can be applied to the migration from a monolith mainframe persistence to a microservice architecture based multi-model polyglot persistence, and multi-model polyglot can be used as storage persistence for microservices. By using the proposed migration approach, it is possible to improve the quality of the consistency, understandability, availability, and portability attributes. On the other hand, every monolith application could have a significant architectural difference. Therefore, the proposed approach must be adopted based on the specific application.

The next step in this work is to investigate the possibility of the automation of the proposed migration approach. Currently, all steps of the proposed migration approach have to be implemented manually for each application, which makes it a very slow and expensive process. A tool that could automatically execute migration or part of it could dramatically reduce the time and cost.

Author Contributions: **C**onceptualization, J.K., D.M. and D.K.; methodology, J.K.; software, J.K.; validation, J.K., D.M. and D.K.; formal analysis, J.K.; investigation, J.K.; resources, J.K.; data curation, J.K.; writing—original draft preparation, J.K.; writing—review and editing, J.K.; visualization, J.K.; supervision, D.M.; project administration, J.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. International Data Corporation Web Site. Availaible online: https://www.idc.com/ (accessed on 30 October 2021).
2. Newman, S. *Monolith to Microservices. Evolutionary Patterns to Transform Your Monolith*, 1st ed.; 1005 Gravenstein Highway North; O'Reilly Media: Sebastopol, CA, USA, 2019; pp. 1–272.
3. Columbus, L. IDC Top 10 Predictions for Worldwide IT. 2019. Available online: https://www.forbes.com/sites/louiscolumbus/2018/11/04/idc-top-10-predictions-for-worldwide-it-2019/?sh=5e55583c7b96 (accessed on 30 October 2021).
4. Francesco, P.D.; Lago, P.; Malavolta, I. Migrating towards microservice architectures: An industrial survey. In Proceedings of the International conference on software architecture (IEEE), Seattle, WA, USA, 30 April–4 May 2018; pp. 29–2909. https://doi.org/10.1109/ICSA.2018.00012.
5. Knoche, H.; Hasselbring, W. Using Microservices for Legacy Software Modernization. *IEEE Softw.* **2018**, *35*, 44–49. https://doi.org/10.1109/MS.2018.2141035.
6. Wang, Y.; Kadyala, H.; Rubin, J. Promises and Challenges of Microservices: An Exploratory Study. *Empir. Softw. Eng.* **2020**, *26*, 63. https://doi.org/10.1007/s10664-020-09910-y.
7. Wolfart, D.; Assunção, W.; Silva, I.; Domingos, D.; Schmeing, E.; Villaca, G.; Paza, D. Modernizing Legacy Systems with Microservices: A Roadmap. *EASE* **2021**, *2021*, 149–159. https://doi.org/10.1145/3463274.3463334.
8. Beni, E.H.; Lagaisse, B.; Joosen, W. Infracomposer: Policy-driven adaptive and reflective middleware for the cloudification of simulation & optimization workflows. *J. Syst. Archit.* **2019**, *95*, 36–46. https://doi.org/10.1016/j.sysarc.2019.03.001.
9. Mohamed, D.; Mezouari, A.; Faci, N.; Benslimane, D.; Maamar, Z.; Fazziki, A. A multi-model based microservices identiftcation approach. *J. Syst. Archit.* **2021**, *118*, 102200. https://doi.org/10.1016/j.sysarc.2021.102200.
10. Azevedo, L. G.; Ferreira, R. S.; Silva, V. T.; Bayser, M.; Soares, E. F. de S.; Thiago, R. M. Geological Data Access on a Polyglot Database Using a Service Architecture. In Proceedings of the XIII Brazilian Symposium on Software Components, Architectures, and Reuse, Salvador, Brazil, 23–27 September 2019; pp. 103–112. https://doi.org/10.1145/3357141.3357603.
11. Cruz, P.; Astudillo, H.; Hilliard, R.; Collado, M. Assessing Migration of a 20-Year-Old System to a Micro-Service Platform Using ATAM. In Proceedings of the 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), Hamburg, Germany, 25–26 March 2019; pp. 174–181. https://doi.org/10.1109/ICSA-C.2019.00039.
12. Gouigoux, J.P.; Tamzalit, D. From Monolith to Microservices: Lessons Learned on an Industrial Migration to a Web Oriented Architecture. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 5–7 April 2017; pp. 62–65. https://doi.org/10.1109/ICSAW.2017.35.
13. Hasselbring, W.; Steinacker, G. Microservice Architectures for Scalability, Agility and Reliability in E-Commerce. In Proceedings of the 2017 IEEE International Conference on Software Architecture Workshops (ICSAW), Gothenburg, Sweden, 5–7 April 2017; pp. 243–246. https://doi.org/10.1109/ICSAW.2017.11.
14. Krylovskiy, A.; Jahn, M.; Patti, E. Designing a Smart City Internet of Things Platform with Microservice Architecture. In Proceedings of the 2015 3rd International Conference on Future Internet of Things and Cloud, Rome, Italy, 24–26 August 2015; pp. 25–30.
15. Lotz, J.; Vogelsang, A.; Benderius, O.; Berger, C. Microservice Architectures for Advanced Driver Assistance Systems: A Case-Study. In Proceedings of the 2019 IEEE International Conference on Software Architecture Companion (ICSA-C), Hamburg, Germany, 25–26 March 2019; pp. 45–52. https://doi.org/10.1109/ICSA-C.2019.00016.
16. Esposte, A.M.; Kon, F.; Costa, F.M.; Lago, N. InterSCity: A Scalable Microservice-Based Open Source Platform for Smart Cities. In Proceedings of the 6th International Conference on Smart Cities and Green ICT Systems, Porto, Portugal, 22–24 April 2017; pp. 35–46. https://doi.org/10.5220/0006306200350046.
17. Mazzara, M.; Dragoni, N.; Bucchiarone, A.; Giaretta, A.; Larsen, S.T.; Dustdar, S. Microservices: Migration of a Mission Critical System. *IEEE Trans. Serv. Comput.* **2018**, *14*, 1464–1477.
18. Singhal, H.; Saxena, A.; Mittal, N.; Dabas, C.; Kaur, P. Polyglot Persistence for Microservices-Based Applications. *Int. J. Inf. Technol. Syst. Approach* **2021**, *14*, 17–32. https://doi.org/10.4018/IJITSA.2021010102.
19. Carrasco, A.; Bladel, B.; Demeyer, S. Migrating towards microservices: Migration and architecture smells. In Proceedings of the 2nd International Workshop on Refactoring, Montpellier, France, 4 September 2018; pp. 1-6. https://doi.org/10.1145/3242163.3242164.
20. Carvalho, L.; Garcia, A.; Assunção, W.; Mello, R.; de Lima, M.J. Analysis of the criteria adopted in industry to extract microservices. In Proceedings of the 2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry (CESI) and 6th International Workshop on Software Engineering Research and Industrial Practice (SER&IP), Montreal, QC, Canada, 28 May 2019; pp. 22–29. https://doi.org/10.1109/CESSER-IP.2019.00012.
21. Mazlami, G.; Cito, J.; Leitner, P. Extraction of Microservices from Monolithic Software Architectures. In Proceedings of the 2017 IEEE International Conference on Web Services (ICWS), Honolulu, HI, USA, 25–30 June 2017; pp. 524–531. https://doi.org/10.1109/ICWS.2017.61.
22. Fan, C.; Ma, S. Migrating Monolithic Mobile Application to Microservice Architecture: An Experiment Report. In Proceedings of the 2017 IEEE International Conference on AI & Mobile Services (AIMS), Honolulu, HI, USA, 25–30 June 2017; pp. 109–112. https://doi.org/10.1109/AIMS.2017.23.
23. Furda, A.; Fidge, C.; Zimmermann, O.; Kelly, W.; Barros, A. Migrating Enterprise Legacy Source Code to Microservices: On Multitenancy, Statefulness, and Data Consistency. *IEEE Softw.* **2018**, *35*, 63–72.

24. Mishra, M.; Kunde, S.; Nambiar, M. Cracking the Monolith: Challenges in Data Transitioning to Cloud Native Architectures. In Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings, Madrid, Spain, 24–28 September 2018; pp. 1–4. https://doi.org/10.1109/MS.2017.440134612.
25. Laigner, R.; Zhou, Y.; Salles, M.A.V.; Liu, Y.; Kalinowski, M. Data Management in Microservices: State of the Practice, Challenges, and Research Directions. *Proc. VLDB Endow.* **2021**, *14*, 3348–3361. https://doi.org/10.14778/3484224.3484232.
26. Azevedo, L.G.; Ferreira RD, S.; Silva VT, D.; de Bayser, M.; Soares, E.F.D.S.; Thiago, R.M. Geological Data Access on a Polyglot Database Using a Service Architecture. In Proceedings of the XIII Brazilian Symposium on Software Components, Architectures,
27. Richter, D.; Konrad, M.; Utecht, K.; Polze, A. Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice. In 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Prague, Czech Republic, 25–29 July 2017; pp. 130–137. https://doi.org/10.1109/TSC.2018.2889087.
28. Francesco, P.; Malavolta, I.; Lago, P. Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In International Conference on Software Architecture, Gothenburg, Sweden, 3–7 April 2017; pp. 21–30. https://doi.org/10.1109/ICSA.2017.24.
29. Knoche, H.; Hasselbring, W. Drivers and Barriers for Microservice Adoption—A Survey among Profesionals in Germany. *Enterp. Model. Inf. Syst. Archit. (EMISAJ)–Int. J. Concept. Modeling* **2019**, *14*, 1–35. https://doi.org/10.18417/emisa.14.1.
30. Luz, W.; Agilar, E.; Oliveira, M.S.; Melo, C.E.R.; Pinto, G.; Bonifácio, R. An Experience Report on the Adoption of Microservices in Three Brazilian Government Institutions. In Proceedings of the XXXII Brazilian Symposium on Software Engineering, Sao Carlos, Brazil, 17–21 September 2018; SBES '18; ACM: New York, NY, USA, 2018; pp. 32–41. https://doi.org/10.1145/3266237.3266262.
31. Soldani, J.; Tamburri, D.A.; Van Den Heuvel, W.J. The pains and gains of microservices: A Systematic grey literature review. *J. Syst. Softw.* **2018**, *146*, 215–232. https://doi.org/10.1016/j.jss.2018.09.082.
32. Levcovitz, A.; Terra, R.; Valente, M.T. Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems. In Proceedings of the 3rd Brazilian Workshop on Software Visualization, Evolution and Maintenance (VEM), Belo Horizonte, Brazil, 23 September 2015; pp. 97–104. https://doi.org/10.48550/arXiv.1605.03175.
33. Tozzi, C. 9 Mainframe Statistics That May Surprise You. 2021. Available online: https://www.precisely.com/blog/mainframe/9-mainframe-statistics (accessed on 30 October 2021).
34. Precisely Editor. Why The Mainframe Still Matters in 2021. 2021. Available online: https://www.precisely.com/blog/mainframe/mainframe-still-matters (accessed on 30 October 2021).
35. Henry, A. Mainframe Batch to Microservice. 2018. Available online: https://aws.amazon.com/fr/blogs/apn/how-to-migrate-mainframe-batch-to-cloud-microservices-with-blu-age-and-aws/ (accessed on 30 October 2021).
36. Kazanavičius, J.; Mažeika, D. Analysis of Legacy Monolithic Software Decomposition into Microservices. In Proceedings of Baltic-DB&IS-Forum-DC 2020, Tallin, Estonia, 16–19 June 2020.
37. Kazanavičius, J.; Mažeika, D. Migrating Legacy Software to Microservices Architecture. In Proceedings of the 2019 Open Conference of Electrical, Electronic and Information Sciences (eStream), Vilnius, Lithuania, 25 April 2019; pp. 1–5. https://doi.org/10.1109/eStream.2019.8732170.
38. Brewer, E.A. Towards robust distributed systems. In Proceedings of the Symposium on Principles of Distributed Computing (PODC), Portland, OR, USA, 16–19 July 2000. https://doi.org/10.1145/343477.343502.
39. Khine, P.P.; Wang, Z. A Review of Polyglot Persistence in the Big Data World. *Information* **2019**, *10*, 141. https://doi.org/10.3390/info10040141.
40. Meier, A.; Kaufmann, M. *SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management*; Springer: Wiesbaden, Germany 2019. https://doi.org/10.1007/978-3-658-24549-8.
41. Shah, C.; Srivastava, K.; Shekokar, N.M. A novel polyglot data mapper for an E-Commerce business model. In Proceedings of the 2016 IEEE Conference on e-Learning, e-Management and e-Services (IC3e), Langkawi, Malaysia, 10–12 October 2016; pp. 40–45. https://doi.org/10.1109/IC3e.2016.8009037.
42. Davoudian, A.; Chen, L.; Liu, M. A Survey on NoSQL Stores. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–43.
43. Krishnan, G. IBM Mainframe Database Overview and Evolution of DB2 as Web Enabled Scalable Server. *Datenbank-Spektrum* **2002**, *3*, 6–14.
44. Sharma, V.; Dave, M. SQL and NoSQL Databases. *Int. J. Adv. Res. Comput. Sci. Softw. Eng.* **2012**, *12*, 467-471. https://doi.org/10.1145/3158661.
45. Nayak, A.; Poriya, A.; Poojary, D. Type of nosql databases and its comparison with relational databases. *Int. J. Appl. Inf. Syst.* **2013**, *5*, 16–19.
46. DB-ENGINES. DB-Engines Ranking, 2021. Available online: https://db-engines.com/en/ranking (accessed on 30 October 2021).
47. Zdepski, C.; Bini, T.A.; Matos, S.N. An Approach for Modeling Polyglot Persistence. In Proceedings of the International Conference On Information Systems (ICEIS), Funchal, Madeira, 21–24 March 2018. https://doi.org/10.5220/0006684901200126.
48. Serra, J. What is Polyglot Persistence? 2015. Available online: https://www.jamesserra.com/archive/2015/07/what-is-polyglot-persistence/ (accessed on 30 October 2021).
49. Wiese, L. Polyglot Database Architectures. In Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB, Trier, Germany, 7–9 October 2015.
50. Chawla, H.; Kathuria, H. *Building Microservices Applications on Microsoft Azure*; Apress: Berkeley, CA, USA, 2019.

51. Brown, K.; Bobby, W. Implementation patterns for microservices architectures. In Proceedings of the Pattern Language of Programs Conference, Allerton Park, IL, USA, 24–26 November 2016; p. 35.
52. Ntentos, E.; Zdun, U.; Plakidas, K.; Meixner, S.; Geiger, S. Assessing Architecture Conformance to Coupling-Related Patterns and Practices in Microservices. In *European Conference on Software Architecture*; Springer: Cham, Switzerland, 2020. https://doi.org/10.1007/978-3-030-58923-3_1.
53. Messina, A.; Rizzo, R.; Storniolo, P.; Urso, A. A Simplified Database Pattern for the Microservice Architecture. In Proceedings of the Conference: DBKDA 2016, The Eighth International Conference on Advances in Databases, Knowledge, and Data Applications, Lisbon, Portugal, 26–30 June 2016; pp. 223–233. https://doi.org/10.13140/RG.2.1.3529.3681.
54. Villaça, L.H.; Azevedo, L.G.; Siqueira, S.W. Microservice Architecture for Multistore Database Using Canonical Data Model. In Proceedings of the XVI Brazilian Symposium on Information Systems, São Bernardo do Campo, Brazil, 3–6 November 2020**.** https://doi.org/10.1145/3411564.3411629.
55. Viennot, N.; Lécuyer, M.; Bell, J.; Geambasu, R.; Nieh, J. Synapse: A microservices architecture for heterogeneous-database web applications. In Proceedings of the 10th European Conference on Computer Systems, Bordeaux, France, 21–24 April 2015. https://doi.org/10.1145/2741948.2741975.
56. Rodriguez, J.; Crasso, M.; Mateos, C.; Zunino, A.; Campo, M. Bottom-up and top-down COBOL system migration to Web Services: An experience report. *IEEE Internet Comput.* **2011**, *17*, 44–51.
57. ArangoDB. Available online: https://www.arangodb.com/ (accessed on 19 March 2022).
58. C# Documentation. Available online: https://docs.microsoft.com/en-us/dotnet/csharp/ (accessed on 19 March 2022).
59. Visual Studio. Available online: https://visualstudio.microsoft.com/ (accessed on 19 March 2022).
60. NuGet. Available online: https://www.nuget.org/ (accessed on 19 March 2022).
61. Docker. Available online: https://www.docker.com (accessed on 19 March 2022).
62. OpenShift. Available online: https://www.redhat.com/en/technologies/cloud-computing/openshift (accessed on 19 March 2022).
63. Kubernetes. Available online: https://kubernetes.io (accessed on 19 March 2022).
64. AzureDevOps. Available online: https://azure.microsoft.com/en-us/services/devops (accessed on 19 March 2022).
65. Dodge, Y.; Cox, D.; Commenges, D. *The Oxford Dictionary of Statistical Terms*; Oxford University Press: Oxford, UK, 2006.